CD SECURITY

AUDIT REPORT

eSIM Wallet
February 2025

Prepared by
tsvetanovv
dimulski

Introduction

A time-boxed security review of the **eSIM Wallet** protocol was done by **CD Security**, with a focus on the security aspects of the application's implementation.

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 problems with your smart contracts. Subsequent security reviews, bug bounty programs, and on-chain monitoring are strongly recommended.

About **eSIM Wallet**

The eSIM Wallet Smart Contract Suite enables secure, user-friendly blockchain wallets by integrating `WebAuthn` authentication and eSIM technology.

Instead of traditional private keys, users authenticate transactions using device-bound credentials, such as biometric authentication or secure enclave keys. The protocol supports ERC-4337 account abstraction, allowing gasless transactions through smart contract wallets.

Lazy wallet registration optimizes gas costs by enabling off-chain wallet initialization before on-chain deployment. The `WebAuthn` integration provides phishing-resistant authentication, ensuring only authorized devices can sign transactions.

Signature verification is handled via P-256 elliptic curve cryptography, adding an extra layer of security. The suite includes smart contract wallets, wallet factories, registry mechanisms, and helper contracts to manage user credentials securely. Device wallets allow multi-device authentication while maintaining security and usability.

The protocol's design enhances user experience by eliminating the need for seed phrases while ensuring trustless and tamper-proof authentication. By leveraging hardware-backed security, it significantly reduces the risks associated with private key management and unauthorized access.

Severity classification

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 of a successful attack

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

Severity - the overall criticality of the risk

Security Assessment Summary

***review commit hash* - fd26940241d6496e030fe46004055d3647666b60**

Scope

The following folders were in scope of the audit:

- `contracts/*`

The following number of issues were found, categorized by their severity:

- Critical & High: 4 issues
- Medium: 2 issues
- Low: 5 issues

Findings Summary

ID	Title	Severity	Status
[C-01]	Malicious but registered Device Wallet can steal an eSIM Wallet	Critical	Fixed
[C-02]	Frontrunning Attack Leading to Permanent ETH Loss in `createAccount`	Critical	Fixed
[H-01]	Device wallet creation can be DoSed, making the whole protocol obsolete	High	Fixed
[H-02]	The `deployLazyWalletAndSetESIMIdentifier` function won't send ETH to the registry and will revert, resulting in a core protocol functionality not working	High	Fixed
[M-01]	Excess ETH not returned in `deployDeviceWalletAsAdmin` and `createAccount`	Medium	Fixed
[M-02]	Corruptible Upgradability Pattern	Medium	Fixed
[L-01]	Lack of signature expiration in `verifySignature`	Low	Fixed
[L-02]	Missing zero address check in `withdrawDepositTo`	Low	Fixed
[L-03]	Discrepancy in nat spec and code implementation	Low	Fixed
[L-04]	Removed DeviceWallet can change eSIMWallet configuration in the registry	Low	Fixed

Detailed Findings

[C-01] Malicious but registered Device Wallet can steal an eSIM Wallet

Severity

Impact: High

Likelihood: High

Description

The updateDeviceWalletAssociatedWithESIMWallet() function in Registry.sol links an eSIM wallet to a specific device wallet.

```
    function updateDeviceWalletAssociatedWithESIMWallet(
        address _eSIMWalletAddress,
        address _deviceWalletAddress
    ) external onlyDeviceWallet {
        isESIMWalletValid[_eSIMWalletAddress] = _deviceWalletAddress;
        emit
UpdatedDeviceWalletassociatedWithESIMWallet(_eSIMWalletAddress,
_deviceWalletAddress);
    }
```

The function has a modifier that ensures that only valid Device Wallets can call this function:

```
modifier onlyDeviceWallet() {
    if(isDeviceWalletValid[msg.sender] != true) revert
Errors.OnlyDeviceWallet();
    _;
}
```

The problem here is that updateDeviceWalletAssociatedWithESIMWallet() currently lacks an ownership check, meaning any valid **Device Wallet** could potentially link itself to any **eSIM Wallet**, even if that **eSIM Wallet** is already linked to another device.

