33AUDITS & CO.

Plazm EasyMode Audit Report

Introduction

A security review of the Plazm-EasyMode protocol was done, focusing on the security aspects of the smart contracts. This audit was performed by 33Audits & Co with Sam as 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 PlazmEasyMode

This audit is being performed on the PlazmEasyMode contract, which is a user-facing convenience layer for the Plazm protocol. The PlazmEasyMode contract simplifies the user experience by providing automated entry points that select between creation and staking paths, handles token swaps, and manages referral tracking. The contract integrates with PlazmCreateAndStake (the core protocol contract) and ReferralRegistry for referral fee distribution.

Repository: https://github.com/BuildTheTech/Plazm-Contracts

Scope: `PlazmEasyMode.sol`

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]	First Referrer from PlazmEasyMode Will Indefinitely Receive All Referral Fees	Critical	Fixed
[H-01]	Referral System Manipulation Allows Users To Receive ReferrerFee For Their Own Position	High	Fixed
[M-01]	Incorrect Path Validation Logic For UniswapV3	Medium	Fixed
[L-01]	ETH Transfers Can Silently Fail, Resulting In Locked Funds In PlazmEasyMode	Low	Fixed
[L-02]	Fee On Transfer Tokens Will Always Fail In Used For Token Path	Low	Fixed
[L-03]	Precision Loss For Positions Created With Eshare Token	Low	Fixed

Critical

[C-01] First Referrer from PlazmEasyMode Will Indefinitely Receive All Referral Fees

Description

The `PlazmEasyMode` contract incorrectly handles referral system updates, causing the first referrer ever used to become permanently associated with the `PlazmEasyMode` contract address. This results in that first referrer receiving referral fees for ALL subsequent operations through `PlazmEasyMode`, regardless of the actual referrer specified by users.

The referral system has a dual-call architecture where both `PlazmEasyMode` and `PlazmCreateAndStake` call `resolveAndMaybeSetReferrer()`, but `PlazmEasyMode` is not set as the encapsulator in `PlazmCreateAndStake`, causing the wrong referrer resolution logic to execute.

Affected Code:

```
// PlazmEasyMode.sol - _executeCreate() and similar functions
IReferralRegistry rr = referralRegistry;
if (address(rr) != address(0)) {
    rr.resolveAndMaybeSetReferrer(user, referrer);
}
createAndStake.createPlazm{value: ethAmount}(power, lengthInDays, user,
referrer);
```

```
// PlazmCreateAndStake.sol - createPlazm()
IReferralRegistry rr = referralRegistry;
address finalRef = address(0);
address referee = msg.sender;
if (address(rr) != address(0)) {
    if (msg.sender == encapsulator) {
        referee = user;
        try rr.resolveAndMaybeSetReferrer(user, referrer) returns (address
refResult) {
            finalRef = refResult;
        } catch {}
    } else {
        referee = msg.sender;  // ← msg.sender = PlazmEasyMode contract
        try rr.resolveAndMaybeSetReferrer(msg.sender, referrer) returns
(address refResult) {
            finalRef = refResult;  // ← Returns existing referrer for
PlazmEasyMode
        } catch {}
    }
}
```

```
// ReferralRegistry.sol - resolveAndMaybeSetReferrer()
function resolveAndMaybeSetReferrer(address user, address referrer)
external onlyWriter returns (address) {
    address existing = userReferrer[user];
    if (existing != address(0)) {
        return existing;  // ← Returns first referrer set for
PlazmEasyMode
    }
    if (referrer == address(0) || referrer == user) {
        return address(0);
    }
    userReferrer[user] = referrer;  // ← Sets PlazmEasyMode's referrer on
first call
    // ... rest of logic
    return referrer;
}
```

