Introduction

A security review of the Torus protocol was done, focusing on the security aspects of the smart contracts. This audit was performed by 33Audits & Co with Samuel as the Security Researcher.

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 vulnerabilities. Subsequent security reviews, bug bounty programs, and on-chain monitoring are recommended.

Commit: 12f94a673731cc5872e5c63dd640b6e03a84aa9c - Scope: `TorusEncapsulator`, `TorusPositionNFT`, `Torus`, `TorusUIHelper`

About 33 Audits & Company

33Audits is an independent smart contract security researcher company and development group. We conduct audits a as a group of independent auditors with various experience and backgrounds. We have conducted over 15 audits with dozens of vulnerabilities found and are experienced in building and auditing smart contracts. We have over 4 years of Smart Contract development with a focus in Solidity, Rust and Move. Check our previous work here or reach out on X @33audits.

About Torus

This audit is being performed on the core protocol contracts for Torus. The contracts in scope include `TorusEncapsulator`, `TorusPositionNFT`, `Torus`, and `TorusUIHelper`.

Severity Definitions

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

Impact - The technical, economic, and reputation damage from a successful attack

Likelihood - The chance that a particular vulnerability gets discovered and exploited

Severity - The overall criticality of the risk

Informational - Findings in this category are recommended changes for improving the structure, usability, and overall effectiveness of the system.

Findings Summary

ID	Title	Severity	Status
[C-01]	Reentrancy via delegatecall Leading to Index Desynchronization	Critical	Fixed
[M-01]	SafeTransferFrom missing onERC721Received Callback	Medium	Fixed
[M-02]	ownerOf Should Revert On Non-Existent Tokens Instead Of Returning address(0)	Medium	Fixed
[L-01]	Redundant Context Inheritance in TorusPositionNFT	Low	Fixed
[L-02]	Inconsistent msg.sender vs _msgSender() Usage in TorusPositionNFT	Low	Fixed

Critical

[C-01] Reentrancy via delegatecall Leading to Index Desynchronization

Description

The `TorusEncapsulator` contract's refund mechanism allows for a critical reentrancy attack through delegatecall. During ETH refunds, a malicious user can execute code with `msg.sender = TorusEncapsulator`, enabling them to create unauthorized stakes in TorusCreateAndStake. This permanently breaks the index synchronization system, causing all future position mappings to become corrupted and leading to mass fund loss.

The refund logic in `createTorus()` and `stakeTorus()` makes an external call to potentially malicious contracts after critical state has been set up but before position creation is complete, allowing attackers to manipulate the indexing system.

Snippet

```
// TorusEncapsulator.createTorus (refund occurs before state updates /
indexing)
createAndStake.createTorus{value: msg.value}(power, lengthInDays);
if (msg.value > 0) {
    uint256 ethBalanceAfter = address(this).balance;
    uint256 refund = ethBalanceAfter - ethBalanceBefore;
    if (refund > 0) {
        (bool ok, ) = payable(msg.sender).call{value: refund}("");
        require(ok, "refund fail");
    }
}
// ... then updates nextStakeIndex/positions
```

```
// TorusEncapsulator.stakeTorus (same ordering issue)
createAndStake.stakeTorus{value: msg.value}(torusAmount, stakingDays);
if (msg.value > 0) {
    uint256 ethBalanceAfter = address(this).balance;
    uint256 refund = ethBalanceAfter - ethBalanceBefore;
    if (refund > 0) {
        (bool ok, ) = payable(msg.sender).call{value: refund}("");
        require(ok, "refund fail");
    }
}
// ... then updates nextStakeIndex/positions
```

Impact

- **High Impact**: Permanently corrupts the accounting and indexing of the protocol
- **High Likelihood**: Attack can be executed during any refund operation

Recommendations

- Move refund logic to the very end of functions after all state updates

Example snippet

```
// Checks-Effects-Interactions pattern; refund last
function createTorus(...) external nonReentrant returns (uint256
positionId) {
    // 1) Checks
    // 2) Effects (all state updates first)
    positionId = _createPosition(...);

    // 3) Interactions (ETH refund last)
    uint256 refundAmount = address(this).balance -
expectedPostStateBalance;
    if (refundAmount > 0) {
        (bool success, ) = payable(msg.sender).call{value: refundAmount}
("");
        require(success, "REFUND_FAILED");
    }
}
```

Status

Fixed - The developer has moved the refund to be the final action for both functions.

Medium

[M-01] SafeTransferFrom missing onERC721Received Callback

Description

The `TorusPositionNFT` contract's `safeTransferFrom` functions are **NOT actually safe** - they're missing the critical `onERC721Received` callback check that prevents NFTs from being permanently locked in contracts that can't handle them. The functions just call regular `transferFrom` without the safety checks.

