33AUDITS & CO.

Plazm Audit Report

Introduction

A security review of the Plazm 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.

About 33 Audits & Company

33Audits is an independent smart contract security researcher company and development group. We conduct audits 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 Plazm

This audit is being performed on the core protocol contracts for Plazm. The contracts in scope include `PlazmEncapsulator`, `PlazmCreateAndStake`, `Plazm`, `PlazmBuyAndProcess`, and `PlazmNFTPosition`. The Plazm protocol is a token creation and staking system that allows users to create PLAZM tokens by depositing ESHARE/ETH and stake their PLAZM tokens to earn rewards. The protocol includes a referral system and NFT encapsulation features.

Commit: 7e88cd5f06c618ec7fb16ae280e681e2035f79f8 - Scope: `PlazmEncapsulator`, `PlazmCreateAndStake`, `Plazm`, `PlazmBuyAndProcess`, `PlazmNFTPosition`

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
[H-01]	Referral System Manipulation Allows Users To Receive ReferrerFee For Their Own Position	High	Fixed
[L-01]	Precision Loss For Positions Created With Eshare Token	Low	Fixed

High

[H-01] Referral System Manipulation Allows Users To Receive ReferrerFee For Their Own Position

Description

There is strict and explicit checks to ensure that a user cannot receive the referrerFee for their own positions - but this can be bypassed by the user, allowing the user to receive the referrerFee for themselves.

When a user calls PlazmEncapsulator and sets the referrer to the PlazmEncapsulator contract, the ETH referrer fee sent from PlazmCreateAndStake to the referrer increases the encapsulator's ETH balance. The encapsulator then treats this delta as surplus and refunds it back to the user. The referrer fee effectively boomerangs back to the caller instead of being retained by the protocol.

This is the flow in which this bypass happens:

1. User calls PlazmEncapsulator User invokes one of the public entry points on PlazmEncapsulator and provides a referrer address:

```
// PlazmEncapsulator.sol
function createPlazm(uint256 power, uint24 lengthInDays, bool
encapsulateWithNFT, address referrer)
    external
    payable
    returns (uint256 positionId)
{ /* ... */ }

function stakePlazm(uint256 plazmAmount, uint24 stakingDays, bool
encapsulateWithNFT, address referrer)
    external
    payable
    returns (uint256 positionId)
{ /* ... */ }
```

Encapsulator snapshots ETH balance before the call chain proceeds:

```
// PlazmEncapsulator.sol
uint256 ethBalanceBefore = address(this).balance - msg.value;
```

2. Encapsulator forwards to PlazmCreateAndStake. User Inputs the PlazmEncapsulator as the referrer Encapsulator forwards the call (and referrer) to the core contract:

```
// PlazmEncapsulator.sol
createAndStake.createPlazm{value: msg.value}(power, lengthInDays,
msg.sender, referrer);
// or
createAndStake.stakePlazm{value: msg.value}(plazmAmount, stakingDays,
msg.sender, referrer);
```

3. PlazmCreateAndStake distributes ETH fees, including referrer fee Inside _distributeFees, if msg.value > 0 (ETH path), it computes fee splits and sends the referrer share to finalRef:

```
// PlazmCreateAndStake.sol
uint256 refShare = (requiredETH * plazm.referrerFee()) / BPS_DENOM;
bool payToRef = (finalRef != address(0));
uint256 toRef = payToRef ? refShare : 0;
if (toRef > 0) {
    (bool sRef, ) = payable(finalRef).call{ value: toRef, gas: 21000 }
("");
    require(sRef, "ETH to referrer failed");
    emit ReferrerPaid(referee, finalRef, toRef);
}
```

If the user provided referrer = address(PlazmEncapsulator), this ETH lands in the encapsulator contract.

4. Control returns to Encapsulator; refund logic triggers After the core call finishes, Encapsulator compares its ETH balance against the pre-snapshot:

```
// PlazmEncapsulator.sol
uint256 ethBalanceAfter = address(this).balance;
uint256 refund = ethBalanceAfter - ethBalanceBefore;
if (refund > 0) {
    (bool ok, ) = payable(msg.sender).call{ value: refund, gas: 21000 }
("");
    require(ok, "refund fail");
}
```

Because the referrer fee was sent to the Encapsulator during the nested call, ethBalanceAfter > ethBalanceBefore by exactly the referrer fee amount (plus any other unexpected ETH deltas). Encapsulator refunds this delta to the user.

Result: The user takes advantage of the refferer logic and incorrectly receives the referrer fee.

Impact

- **High Impact**: Users can receive referrer fees for their own positions, bypassing the intended referral system
- **High Likelihood**: Attack can be executed by any user calling the encapsulator with themselves as referrer

Recommendations

There are a few recommendations to mitigate this loophole:

Do not allow users to set the referrer to the PlazmEncapsulator contract, or any protocol contract.