For example:

1. Alice owns an eSIM Wallet (0xESIM1), linked to her device (0xDeviceAlice)**.
2. Bob, a malicious actor, has a registered Device Wallet (0xDeviceBob)**.
3. Bob calls: updateDeviceWalletAssociatedWithESIMWallet(0xESIM1, 0xDeviceBob);
4. Alice's eSIM Wallet is now controlled by Bob's device.
5. Bob gains control over Alice's eSIM wallet.

This allows a malicious but registered Device Wallet to steal an eSIM Wallet by reassigning it.

Recommendations

Ensure that only the current owner (device) of an eSIM Wallet can update it or that only unlinked eSIM wallets can be assigned to a new device. This will prevent overriding an existing association:

```
    require(
        isESIMWalletValid[_eSIMWalletAddress] == address(0) ||
        isESIMWalletValid[_eSIMWalletAddress] == msg.sender,
        "Unauthorised caller or already assigned"
    );
```

[C-02] Frontrunning Attack Leading to Permanent ETH Loss in `createAccount`

Severity

Impact: High

Likelihood: High

Description

The `createAccount()` function in `DeviceWalletFactory.sol` allows users to deploy a new DeviceWallet and prefund it with ETH.

```
    function createAccount(
        string memory _deviceUniqueIdentifier,
        bytes32[2] memory _deviceWalletOwnerKey,
        uint256 _salt,
        uint256 _depositAmount
    ) public payable returns (DeviceWallet deviceWallet) {
        require(
            bytes(_deviceUniqueIdentifier).length != 0,
            "DeviceIdentifier cannot be empty"
        );

        address addr = getAddress(
            _deviceWalletOwnerKey,
            _deviceUniqueIdentifier,
            _salt
        );

        // Check if the device identifier is actually unique
        address wallet =
registry.uniqueIdentifierToDeviceWallet(_deviceUniqueIdentifier);
        if(wallet != address(0)) {
            return DeviceWallet(payable(wallet));
        }

        // Check if P256 public key is actually unique
        bytes32 keyHash = keccak256(abi.encode(_deviceWalletOwnerKey[0],
_deviceWalletOwnerKey[1]));
        wallet = registry.registeredP256Keys(keyHash);
        if(wallet != address(0)) {
            return DeviceWallet(payable(wallet));
        }

        uint256 codeSize = addr.code.length;
        if (codeSize > 0) {
            return DeviceWallet(payable(addr));
        }

        // Prefund the account with msg.value
        if (msg.value > 0 && _depositAmount <= msg.value) {
            entryPoint.depositTo{value: _depositAmount}(addr);
        }

        deviceWallet = DeviceWallet(
            payable(
                new BeaconProxy{salt : bytes32(_salt)}(
                    address(beacon),
                    abi.encodeCall(
                        DeviceWallet.init,
                        (address(registry), _deviceWalletOwnerKey,
_deviceUniqueIdentifier)
                    )
                )
            )
        );

        registry.updateDeviceWalletInfo(address(deviceWallet),
_deviceUniqueIdentifier, _deviceWalletOwnerKey);
    }
```

However, an attacker can front-run a victim’s transaction by deploying the wallet first. This causes the victim's transaction to return the already deployed wallet without executing the ETH deposit.

```
        // Check if the device identifier is actually unique
        address wallet =
registry.uniqueIdentifierToDeviceWallet(_deviceUniqueIdentifier);
        if(wallet != address(0)) {
            return DeviceWallet(payable(wallet));
        }
```

As a result, the victim's ETH remains permanently locked in the factory contract.

Attack Vector

1. The victim sends a transaction to `createAccount()` with `_deviceUniqueIdentifier`, `_deviceWalletOwnerKey`, and `_depositAmount`, and also sends ETH to prefund the wallet.
2. The attacker sees the transaction in the mempool and submits the **same** `createAccount()` call with:
   - The same `_deviceUniqueIdentifier` and `_deviceWalletOwnerKey`
   - Higher gas fees to ensure their transaction executes first.
3. The attacker's transaction executes first, successfully deploying the wallet without depositing ETH.
4. Victim's transaction executes second and reaches this check:

```
address wallet =
registry.uniqueIdentifierToDeviceWallet(_deviceUniqueIdentifier);
if (wallet != address(0)) {
    return DeviceWallet(payable(wallet));
}
```

Since the wallet already exists, the function returns the attacker's deployed wallet instead of deploying a new one.