Snippet

```
function safeTransferFrom(address from, address to, uint256 tokenId)
external override {
    transferFrom(from,to,tokenId);
}
function safeTransferFrom(address from, address to, uint256 tokenId, bytes
calldata) external override {
    transferFrom(from,to,tokenId);
}
```

Impact

- **Medium Impact**: NFTs can be permanently locked in contracts, causing complete loss
- **Medium Likelihood**: Standard ERC721 non-compliance issue

Recommendations

- Follow the ERC721 standard for `SafeTransferFrom` to include the `onERC721Received` check
- If `to` is a smart contract, it must implement `IERC721Receiver.onERC721Received`
- The callback must return the correct selector to confirm receipt

Example snippet

```
function safeTransferFrom(address from, address to, uint256 tokenId, bytes
memory data) public {
    transferFrom(from, to, tokenId);
    if (to.code.length > 0) {
        try IERC721Receiver(to).onERC721Received(msg.sender, from,
tokenId, data) returns (bytes4 retval) {
            require(retval == IERC721Receiver.onERC721Received.selector,
"ERC721: wrong return");
        } catch {
            revert("ERC721: non ERC721Receiver implementer");
        }
    }
}
```

Status

Fixed - Issue has been resolved by the development team.

[M-02] ownerOf Should Revert On Non-Existent Tokens Instead Of Returning address(0)

Description

The `TorusPositionNFT` contract's `ownerOf()` function violates the ERC721 standard by returning `address(0)` for non-existent tokens instead of reverting. The function does not follow ERC721 compliance, which should revert when a `tokenId` is non-existent.

Snippet

```
function ownerOf(uint256 tokenId) public view override returns (address) {
    return _owners[tokenId];
}
```

Impact

- **Medium Impact**: Not fully ERC721 compliant, can lead to silent failures
- **Medium Likelihood**: Standard compliance issue affecting all non-existent token queries

Recommendations

- Follow ERC721 compliance and revert the `ownerOf` function when queried for a non-existent tokenId

Example snippet

```
function ownerOf(uint256 tokenId) public view returns (address) {
    address owner = _owners[tokenId];
    require(owner != address(0), "ERC721: invalid token ID");
    return owner;
}
```

Status

Fixed - Issue has been resolved by the development team.

Low

[L-01] Redundant Context Inheritance in TorusPositionNFT

Description

The `TorusPositionNFT` contract redundantly inherits from `Context` even though `Ownable` already inherits from it. This creates unnecessary inheritance complexity, increases deployment gas costs, and violates the DRY (Don't Repeat Yourself) principle without providing any additional functionality.

The current redundant inheritance chain shows `Context` being inherited twice - directly and via `Ownable`.

Snippet

```
contract TorusPositionNFT is Context, Ownable, IERC721, IERC721Metadata {
    // ...
}
```

Impact

- **Low Impact**: Unnecessary gas costs and code complexity
- **Low Likelihood**: No functional impact, just optimization issue

Recommendations

- Remove redundant `Context` inheritance, as `Ownable` already provides it

Example snippet

```
// Before: contract TorusPositionNFT is Context, Ownable { ... }
contract TorusPositionNFT is Ownable { /* ... */ }
```

Status

Fixed - Removed the duplicate context inheritance.

[L-02] Inconsistent msg.sender vs _msgSender() Usage in TorusPositionNFT

Description

The contract inherits from `Context` which provides `_msgSender()` but inconsistently uses both `msg.sender` and `_msgSender()` throughout the codebase. Some functions use `msg.sender` directly while others use `_msgSender()`, creating inconsistent access control patterns.

Snippet

```
// Uses msg.sender
modifier onlyMinter() {
    if (msg.sender != minter) revert NotMinter();
```

```
    _;
}

// Uses _msgSender()
function setApprovalForAll(address operator, bool approved) public
override {
    _operatorApprovals[_msgSender()][operator] = approved;
    emit ApprovalForAll(_msgSender(), operator, approved);
}
```

Impact

- **Low Impact**: Inconsistent access control and meta-transaction incompatibility
- **Low Likelihood**: May cause issues in specific contexts like meta-transactions

Recommendations

- Standardize the use of `_msgSender()` throughout the logic
- Replace all instances of access control using `msg.sender` with `_msgSender()`

Example snippet

```
function mint(...) external {
    address caller = _msgSender();
    require(caller == owner(), "Not authorized");
    _mint(caller, ...);
}
```

Status

Fixed - Standardized with _msgSender().

Conclusion

This security audit of the Torus protocol identified several areas for improvement, all of which have been promptly addressed and resolved by the development team. The protocol demonstrates excellent responsiveness to security concerns and maintains good security practices overall.

Timeline

- **Audit Period**: [08/10/2025 - 08/15/2025]
- **Report Delivery**: [08/21/2025]

This report was prepared by 33Audits & Co and represents our independent security assessment of the Torus protocol smart contracts.