Attack Flow

Step 1: First Transaction - Alice uses Bob as referrer

1. `PlazmEasyMode` calls: `rr.resolveAndMaybeSetReferrer(Alice, Bob)`
    - Sets: `userReferrer[Alice] = Bob` ✓ (correct)
2. `PlazmEasyMode` calls `createAndStake.createPlazm()` which internally calls:
    - `rr.resolveAndMaybeSetReferrer(PlazmEasyMode, Bob)`

- Since `PlazmEasyMode` is not the encapsulator, the `else` branch executes
- Sets: `userReferrer[PlazmEasyMode] = Bob` ✗ (incorrect - Bob becomes permanent)

3. `finalRef = Bob` → Bob receives fees ✓ (correct for this transaction)

Step 2: Subsequent Transactions - Charlie uses Dave as referrer

1. `PlazmEasyMode` calls: `rr.resolveAndMaybeSetReferrer(Charlie, Dave)`
   - Sets: `userReferrer[Charlie] = Dave` ✓ (correct)
2. `PlazmEasyMode` calls `createAndStake.createPlazm()` which internally calls:
   - `rr.resolveAndMaybeSetReferrer(PlazmEasyMode, Dave)`
   - `existing = userReferrer[PlazmEasyMode] = Bob` (from Step 1)
   - Returns `existing = Bob` ✗ (should be Dave)
3. `finalRef = Bob` → Bob receives fees ✗ (should be Dave)

Step 3: All Future Transactions

Every subsequent transaction through `PlazmEasyMode` will:

- Correctly set the referrer for the individual user
- But always return Bob as the referrer for `PlazmEasyMode` contract
- Result in Bob receiving all referral fees indefinitely

Impact

- **High Impact**: The first referrer receives all referral fees for all future transactions through `PlazmEasyMode`, regardless of the intended referrer. This represents a permanent misallocation of referral rewards.
- **High Likelihood**: This occurs on the very first transaction and affects all subsequent operations.

Economic Impact:

- Legitimate referrers lose all referral fees
- The first referrer receives unintended windfall
- Protocol reputation damage due to unfair reward distribution
- Potential for exploitation if the first referrer is maliciously chosen

Recommendations

The issue stems from `PlazmEasyMode` not being recognized as an encapsulator in `PlazmCreateAndStake`. The recommended fix is to set `PlazmEasyMode` as the encapsulator:

```
// PlazmCreateAndStake.sol - createPlazm()
if (msg.sender == encapsulator || msg.sender == address(plazmEasyMode)) {
    referee = user;
    try rr.resolveAndMaybeSetReferrer(user, referrer) returns (address
refResult) {
        finalRef = refResult;
```

```
    } catch {}
} else {
    referee = msg.sender;
    try rr.resolveAndMaybeSetReferrer(msg.sender, referrer) returns
(address refResult) {
        finalRef = refResult;
    } catch {}
}
```

Alternatively, implement a `ReferralForwarder` contract pattern where:

1. `PlazmEasyMode` always has its referrer set to a `ReferralForwarder` contract
2. The `ReferralForwarder` contract receives referral fees and forwards them to the actual user's referrer
3. This ensures the correct referrer always receives fees while maintaining the dual-call architecture

Status

Fixed - The developer has implemented a `ReferralForwarder` contract pattern. `PlazmEasyMode` now always has its referrer set to the `ReferralForwarder` contract, which forwards referral fees to the actual user's referrer. This ensures correct referrer resolution while maintaining the existing architecture.

The fix ensures that:

- The first transaction properly sets up the `ReferralForwarder` as the referrer for `PlazmEasyMode`
- All subsequent transactions correctly route referral fees through the forwarder to the intended referrer
- No single referrer can permanently capture all referral fees

---

High

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

Description

Despite explicit checks to prevent users from receiving referrer fees for their own positions, this protection can be bypassed by setting the referrer to a protocol contract (such as `PlazmEncapsulator`). When a user sets the referrer to the encapsulator 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, effectively allowing the user to receive the referrer fee for their own position.

This vulnerability exploits the refund logic in encapsulator contracts that calculate refunds based on ETH balance deltas, without accounting for referral fees that may have been sent to the contract during nested calls.

