hexens × mintify

MAR.25

SECURITY REVIEW REPORT FOR MINTIFY

CONTENTS

- About Hexens
- Executive summary
    - Overview
- Auditing details
- Severity structure
    - Severity characteristics
    - Issue symbolic codes
- Findings summary
- Weaknesses
    - Insufficient verification of staking account in Unstake struct
    - Users may unstake using staking accounts derived using a different season ID
    - Direct theft of funds via claiming, should off-chain services fail
    - Project cannot compile due to missing semicolon
    - Unstake event emissions may contain invalid season IDs
    - Incorrect LEN parameter in Staking and ClaimState
    - Users may stake funds using arbitrary season IDs
    - Unused event_cpi attribute macros
    - Move validation to the account constraints
    - Unused errors
    - Consider more robust logging methods
    - Unused imports
    - Outdated Anchor version with vulnerable third-party crates

ABOUT HEXENS

Hexens is a cybersecurity company that strives to elevate the standards of security in Web 3.0, create a safer environment for users, and ensure mass Web 3.0 adoption.

Hexens has multiple top-notch auditing teams specialized in different fields of information security, showing extreme performance in the most challenging and technically complex tasks, including but not limited to: Infrastructure Audits, Zero Knowledge Proofs / Novel Cryptography, DeFi and NFTs. Hexens not only uses widely known methodologies and flows, but focuses on discovering and introducing new ones on a day-to-day basis.

In 2022, our team announced the closure of a $4.2 million seed round led by IOSG Ventures, the leading Web 3.0 venture capital. Other investors include Delta Blockchain Fund, Chapter One, Hash Capital, ImToken Ventures, Tenzor Capital, and angels from Polygon and other blockchain projects.

Since Hexens was founded in 2021, it has had an impressive track record and recognition in the industry: Mudit Gupta - CISO of Polygon Technology - the biggest EVM Ecosystem, joined the company advisory board after completing just a single cooperation iteration. Polygon Technology, 1inch, Lido, Hats Finance, Quickswap, Layerswap, 4K, RociFi, as well as dozens of DeFi protocols and bridges, have already become our customers and taken proactive measures towards protecting their assets.

EXECUTIVE SUMMARY

OVERVIEW

Hexens conducted a 4-day security assessment of the Mintify Staking Solana program. The program serves as the on-chain component of a Solana-based staking solution, which fulfills orders validated and co-signed by an off-chain API. Note that off-chain components were not in-scope for review.

Hexens identified 13 findings in the Solana program. This included high and medium risk issues which would have allowed malicious stakers to supply mismatched staking accounts, resulting in incorrect order fulfillment and theft of funds. Several lower severity issues and optimizations were also raised, along with further defense-in-depth suggestions for off-chain components. All relevant findings were promptly remediated by the developers.

We can confidently say that the overall security and code quality have increased after completion of our assessment.