However, the victim has already sent ETH, which is now stuck in the factory contract forever.

A malicious user can do this attack constantly. This would only cost him the price of the gas, while the honest user would lose all his ETH.

Recommendations

Revert the transactions if ETH is sent, but Wallet already exists.

```
    if (existingWallet != address(0)) {
        require(msg.value == 0, "ETH will be lost if wallet already
exists");
        return DeviceWallet(payable(existingWallet));
    }
```

The other option is to return the ETH if the Wallet already exists.

In this way, the honest user will not lose funds, and the attack will be rendered meaningless.

[H-01] Device wallet creation can be DoSed, making the whole protocol obsolete

Severity

Impact: Medium

Likelihood: High

Description

The DeviceWalletFactory::createAccount() function is permissionless, and it is responsible for deploying and registering the device wallets. The function takes 4 different parameters one of which is a string `_deviceUniqueIdentifier`. There are several checks that are performed in order to see whether a device wallet with the same parameters has already been registered.

```
        address wallet =
registry.uniqueIdentifierToDeviceWallet(_deviceUniqueIdentifier);
        if(wallet != address(0)) {
            return DeviceWallet(payable(wallet));
        }
```

A malicious user can frontrun the admin restricted DeviceWalletFactory::deployDeviceWalletAsAdmin() function, which internally calls the DeviceWalletFactory::createAccount() function, by calling the DeviceWalletFactory::createAccount() function with the same `_deviceUniqueIdentifier` parameter but a different `bytes32[2] memory _deviceWalletOwnerKey` parameter. A call to the DeviceWalletFactory::createAccount() function can also be frontrun. A DeviceWallet will be created however the owners will be different. Given the fact that the main purpose of the protocol is to allow users to create DeviceWallets and interact with them, dosing the creation of wallets by paying only the transaction fees is a vulnerability with high severity. This attack can also be utilized to DOS the LazyWalletRegistry::batchPopulateHistory() and LazyWalletRegistry::deployLazyWalletAndSetESIMIdentifier() functions.

Recommendations

Consider whether the uniqueIdentifierToDeviceWallet mapping is necessary.

Also, consider binding `salt` to `msg.sender` to prevent unauthorized deployments:

```
function createAccount(
    string memory _deviceUniqueIdentifier,
    bytes32[2] memory _deviceWalletOwnerKey,
    uint256 _salt,
    uint256 _depositAmount
) public payable returns (DeviceWallet deviceWallet) {
    bytes32 uniqueSalt = keccak256(abi.encodePacked(msg.sender, _salt));

    address addr = getAddress(_deviceWalletOwnerKey,
_deviceUniqueIdentifier, uniqueSalt);

    .....
}
```

[H-02] The `deployLazyWalletAndSetESIMIdentifier` function won't send ETH to the registry and will revert, resulting in a core protocol functionality not working

Severity

Impact: Medium

Likelihood: High

Description

The LazyWalletRegistry::deployLazyWalletAndSetESIMIdentifier() function is an admin-controlled function that is supposed to deploy a device wallet and eSIM wallet for a certain user.

```
    function deployLazyWalletAndSetESIMIdentifier(
        bytes32[2] memory _deviceOwnerPublicKey,
        string calldata _deviceUniqueIdentifier,
        uint256 _salt,
        uint256 _depositAmount
    ) external payable onlyESIMWalletAdmin returns (address, address[]
memory) {
        require(_depositAmount == msg.value, "Incorrect ETH");
        require(isLazyWalletDeployed(_deviceUniqueIdentifier) == false,
"Already deployed");

        ...

        (deviceWallet, eSIMWallets) = registry.deployLazyWallet(
            _deviceOwnerPublicKey,
            _deviceUniqueIdentifier,
            _salt,
            eSIMUniqueIdentifiers,
            listOfDataBundleDetails,
            _depositAmount
        );

        ...
    }
```

As can be seen from the code snippet above, the function is payable and it requires the msg.value to be equal to the `_depositAmount` parameter, which is used in the RegistryHelper::deployLazyWallet() function, however, the msg.value won't be transferred to the `Registry.sol` instance and the function will revert. The RegistryHelper::deployLazyWallet() function also won't transfer any ETH sent to it when the call is made to the `DeviceWalletFactory.sol` contract, and thus revert as well. This is an important protocol functionality, however the functions will always revert if the msg.value sent is bigger than 0 when the call to them is made. This makes core protocol functionality obsolete.