Affected Code:

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

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

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");
}
```

```
// PlazmCreateAndStake.sol - _distributeFees()
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);
}
```

Attack Flow

Step 1: User calls PlazmEncapsulator with self as referrer

1. User invokes `createPlazm()` or `stakePlazm()` on `PlazmEncapsulator`
2. User sets `referrer = address(PlazmEncapsulator)`
3. Encapsulator snapshots ETH balance: `ethBalanceBefore = address(this).balance - msg.value`

Step 2: Encapsulator forwards to PlazmCreateAndStake

1. Encapsulator calls `createAndStake.createPlazm{value: msg.value}(...)` or `stakePlazm{value: msg.value}(...)`
2. The referrer (`PlazmEncapsulator` address) is passed through to the core contract

Step 3: PlazmCreateAndStake distributes ETH fees

1. Inside `_distributeFees()`, the referrer fee is calculated: `refShare = (requiredETH * plazm.referrerFee()) / BPS_DENOM`

2. Since `finalRef = address(PlazmEncapsulator)`, the referrer fee is sent to the encapsulator contract:

```
payable(PlazmEncapsulator).call{ value: refShare, gas: 21000 }("")
```

3. This increases the encapsulator's ETH balance by the referrer fee amount

Step 4: Control returns to Encapsulator; refund logic triggers

1. Encapsulator calculates: `ethBalanceAfter = address(this).balance`
2. The balance delta includes the referrer fee that was just sent to the encapsulator
3. Encapsulator calculates refund: `refund = ethBalanceAfter - ethBalanceBefore`
4. Since `refund > 0` (includes the referrer fee), the entire delta is refunded to the user
5. **Result**: User receives the referrer fee that should have been retained by the protocol

Impact

- **High Impact**: Users can bypass the intended restriction and receive referrer fees for their own positions, directly reducing protocol revenue and undermining the referral incentive system.
- **High Likelihood**: This attack is straightforward to execute - users simply need to set the referrer to the encapsulator contract address.

Economic Impact:

- Protocol loses referrer fee revenue that should be distributed to legitimate referrers
- Users can effectively reduce their position costs by receiving their own referrer fees
- Undermines the integrity of the referral system
- Potential for systematic exploitation if not addressed

Recommendations

The recommended fix is to prevent users from setting protocol contracts as referrers. This can be implemented in multiple ways:

Option 1: Block protocol contracts as referrers

Add validation in `PlazmCreateAndStake` and encapsulator contracts to reject protocol contract addresses:

```
// PlazmCreateAndStake.sol - createPlazm() and stakePlazm()
if (referrer == address(this) || referrer == encapsulator || referrer ==
address(plazmEasyMode)) {
    revert InvalidReferrer();
}
```

Option 2: EOA-only referrers with codesize check

Require referrers to be Externally Owned Accounts (EOAs) only:

```
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");
}

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);
    }

    // ... rest of fee distribution logic ...
}
```

Option 3: Exclude referral fees from refund calculations

Modify the refund logic in encapsulator contracts to account for referral fees:

```
// PlazmEncapsulator.sol
uint256 ethBalanceBefore = address(this).balance - msg.value;
uint256 expectedRefShare = 0;

// Calculate expected referrer fee if referrer is this contract
if (referrer == address(this)) {
    // Prevent this scenario or calculate expected fee
    revert InvalidReferrer();
}

createAndStake.createPlazm{value: msg.value}(power, lengthInDays,
msg.sender, referrer);

uint256 ethBalanceAfter = address(this).balance;
uint256 refund = ethBalanceAfter - ethBalanceBefore;
// Subtract any referral fees that may have been sent to this contract
if (referrer == address(this)) {
    refund -= expectedRefShare;
}
if (refund > 0) {
```