AUDITING DETAILS

STARTED
10.03.2025

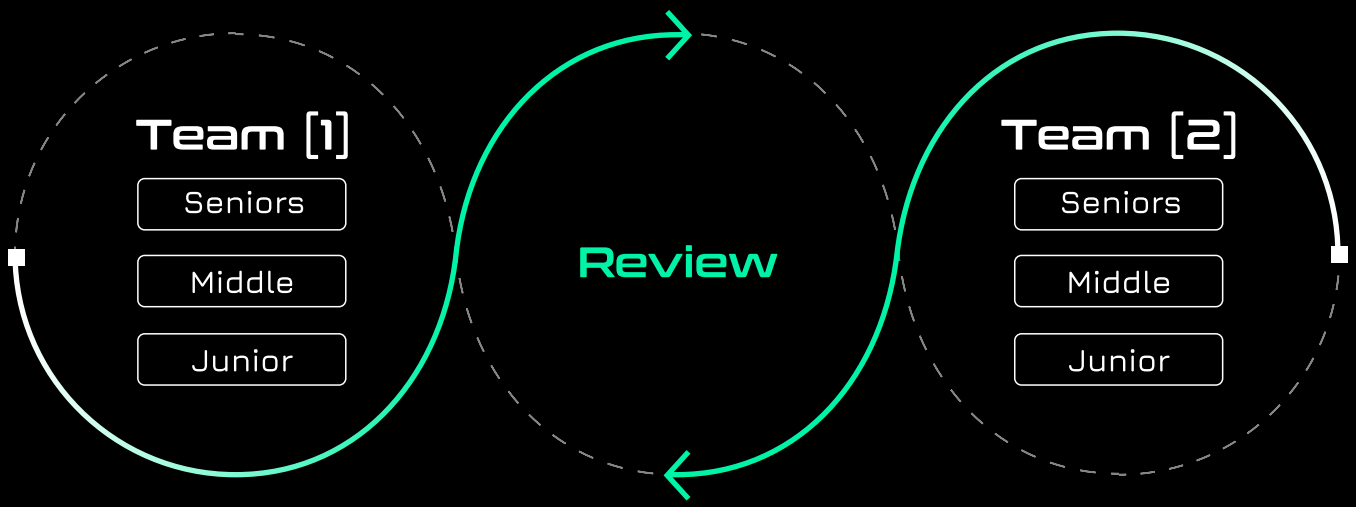
DELIVERED
14.03.2025

Review Led by

HANNAY AL MOHANNA
Senior Security Researcher | Hexens

HEXENS METHODOLOGY

Hexens methodology involves 2 teams, including multiple auditors of different seniority, with at least 5 security engineers. This unique cross-checking mechanism helps us provide the best quality in the market.



SEVERITY STRUCTURE

The vulnerability severity is calculated based on two components

- Impact of the vulnerability
- Probability of the vulnerability

Impact	Probability			
	rare	unlikely	likely	very likely
Low/Info	Low/Info	Low/Info	Medium	Medium
Medium	Low/Info	Medium	Medium	High
High	Medium	Medium	High	Critical
Critical	Medium	High	Critical	Critical

SEVERITY CHARACTERISTICS

Smart contract vulnerabilities can range in severity and impact, and it's important to understand their level of severity in order to prioritize their resolution. Here are the different types of severity levels of smart contract vulnerabilities:

Critical

Vulnerabilities with this level of severity can result in significant financial losses or reputational damage. They often allow an attacker to gain complete control of a contract, directly steal or freeze funds from the contract or users, or permanently block the functionality of a protocol. Examples include infinite mints and governance manipulation.

High

Vulnerabilities with this level of severity can result in some financial losses or reputational damage. They often allow an attacker to directly steal yield from the contract or users, or temporarily freeze funds. Examples include inadequate access control integer overflow/underflow, or logic bugs.

Medium

Vulnerabilities with this level of severity can result in some damage to the protocol or users, without profit for the attacker. They often allow an attacker to exploit a contract to cause harm, but the impact may be limited, such as temporarily blocking the functionality of the protocol. Examples include uninitialized storage pointers and failure to check external calls.

Low

Vulnerabilities with this level of severity may not result in financial losses or significant harm. They may, however, impact the usability or reliability of a contract. Examples include slippage and front-running, or minor logic bugs.

Informational

Vulnerabilities with this level of severity are regarding gas optimizations and code style. They often involve issues with documentation, incorrect usage of EIP standards, best practices for saving gas, or the overall design of a contract. Examples include not conforming to ERC20, or disagreement between documentation and code.

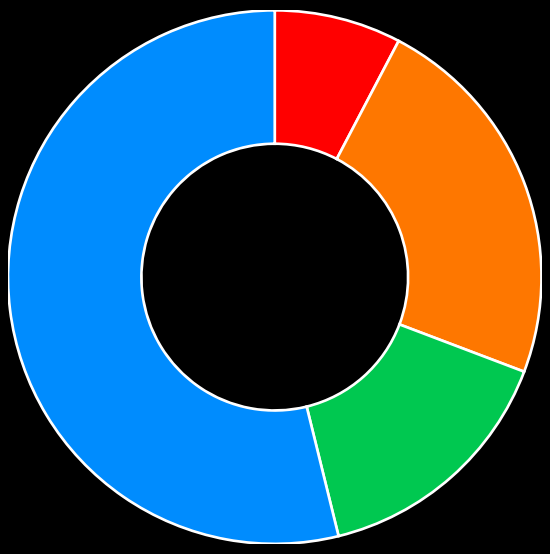
ISSUE SYMBOLIC CODES

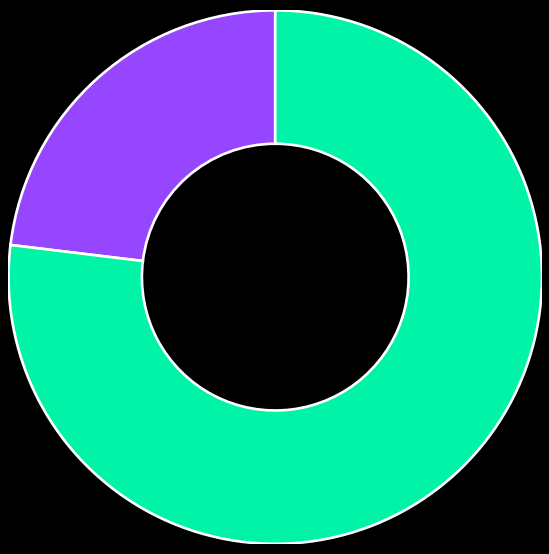
Every issue being identified and validated has its unique symbolic code assigned to the issue at the security research stage. Cause of the vulnerability reporting flow design, some of the rejected issues could be missing.