There should be codesize checks, to ensure the recipient is strictly an EOA. An example of this, would be the following:

EOA-only referrers with codesize check

```
function _validateReferrer(address referrer) internal view {
    // Ensure referrer is not a contract (EOA only)
    uint256 codeSize;
    assembly {
        codeSize := extcodesize(referrer)
    }
    require(codeSize == 0, "Referrer must be EOA, not contract");

    // Additional safety: block zero address
    require(referrer != address(0), "Invalid referrer address");
}
```

Implementation in PlazmCreateAndStake.sol:

```
function _distributeFees(
    uint256 requiredETH,
    uint256 costEshare,
    address referee,
    address referrer
) internal {

    // Validate referrer is EOA before any fee distribution
    if (finalRef != address(0)) {
        _validateReferrer(finalRef);
    }

    if (msg.value > 0) {
        // ... existing ETH distribution logic ...
        if (toRef > 0) {
            // Now safe to send to referrer since it's guaranteed to be
```

```
EOA
            (bool sRef, ) = payable(finalRef).call{ value: toRef, gas:
21000 }("");
            require(sRef, "ETH to referrer failed");
            emit ReferrerPaid(referee, finalRef, toRef);
        }
    }
    // ... rest of function ...
}
```

Status

Fixed - The developer has added restrictions in createPlazm, stakePlazm in both the createAndStake contract, as well as the encapsulator contract to prevent users from abusing the refund logic.

Low

[L-01] Precision Loss For Positions Created With Eshare Token

Description

The ESHARE payment path in PlazmCreateAndStake.sol has a precision error in fee distribution calculations. The code subtracts the raw referrerFee BPS value from each individual fee calculation instead of properly adjusting the denominator first.

The current logic performs:

Multiplication and division first: (costEshare * feeBPS) / BPS_DENOM Then subtracts raw BPS value: - plazm.referrerFee() The intention is to reduce the BPS for this path, since the referrer fee is not being computed. But that is not done correctly, instead of reducing the BPS by the referrerFee - it instead just subtracts the referrerFee (200) from the final calculation.

Example Calculation Let's assume costEshare is 1,000e18

Current Fee Distribution Model:

```
genesisFee = 500 BPS (5%)
burnFee = 6500 BPS (65%)
buildFee = 2800 BPS (28%)
referrerFee = 200 BPS (2%)
BPS_DENOM = 10000
```

Current Logic (Incorrect):

```
Genesis: (1000e18 * 500) / 10000 - 200 = 50e18 - 200
Burn:    (1000e18 * 6500) / 10000 - 200 = 650e18 - 200
```

```
Build:   (1000e18 * 2800) / 10000 - 200 = 280e18 - 200
```

Correct Distribution Should Be: First: Reduce the BPS by the referrerFee percentage amount: (10,000 - 200) = 9800.

```
Genesis: 1000 * 500 / 9800 = 51
Burn:    1000 * 6500 / 9800 = 663
Build:   1000 * 2800 / 9800 = 286
```

Impact

- **Low Impact**: Precision loss in fee distribution calculations for ESHARE payment path
- **Low Likelihood**: Affects all positions created with ESHARE tokens

Numerical Example: When costEshare is 1000e18, the precision loss results in:

```
// Current (Incorrect) Calculation:
Genesis: 4999999999999999800
Burn:    64999999999999999800
Build:   27999999999999999800
Total:   97999999999999999400
// Error: 2.000000%

// Correct Calculation:
Genesis: 5102040816326530612
Burn:    66326530612244897959
Build:   28571428571428571428
Total:   99999999999999999999
// Precision loss: 2.000000%
```

Recommendations

Normalize the BPS first and adjust the denominator, by subtracting the referrerFee directly from the BPS.

```
else {
    // Calculate effective denominator excluding referrer fee
    uint256 effectiveDenom = BPS_DENOM - plazm.referrerFee();

    uint256 requiredGenesisEshare = (costEshare * plazm.genesisFee()) /
effectiveDenom;
    uint256 requiredBurnEshare = (costEshare * plazm.burnFee()) /
effectiveDenom;
    uint256 requiredBuildEshare = (costEshare * plazm.buildFee()) /
effectiveDenom;
}
```

Status

Fixed - The developer has applied the effectiveDenom method to correct the precision loss.

Conclusion

This security audit of the Plazm protocol identified a few areas for improvement; however, overall the protocol was well developed and highly secure due to multiple previous audits. All identified issues have been promptly addressed and resolved by the development team. The protocol demonstrates excellent responsiveness to security concerns and maintains good security practices overall.

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