Recommendations

Consider using the following pattern to transfer the msg.value successfully

```
    registry.deployLazyWallet{value:msg.value}(...)
```

[M-01] Excess ETH not returned in `deployDeviceWalletAsAdmin` and `createAccount`

Severity

Impact: Medium

Likelihood: Low

Description

In `DeviceWalletFactory.sol`, both `createAccount()` and `deployDeviceWalletAsAdmin()` functions do not properly return excess ETH when prefunding a `DeviceWallet`.

```
    function deployDeviceWalletAsAdmin(
        string memory _deviceUniqueIdentifier,
        bytes32[2] memory _deviceWalletOwnerKey,
        uint256 _salt,
        uint256 _depositAmount
    ) public payable onlyAdmin returns (Wallets memory) {
        address deviceWalletAddress = address(
            createAccount(
                _deviceUniqueIdentifier,
                _deviceWalletOwnerKey,
                _salt,
                _depositAmount
            )
        );

        ESIMWalletFactory eSIMWalletFactory =
registry.eSIMWalletFactory();
        address eSIMWalletAddress =
eSIMWalletFactory.deployESIMWallet(deviceWalletAddress, _salt);
        DeviceWallet(payable(deviceWalletAddress)).addESIMWallet(
            eSIMWalletAddress,
            true
        );
        emit DeviceWalletDeployed(deviceWalletAddress, eSIMWalletAddress,
_deviceWalletOwnerKey);

        return Wallets(deviceWalletAddress, eSIMWalletAddress);
    }
```

```
    function createAccount(
        string memory _deviceUniqueIdentifier,
        bytes32[2] memory _deviceWalletOwnerKey,
        uint256 _salt,
        uint256 _depositAmount
    ) public payable returns (DeviceWallet deviceWallet) {
        require(
            bytes(_deviceUniqueIdentifier).length != 0,
            "DeviceIdentifier cannot be empty"
        );

        address addr = getAddress(
            _deviceWalletOwnerKey,
            _deviceUniqueIdentifier,
            _salt
        );

        // Check if the device identifier is actually unique
        address wallet =
registry.uniqueIdentifierToDeviceWallet(_deviceUniqueIdentifier);
        if(wallet != address(0)) {
            return DeviceWallet(payable(wallet));
        }

        // Check if P256 public key is actually unique
        bytes32 keyHash = keccak256(abi.encode(_deviceWalletOwnerKey[0],
_deviceWalletOwnerKey[1]));
        wallet = registry.registeredP256Keys(keyHash);
        if(wallet != address(0)) {
            return DeviceWallet(payable(wallet));
        }

        uint256 codeSize = addr.code.length;
        if (codeSize > 0) {
            return DeviceWallet(payable(addr));
        }

        // Prefund the account with msg.value
        if (msg.value > 0 && _depositAmount <= msg.value) {
            entryPoint.depositTo{value: _depositAmount}(addr);
        }

        deviceWallet = DeviceWallet(
            payable(
                new BeaconProxy{salt : bytes32(_salt)}(
                    address(beacon),
                    abi.encodeCall(
                        DeviceWallet.init,
                        (address(registry), _deviceWalletOwnerKey,
_deviceUniqueIdentifier)
                    )
                )
            )
        );

        registry.updateDeviceWalletInfo(address(deviceWallet),
_deviceUniqueIdentifier, _deviceWalletOwnerKey);
    }
```

Any excess ETH sent during deployment remains trapped in the factory contract, making it inaccessible to the sender.

We should note a few things.

The function `deployDeviceWalletForUsers()` correctly tracks and returns excess ETH, but the other two functions do not, leading to an inconsistent user experience.

This function deploys multiple device wallets at once. When the admin calls it, it calls the above two functions accordingly, and the excess ETH is refunded to the admin.

```
        // return unused ETH
        if(availableETH > 0) {
            (bool success,) = msg.sender.call{value: availableETH}("");
            require(success, "ETH return failed");
        }
```

Recommendations

The excess ETH is not refunded only when the admin calls `deployDeviceWalletAsAdmin()` or a user calls `createAccount()`.