FINDINGS SUMMARY

Severity	Number of Findings
Critical	0
High	1
Medium	3
Low	2
Informational	7

Total: 13





WEAKNESSES

This section contains the list of discovered weaknesses.

MNTF4-1

INSUFFICIENT VERIFICATION OF STAKING ACCOUNT IN UNSTAKE STRUCT

SEVERITY: High

REMEDIATION:

Consider re-validating the **staking** PDA seeds before it is passed to the **unstake_handler** instruction. Ensure that all implemented modifications to account derivation, and the implications this may have for off-chain systems, are thoroughly tested.

STATUS: Fixed

DESCRIPTION:

The **unstake()** instruction facilitates the withdrawal of staked funds for a user. Within the **unstake.rs** instruction implementation, the call to **unstake_handler()** updates the staked amount by deducting **amount** from **staking.tokens_staked** at line 115. This ensures that the **user** account staked balance is reduced accordingly. The issue arise when the **Unstake** struct does not validate that the **staking** account is derived from **ctx.user**'s public key using specific seeds.

This is because the staking account's seeds are only explicitly validated when the account is initialized via the **initialize_claim_handler** instruction, and not within its struct definition in the **staking.rs** state implementation.

As a result, an attacker can pass an arbitrary **staking** PDA belonging to another user. This allows them to deduct funds from another user's staking

account and subsequently claim tokens from **mint_controller_token_account** into their own wallet.

This issue does not affect other PDAs such as **global** and **claim_controller**, as the static seeds for these accounts are implicitly validated in their state structs, and these accounts cannot be re-initialized after their respective initialization handler instructions have been called due to the **init** macro being used.

Note that the off-chain signing API could not be verified for mitigations against this issue. However due to the fact that the passed **staking** account would be a valid PDA for the program, additional mitigations are recommended at the program level.

MNTF4-6

USERS MAY UNSTAKE USING STAKING ACCOUNTS DERIVED USING A DIFFERENT SEASON ID

SEVERITY: Medium

REMEDIATION:

Consider validating the staking PDA seeds before it is used by the unstake_handler instruction.

Depending on the structure of Mintify seasons, additionally consider using separate per-season user token ATA accounts in subsequent releases of the program, to ensure funds are properly segregated between seasons if necessary. Ensure that these changes are thoroughly reviewed for security issues and compatibility with Mintify season tokenomics.

STATUS: Fixed

DESCRIPTION:

Similar in nature to **MNTF4-1**, it is possible for a user's previously derived staking PDA, derived from a seed which includes a particular seasonID, to be used when unstaking from an unrelated season. Consider the following scenario:

- User stakes with **stake(seasonID: 1, amount: 1, nonce(pubkey, seasonID:1))**, creating a valid PDA#1 from the seeds **user.key()**, **"staking"**, and **seasonID = 1**.
- User stakes again with **stake(seasonID: 2, amount: 100, nonce(pubkey, seasonID:2))**, again causing a valid staking PDA#2 to be derived using the seeds **user.key()**, **"staking"**, and **seasonID = 2**.
- User may then unstake with **unstake(seasonID: 1, amount: 100, nonce(pubkey, seasonID:1))**, but supplying PDA#2.
  - This will also cause an event to be emitted which broadcasts that the user has unstaked an amount exceeding their stake for season 1.

The root causes of this are:

- The same user ATA account **user_token_account** is used to hold all staked funds for all seasons.
- There is no current means to differentiate between staking PDAs derived using different **seasonIDs** when unstaking. While staking account nonces are being validated, they are not being validated against any expected season IDs, so a nonce derived from a previous season ID may be used together with a staking PDA from a different season ID.

This may allow users to game season rewards, by using funds staked as part of one season to gain rewards as part of another season. This may result in considerable losses if the staking requirements for the staked season are lower/significantly different from the season being unstaked from.

Without further mitigations, this may still be possible after remediating **MNTF4-1**, as both staking PDAs would share the same **user.key()** seed.