```
    (bool ok, ) = payable(msg.sender).call{ value: refund, gas: 21000 }
("");
    require(ok, "refund fail");
}
```

Status

Fixed - The developer has added restrictions in `createPlazm()` and `stakePlazm()` functions in both the `PlazmCreateAndStake` contract and the encapsulator contract to prevent users from abusing the refund logic by setting protocol contracts as referrers.

The fix ensures that:

- Users cannot set protocol contracts (including encapsulator contracts) as referrers
- The refund logic is protected from manipulation through referrer fee exploitation
- The referral system maintains its intended incentive structure

---

Medium

[M-01] Incorrect Path Validation Logic For UniswapV3

Description

The `PlazmEasyMode` contract incorrectly validates Uniswap V3 swap paths. The minimum path length check is insufficient, only requiring `pathV3.length >= 20`, which ensures a single token is present. However, Uniswap V3 requires at least 43 bytes for a valid swap path (2 tokens + 1 fee).

Uniswap V3 path structure follows this format where each token has 20 bytes and fees are 3 bytes:

```
[token0 (20 bytes)] [fee (3 bytes)] [token1 (20 bytes)] [fee (3 bytes)]
[token2 (20 bytes)] ...
```

This makes a valid path at least 43 bytes (20 bytes for token0 + 3 bytes for fee + 20 bytes for token1). The current validation accepts paths with only 20 bytes, which means single-token paths are incorrectly validated as correct.

Affected Code:

```
// PlazmEasyMode.sol - _acquirePlazm() - Line 844
} else if (swap.routerType == RouterType.UniswapV3) {
    require(swap.pathV3.length >= 20, "path short");  // ← Only validates
1 token
    require(_decodeLastToken(swap.pathV3) == plazmToken, "path end");
    // ...
}
```

```
// PlazmEasyMode.sol - _executeSwapForEth() - Line 902
} else if (swap.routerType == RouterType.UniswapV3) {
    require(swap.pathV3.length >= 20, "path short");  // ← Only validates
1 token
    require(_decodeLastToken(swap.pathV3) == WETH_ADDRESS, "path must end
WETH");
    // ...
}
```

Comparison with UniswapV2 validation:

The UniswapV2 path validation correctly ensures at least 2 tokens are present:

```
// PlazmEasyMode.sol - UniswapV2 validation
if (swap.routerType == RouterType.UniswapV2) {
    require(swap.path.length >= 2, "path short");  // ← Correctly
validates 2 tokens
    // ...
}
```

Impact

- **Medium Impact**: Invalid and malformed UniswapV3 paths are accepted, which can lead to swap failures or unexpected behavior when attempting to execute swaps with invalid paths.
- **Medium Likelihood**: This validation error affects all UniswapV3 swap operations through `PlazmEasyMode`, but may not be immediately exploitable if Uniswap's router itself rejects invalid paths.

Technical Impact:

- Invalid single-token paths (20 bytes) are accepted when they should be rejected
- Required minimum path length (43 bytes for 2 tokens + 1 fee) is not enforced
- Potential for swap execution failures or unexpected behavior
- Inconsistency between UniswapV2 and UniswapV3 validation logic

Recommendations

Update the validation for UniswapV3 paths to require at least 43 bytes, ensuring at least 2 tokens and 1 fee are present in the path:

```
// Replace:
require(swap.pathV3.length >= 20, "path short");

// With:
require(swap.pathV3.length >= 43, "path short");
```

This should be applied in all locations where UniswapV3 path validation occurs:

1. `_acquirePlazm()` function - Line 844
2. `_executeSwapForEth()` function - Line 902
3. Any other functions that validate UniswapV3 paths

This ensures consistency with UniswapV3's path structure requirements and aligns the validation logic with UniswapV2's approach of requiring at least 2 tokens.

Status

Fixed - The developer has updated the UniswapV3 path validation to require at least 43 bytes, ensuring that paths contain at least 2 tokens and 1 fee as required by UniswapV3's path structure.

The fix ensures that:

- Invalid single-token paths are properly rejected
- Path validation aligns with UniswapV3's requirements
- Consistency between validation logic and actual path structure requirements

---

Low

[L-01] ETH Transfers Can Silently Fail, Resulting In Locked Funds In PlazmEasyMode

Description

In `PlazmEasyMode`, ETH fee transfers to the genesis wallet are not validated, allowing failed transfers to pass silently. When fee transfers fail, the ETH remains permanently locked in the contract with no recovery mechanism, as the contract removed the fee withdrawal functions that existed in previous versions.

The contract performs low-level ETH transfers using `call{value: ...}()` but does not check the return value to verify transfer success. When transfers fail for any reason, the ETH remains in the contract with no way to recover it.

Affected Code:

```
// PlazmEasyMode.sol - easyCreateWithETH() - Lines 315-318
if (fee > 0) {
    (bool okFee, ) = payable(plazm.genesisWallet()).call{value: fee, gas:
21000}("");
    okFee;  // ← NO CHECK! Fee can fail silently
}
```

```
// PlazmEasyMode.sol - easyStakeWithETH() - Lines 384-387
if (feeEth > 0) {
    (bool okFee, ) = payable(plazm.genesisWallet()).call{value: feeEth,
```

```
gas: 21000}("");
    okFee;  // ← NO CHECK! Fee can fail silently
}
```

```
// PlazmEasyMode.sol - compoundPosition() - Lines 818-821
if (feeEth > 0) {
    (bool okFee, ) = payable(plazm.genesisWallet()).call{value: feeEth,
gas: 21000}("");
    okFee;  // ← NO CHECK! Fee can fail silently
}
```

Comparison with correct implementation:

The refund logic correctly validates the transfer result:

```
// PlazmEasyMode.sol - Refund logic (CORRECT)
if (refund > 0) {
    (bool ok, ) = msg.sender.call{value: refund}("");
    require(ok, "refund fail");  // ← CORRECT - checks the result
}
```

Impact

- **Medium Impact**: Failed ETH transfers result in funds being permanently locked in the contract with no recovery mechanism.
- **Low Likelihood**: ETH transfers typically succeed, but failures can occur if the recipient is a contract without a receive/fallback function, or in edge cases with gas limits.

Economic Impact:

- Protocol fees may be lost if transfers fail
- No recovery mechanism for locked funds
- Potential accumulation of locked ETH over time
- Inconsistency between fee transfer validation and refund validation

Recommendations

Validate the result of all ETH transfers to ensure they succeed:

```
// Replace:
if (fee > 0) {
    (bool okFee, ) = payable(plazm.genesisWallet()).call{value: fee, gas:
21000}("");
    okFee;  // ← Remove this unused variable
}
```

```
// With:
if (fee > 0) {
    (bool okFee, ) = payable(plazm.genesisWallet()).call{value: fee, gas:
21000}("");
    require(okFee, "Fee transfer failed");
}
```

This should be applied to all fee transfer locations:

1. `easyCreateWithETH()` function
2. `easyStakeWithETH()` function
3. `compoundPosition()` function
4. Any other functions that transfer ETH fees

This ensures consistency with the refund logic and prevents silent failures that could result in locked funds.

Status

Fixed - The developer has added validation checks for all ETH fee transfers to ensure they succeed. Failed transfers now properly revert, preventing funds from being locked in the contract.

The fix ensures that:

- All ETH transfers are validated before execution continues
- Failed transfers properly revert the transaction
- Consistency between fee transfer and refund validation logic
- No funds can be silently locked due to transfer failures

[L-02] Fee On Transfer Tokens Will Always Fail In Used For Token Path

Description

In `PlazmEasyMode`, fee-on-transfer tokens are not properly handled in the `easyStakeWithSwap()` function. The contract uses the original `swapToPlazm.amountIn` instead of the actual amount received after the transfer, causing over approvals, swap failures, and incorrect calculations for fee-on-transfer tokens.

Fee-on-transfer tokens deduct a fee during the transfer itself. When `safeTransferFrom()` is called with `amountIn`, the contract receives less than `amountIn` due to the transfer fee. However, the code calculates `netToken = swapToPlazm.amountIn` (or `swapToPlazm.amountIn - feeToken`), which assumes the full amount was received.

Affected Code:

```
// PlazmEasyMode.sol - easyStakeWithSwap() - Line 434
IERC20(swapToPlazm.tokenIn).safeTransferFrom(msg.sender, address(this),
swapToPlazm.amountIn);
// User sends: swapToPlazm.amountIn (1000 tokens)
// Contract receives: Less than amountIn (950 tokens due to 5% fee)
```

```
// PlazmEasyMode.sol - Fee calculation incorrectly using full amountIn
if (payFeeInEth) {
    netToken = swapToPlazm.amountIn;  // ← Uses original amount, not
received
} else {
    uint256 feeToken = _takePlazmFee(swapToPlazm.amountIn);
    netToken = swapToPlazm.amountIn - feeToken;  // ← Still uses original
amount!
}
// Calculates netToken based on amountIn, not actual received amount
// For FOT tokens netToken > actual balance
```

Problem Flow:

1. User sends: 1000 tokens (5% transfer fee)
2. Contract receives: 950 tokens
3. Contract/protocol fee: 50 tokens (5% of 950)
4. Current calculation: `netToken = 950` (if `payFeeInEth=false`)
5. Actual after contract fee: 900 tokens
6. **Problem**: Still uses 950 instead of 900 for swap calculations

Impact

- **Medium Impact**: Swaps for fee-on-transfer tokens will fail, preventing these tokens from being used in the token path within `PlazmEasyMode`.
- **Medium Likelihood**: This affects all fee-on-transfer tokens used in swap operations, which are common in DeFi (e.g., PAXG, some stablecoins).

Technical Impact:

- Swap executions fail with "Insufficient balance" errors
- Over-approval of routers with amounts exceeding actual balance
- Incorrect fee calculations based on sent amount rather than received amount
- Fee-on-transfer tokens become unusable in the token swap path

Recommendations

Update the logic for the token path to use actual balances received. Use the actual balance received for the protocol fee calculation, and then use the actual amount received minus the protocol fee for swap inputs:

```
// Get actual received amount (handles fee-on-transfer tokens)
uint256 balanceBefore =
IERC20(swapToPlazm.tokenIn).balanceOf(address(this));
IERC20(swapToPlazm.tokenIn).safeTransferFrom(msg.sender, address(this),
swapToPlazm.amountIn);
uint256 balanceAfter =
IERC20(swapToPlazm.tokenIn).balanceOf(address(this));
```

```
    uint256 actualReceived = balanceAfter - balanceBefore;

    uint256 netToken;
    if (payFeeInEth) {
        (uint256 netEthMsg, uint256 feeEth) = _takeEthFee(ethAvailable);
        if (feeEth > 0) {
            (bool okFee2, ) = payable(plazm.genesisWallet()).call{value:
    feeEth, gas: 21000}("");
            if (!okFee2) revert FeeTransferFailed();
        }
        ethAvailable = netEthMsg;
        netToken = actualReceived;  // ← CORRECT: Use actual received amount
    } else {
        uint256 feeToken = _takePlazmFee(actualReceived);  // ← CORRECT:
    Calculate fee on actual amount
        if (feeToken > 0) {
            IERC20(swapToPlazm.tokenIn).safeTransfer(plazm.genesisWallet(),
    feeToken);
        }
        netToken = actualReceived - feeToken;  // ← CORRECT: Use actual amount
    minus fee
    }
```

This pattern should be applied wherever tokens are transferred from users and used in calculations, ensuring fee-on-transfer tokens are properly handled.

Status

Fixed - The developer has updated the token path logic to use actual balances received after transfers. The contract now properly handles fee-on-transfer tokens by calculating fees and swap amounts based on the actual received amount rather than the sent amount.

The fix ensures that:

- Actual received amounts are used for all calculations
- Fee calculations are based on received amounts, not sent amounts
- Swaps use correct amounts that match actual token balances
- Fee-on-transfer tokens can be successfully used in swap operations

---

[L-03] 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:

1. Multiplication and division first: `(costEshare * feeBPS) / BPS_DENOM`

2. 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. However, instead of reducing the BPS by the `referrerFee`, it subtracts the raw `referrerFee` value (200) from the final calculation, causing precision loss.

Affected Code:

```
// PlazmCreateAndStake.sol - _distributeFees() - ESHARE path (INCORRECT)
else {
    // Current incorrect logic
    uint256 requiredGenesisEshare = (costEshare * plazm.genesisFee()) /
BPS_DENOM - plazm.referrerFee();
    uint256 requiredBurnEshare = (costEshare * plazm.burnFee()) /
BPS_DENOM - plazm.referrerFee();
    uint256 requiredBuildEshare = (costEshare * plazm.buildFee()) /
BPS_DENOM - plazm.referrerFee();
}
```

Example Calculation:

Assuming `costEshare = 1,000e18` and fee structure:

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

Current (Incorrect) Logic:

```
Genesis: (1000e18 * 500) / 10000 - 200 = 50e18 - 200 = 4999999999999999800
Burn:    (1000e18 * 6500) / 10000 - 200 = 650e18 - 200 =
64999999999999999800
Build:   (1000e18 * 2800) / 10000 - 200 = 280e18 - 200 =
27999999999999999800
Total:   97999999999999999400 (error of 2%)
```

Correct Distribution Should Be:

First, reduce the BPS denominator by the `referrerFee` percentage: `(10,000 - 200) = 9800`

```
Genesis: (1000e18 * 500) / 9800 = 5102040816326530612
Burn:    (1000e18 * 6500) / 9800 = 66326530612244897959
Build:   (1000e18 * 2800) / 9800 = 28571428571428571428
Total:   99999999999999999999 (correct)
```

Precision Loss:

- Difference: `2,000,000,000,000,000,599`
- Precision loss: `2.000000%`

Impact

- **Low Impact**: Results in a 2% precision loss in fee distribution calculations for positions created with ESHARE tokens. The total distributed amount is less than the `costEshare` amount.
- **Low Likelihood**: This affects all positions created using the ESHARE payment path, but the impact is relatively small (2% precision loss).

Economic Impact:

- Fee distribution does not sum to 100% of `costEshare`
- Approximately 2% of ESHARE fees are not properly distributed
- Inconsistency between intended fee distribution and actual distribution
- Potential accumulation of unaccounted ESHARE over time

Recommendations

Normalize the BPS first by adjusting the denominator, subtracting the `referrerFee` directly from the BPS denominator:

```
// PlazmCreateAndStake.sol - _distributeFees() - ESHARE path (CORRECT)
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;
}
```

This ensures that:

- The denominator is properly adjusted to exclude the referrer fee
- Fee calculations are proportional and sum to 100% of `costEshare`
- No precision loss occurs in the fee distribution

Status

Fixed - The developer has applied the `effectiveDenom` method, calculating the effective denominator by subtracting the `referrerFee` from `BPS_DENOM` before performing fee calculations.

The fix ensures that:

- Fee distribution calculations are mathematically correct
- Total distributed fees equal 100% of `costEshare`
- No precision loss occurs in ESHARE fee distribution
- Proper proportional distribution of fees across genesis, burn, and build allocations

---

Conclusion

This security audit of the PlazmEasyMode contract identified various vulnerabilities across different severity levels, including critical issues in the referral system, path validation errors, and precision loss in fee calculations. All identified issues have been promptly addressed and resolved by the development team.

The development team's commitment to addressing these findings demonstrates a strong security-focused approach to smart contract development.

Fix commit: b9ecb5f

This report was prepared by 33Audits & Co and represents our independent security assessment of the PlazmEasyMode contract.