You need to be careful about whether and how you implement the refund of the excess ETH because doing so can break `deployDeviceWalletForUsers()`, which already refunds the excess ETH and calls the other two functions.

[M-02] Corruptible Upgradability Pattern

Severity

Impact: High

Likelihood: Low

Description

The `Registry.sol` contract is a UUPSUpgradeable contract that inherits from the `RegistryHelper.sol` contract. When creating upgradable contracts that inherit from other contracts, it is important that there are storage gaps in case storage variables are added to inherited contracts. If an inherited contract is a stateless contract (i.e. it doesn't have any storage) then it is acceptable to omit a storage gap since these function similarly to libraries and aren't intended to add any storage. The issue is that `Registry.sol` inherits from a contract that contains storage that doesn't contain any gaps such as `RegistryHelper.sol`. These contracts can pose a significant risk when updating a contract because they can shift the storage slots of all inherited contracts.

Recommendations

Consider adding storage gaps to the `RegistryHelper.sol` contract.

[L-01] Lack of signature expiration in `verifySignature`

Description

The `P256Verifier.sol` contract is responsible for validating digital signatures using `WebAuthn`.

```
    function verifySignature(
        bytes memory message,
        bool requireUserVerification,
        WebAuthnSignature memory webAuthnSignature,
        uint256 x,
        uint256 y
    ) public view returns (bool) {

        return
            WebAuthn.verifySignature({
                challenge: message,
                requireUV: requireUserVerification,
                webAuthnSignature: webAuthnSignature,
                x: x,
                y: y
            });
    }
```

However, there is no expiration mechanism in the signature verification process.

Setting an expiration date for digital signatures is important in volatile markets where the trend may shift quickly. It also limits the risk of an exploit in case the digital signature algorithm is compromised.

Recommendations

It is recommended that the signature verification process include a deadline parameter to mitigate the risk of never-expiring signatures and enhance the system's security.

[L-02] Missing zero address check in `withdrawDepositTo`

Description

The function `withdrawDepositTo()` in `Account4337.sol` does not validate the withdrawal address, allowing funds to be sent to the zero address (`address(0)`).

```
    function withdrawDepositTo(address payable withdrawAddress, uint256
amount) public onlySelf {
        entryPoint.withdrawTo(withdrawAddress, amount);
    }
```

If this happens, the funds will be permanently lost, as `address(0)` is an irretrievable burn address in Ethereum.

Recommendations

Add a zero address check to `withdrawDepositTo()`.

[L-03] Discrepancy in nat spec and code implementation

The DeviceWallet::deployESIMWallet() function allows for a eSIM wallet to be created and registered with the DeviceWallet itself. From the comments above the function implementation, we see that the device wallet owner is supposed to be the one calling this function, however, as can be seen from the code snippet below the eSIMWalletAdmin (which is an address controlled by the protocol) is the only one who is allowed to call this function:

```
    /// @notice Allow device wallet owner to deploy new eSIM wallet
    /// @dev Don't forget to call setESIMUniqueIdentifierForAnESIMWallet
function after deploying eSIM wallet
    /// @param _hasAccessToETH Set to true if the eSIM wallet is allowed
to pull ETH from this wallet.
    /// @return eSIM wallet address
    function deployESIMWallet(
        bool _hasAccessToETH,
        uint256 _salt
    ) external onlyESIMWalletAdmin returns (address) {
```

Recommendation

Consider using the `onlySelf()` modifier instead of the `onlyESIMWalletAdmin()` modifier.

[L-04] Removed DeviceWallet can change eSIMWallet configuration in the registry

When the ownership of a ESIMWallet is changed, the previous owner(DeviceWallet) can still call the DeviceWallet::removeESIMWallet() function and set the `isESIMWalletValid` mapping to address 0 and the `isESIMWalletOnStandby` mapping to true. This may be problematic if this information is important for off-chain purposes.

Recommendation

When the ownership of ESIMWallet is changed, consider setting the `isValidESIMWallet` mapping of the previous owner(DeviceWallet) to false before the ownership transfer is complete.