DIRECT THEFT OF FUNDS VIA CLAIMING, SHOULD OFF-CHAIN SERVICES FAIL

SEVERITY: Medium

REMEDIATION:

Ensure that all relevant off-chain services, particularly those responsible for singing transactions and validating on-chain states, are thoroughly reviewed prior to deployment, with a secondary focus on any modifications to the on-chain program and how they may affect the overall system.

As a defense in depth measure, consider adding additional on-chain verification of token amounts to subsequent releases of the program.

Additionally, ensure best web application/API security practices are adhered to with respect to the signing API, as the API would be considered a high-value target for sophisticated threat actors. In particular, ensure a robust access control and key management systems are applied to the signing API, even if the API is not intended to be exposed publicly.

References:

- OWASP API Security Top 10 - 2023

STATUS: Acknowledged

DESCRIPTION:

The **claim_handler** instruction takes **amount** and **owed** as parameters. Since the token amount is validated using **owed**, without sufficient validation from the off-chain signing API, it would be possible to specify arbitrary amounts, allowing for the withdrawal of all tokens.

It is understood that the off-chain signing API validates all incoming orders against expected token lock periods and amounts, before signing transactions which may then be co-signed and executed by users. This finding is therefore raised primarily for awareness, but with elevated severity due to the potential risk inherent to the design.

MNTF4-3

PROJECT CANNOT COMPILE DUE TO MISSING SEMICOLON

SEVERITY: Medium

REMEDIATION:

Add a semicolon at the end of the affected line.

STATUS: Fixed

DESCRIPTION:

On line 3 of the file **claim.rs**, the semicolon at the end of the import statement is missing. This will prevent the project from compiling.

MNTF4-7

UNSTAKE EVENT EMISSIONS MAY CONTAIN INVALID SEASON IDS

SEVERITY: Low

REMEDIATION:

Validate that the season_id argument can be used to derive the expected corresponding staking account and nonce, ensuring that only valid season IDs for a particular staking account can be used in the same unstake instruction.

As the on-chain program does not currently keep a record of season IDs, this likely needs to be implemented by off-chain validation services. Ensure season ID validation is thoroughly reviewed prior to deployment.

STATUS: Fixed

DESCRIPTION:

The **unstake_handler** instruction allows any value for **season_id** to be used, which is eventually emitted in the **Unstaked** event without prior validation. This **season_id** value does not need to be a valid season ID previously used to derive a **staking** account, and can be any **u64** value. Therefore it is possible to broadcast events with invalid/incorrect season IDs for otherwise valid unstake operations.

INCORRECT LEN PARAMETER IN STAKING AND CLAIMSTATE

SEVERITY: Low

REMEDIATION:

Consider removing the + 1 from both LEN constants if this extra byte is not necessary.

STATUS: Fixed

DESCRIPTION:

The **Staking** and **ClaimState** state structs contain two **u64** values and one **u64** respectively. According to Anchor documentation, the correct account data size should be 16 and 8 bytes accordingly (plus 8 bytes for Anchor's internal discriminator). Currently however, both structures contain an unidentified **+ 1**, effectively occupying 17 and 9 bytes.

References:

- Anchor documentation regarding account space per datatype

USERS MAY STAKE FUNDS USING ARBITRARY SEASON IDS

SEVERITY: Informational

REMEDIATION:

Ensure that the back-end signing API maintains an accurate record of current valid season IDs, and that the API validates season IDs for all incoming orders before it co-signs any transactions.

Alternatively, consider maintaining an on-chain record of valid season IDs via the global state account, and checking all on-chain operations involving season IDs against the known-valid season ID. Only program administrators should be able to set the season ID, similar to the pause_handler instruction for pausing functionality. Note that this would limit the program to only allowing orders from current season IDs, so this behaviour should be verified against Mintify season tokenomics to ensure compatibility.

STATUS: Fixed

DESCRIPTION:

There is no on-chain control preventing users from staking with arbitrary season IDs. If the **season_id** for a particular impending season release is predictable (e.g. sequential **u64**) or otherwise known ahead of time, users would be able to establish stakes for season IDs ahead of the intended season launch date. This may be abused to gain excessive staking rewards based on an unexpectedly long holding period, potentially breaking assumptions around season length and validity.

This may be particularly problematic if certain seasons are expected to have staking or participant caps, as malicious users may issue small stakes across many potential season IDs, or pool together resources to proactively hit staking caps for a particular season as soon as the season launches.

The back-end API was not available for review, therefore it cannot be determined if off-chain validation of season IDs are sufficient protections

against this issue. Therefore, this finding has been raised primarily for awareness.

References:

- Mintify SZN1 reward overview

MNTF4-10

UNUSED EVENT_CPI ATTRIBUTE MACROS

SEVERITY: Informational

REMEDIATION:

It is recommended to remove the **event_cpi** annotation from the affected structs if **emit_cpi!** is not used for event emission. Alternatively, consider using **emit_cpi** to emit events in a more reliable manner than the current **emit!** macro allows. See finding **MNTF4-11** for further information.

STATUS: Fixed

DESCRIPTION:

All events (**Unstaked**, **Claimed**, **Staked**) are emitted via Anchor's default **emit!** macro. However, the affected account structs are configured with the **event_cpi** annotation, despite the **emit_cpi!** macro never being used, or in the case of **withdraw_claim**, no events being emitted. This adds unnecessary overhead, and additional unused **event_authority** PDAs will be included in the program's IDL due to the use of the **event_cpi**.

MOVE VALIDATION TO THE ACCOUNT CONSTRAINTS

SEVERITY: Informational

REMEDIATION:

Consider consistently checking account constraints in account structs using account attribute macros where possible, to ensure invalid accounts are rejected before any business logic is executed.

STATUS: Acknowledged

DESCRIPTION:

Account constraints are inconsistently verified in either instructions or their account structs, such as **global.pausedcheck** and other key checks. This results in decreased code readability and modularity.

UNUSED ERRORS

SEVERITY: Informational

REMEDIATION:

Remove any unused errors.

STATUS: Fixed

DESCRIPTION:

The **ErrorCode** enum in **errors.rs** defines custom error codes. Several of these error variants are unused throughout the codebase.

- StakingAccountAlreadyExists
- InvalidDeployer
- ZeroAmount
- ClaimFailed

CONSIDER MORE ROBUST LOGGING METHODS

SEVERITY: Informational

REMEDIATION:

Review the off-chain architecture and consider implementing either emit_cpi for event emission, or gRPC-based data streaming services.

STATUS: Acknowledged

DESCRIPTION:

Currently, all events (**Unstaked**, **Claimed**, **Staked**) are emitted via Anchor's **emit!** macro. The **emit!** macro directly emits events to the program's event logs in base64. Solana program logs have a maximum size of 10 kB for all instructions within a transaction before logs may be abruptly truncated.

Although these events do not take up a large amount of storage, should enough instructions be included within a single transaction, this may result in off-chain systems being supplied with incomplete data regarding order activity.

Given the program's integration with off-chain APIs, it may be beneficial to use a more robust logging solution to avoid potential log truncation. The benefits and drawbacks of these logging approaches are outlined below:

Keeping the current **emit!** macro:

- Simplest method, does not require additional account overhead maintenance.
- Potential for RPCs to truncate complete log data, especially if many instructions are used within a transaction.

Using **emit_cpi!** macro:

- Via a self-CPI using additional automatically derived accounts, events are included in the transaction's metadata instead of within program

event logs, bypassing the 10kB log limit.

- Additional client overhead would be necessary to parse event data from instruction logs.

Using Geyser/gRPC streams:

- Solana's Geyser interface for validators is a more robust logging solution versus the **emit!** macro variants. It allows for real time granular streaming/subscription of on chain events to back-ends (message brokers, APIs, external databases, etc.), through validators which support Geyser.
- This solution requires the most overhead, and may come with trust assumptions if third-party RPC endpoints are used instead of self-maintained geyser-enabled validators.

References:

- Anchor documentation on emit and emit_cpi
- Anza documentation on the Geyser plugin
- Helius documentation on gRPC streams

MNTF4-12

UNUSED IMPORTS

SEVERITY: Informational

REMEDIATION:

Remove all unused imports before deployment.

STATUS: Fixed

DESCRIPTION:

withdraw_claim.rs imports the unused **Claimed** event, and unused **ClaimState** and **MintController** states.

OUTDATED ANCHOR VERSION WITH VULNERABLE THIRD-PARTY CRATES

SEVERITY: Informational

REMEDIATION:

Update to the most recent Anchor version to address these issues, which is 0.31.0 as of reporting.

STATUS: Fixed

DESCRIPTION:

The project is configured with Anchor version **0.30.1**, and the project uses crates with the following known third-party vulnerabilities:

- curve25519-dalek v3.2.1 - RUSTSEC-2024-0344
- ed25519-dalek v1.0.1 - RUSTSEC-2022-0093


hexens
mintify