Veridise. Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

Daimo

Veridise Inc.
October 6, 2023

► **Prepared For:**

Daimo
https://daimo.xyz/

► **Prepared By:**

Daniel Domínguez Álvarez
Jacob Van Geffen
Bryan Tan

► **Contact Us:** contact@veridise.com

► **Version History:**

Oct. 6, 2023 V1.00

Contents

1 Executive Summary

From Sep. 13, 2023 to Sep. 26, 2023, Daimo engaged Veridise to review the security of several major components of their Daimo project, a wallet app (for iOS and Android) backed by an on-chain Ethereum smart contract compliant with EIP-4337* account abstraction. Veridise conducted the assessment over 6 person-weeks, with 3 engineers reviewing code over 2 weeks on commit `f0dc56d`. The security assessment was performed in the same audit as that of Daimo's P256Verifier project†, which the Daimo project depends on. The auditing strategy involved a tool-assisted analysis of the source code performed by Veridise engineers as well as extensive manual auditing.

Project Summary. The security assessment covered the following components of Daimo: the on-chain smart contracts used in Daimo, the `daimo-userop` package, and the `daimo-expo-enclave` package. Specifically, the Daimo smart contracts consist of the `DaimoAccount` EIP-4337 smart contract, whose authentication protocol uses ECDSA public keys on NIST curve P-256; the `EphemeralNotes` contract that implements the on-chain part of the payment links feature; and the `DaimoNameRegistry` for mapping human-readable names to on-chain addresses.

The iOS and Android apps (out-of-scope of this audit) are each implemented as a React Native application. The `daimo-userop` package is a TypeScript library used in the apps to invoke the Daimo smart contracts. Lastly, the `daimo-expo-enclave` package provides a TypeScript wrapper around Swift/Kotlin code for accessing the cryptographic functions of the phone's secure enclave.

Code assessment. The Daimo developers provided the source code of Daimo for review‡. The source code appears to be original and written by the developers. It contains some documentation in the form of READMEs and documentation comments on functions and variables. The source code also contains a test suite, which the Veridise auditors noted provides decent test coverage over the major features of each package, including tests of known good situations and some failure cases. Several files in the source code also indicate that the developers use linting and static analysis tools such as ESLint and Solhint.

Summary of issues detected. The audit uncovered 15 issues, 2 of which are assessed to be of medium severity by the Veridise auditors. Specifically, when the phone is unlocked and does not have a secure enclave available, then the user is not prompted for re-authentication and/or re-authorization before a transaction is signed for some devices (V-DMO-VUL-001, V-DMO-VUL-002). The Veridise auditors also identified 5 low-severity issues, including a lack of expiry times on user operation signatures (V-DMO-VUL-007) and lack of status validation on ERC20 token transfers (V-DMO-VUL-006). In addition, the Veridise auditors identified

* Also known as "Account Abstraction Using Alt Mempool": `https://eips.ethereum.org/EIPS/eip-4337`
† The audit report for P256Verifier can be found on our website: `https://veridise.com/audits`
‡ The source code is publicly available at `https://github.com/daimo-eth/daimo`

6 warnings and 2 informational issues. The Daimo developers resolved all of the reported issues.

Disclaimer. We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
Daimo	`f0dc56d`	Solidity, TypeScript, Swift, Kotlin	Ethereum, React Native, iOS, Android

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Sep. 13 - Sep. 26, 2023	Manual & Tools	3	6 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Resolved
Critical-Severity Issues	0	0
High-Severity Issues	0	0
Medium-Severity Issues	2	2
Low-Severity Issues	5	5
Warning-Severity Issues	6	6
Informational-Severity Issues	2	2
TOTAL	15	15

Table 2.4: Category Breakdown.

Name	Number
Data Validation	4
Logic Error	3
Maintainability	3
Access Control	2
Replay Attack	1
Frontrunning	1
Gas Optimization	1

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of the major components of Daimo as described in Section 1. In our audit, we sought to answer questions such as:

- Does the `DaimoAccount` have any weaknesses in its signature scheme?
- Does the `DaimoAccount` correctly validate its signers?
- Can funds be locked in the `EphemeralNotes` contract?
- Does the the `DaimoNameRegistry` correctly handle name-address registrations?
- Are there ways for the app to sign a transaction without obtaining the user's authorization for the transaction?
- Are there potential flaws in the way that transactions are constructed?

3.2 Audit Methodology & Scope

Audit Methodology. To address the questions above, our audit involved a combination of human experts and automated program analysis & testing tools. In particular, we conducted our audit with the aid of the following techniques:

- *Static analysis.* To identify potential common vulnerabilities, we leveraged our custom smart contract analysis tool Vanguard, as well as the open-source tool Slither. These tools are designed to find instances of common smart contract vulnerabilities, such as reentrancy and uninitialized variables.
- *Fuzzing/Property-based Testing.* We also leverage fuzz testing to determine if the protocol may deviate from the expected behavior. To do this, we formalize the desired behavior of the protocol as [V] specifications and then use our fuzzing framework OrCa to determine if a violation of the specification can be found.

Scope. The scope of this audit is limited to the following folders of the source code provided by the Daimo developers:

- `packages/contract/contract/src`
- `packages/daimo-expo-enclave`
- `packages/daimo-userop/src`

All other packages in the provided source code (e.g., the application code in `app/`) are *not* in the scope of the audit. During the audit, the Veridise auditors referred to the excluded files but assumed that they have been implemented correctly.

Methodology. Veridise auditors reviewed relevant specifications such as EIP-4337, inspected the provided tests, and read the Daimo documentation. They then began a manual audit of the code assisted by both static analyzers and automated testing. During the audit, the Veridise auditors regularly met with the Daimo developers to ask questions about the code.

3.3 Classification of Vulnerabilities

When Veridise auditors discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise. Table 3.1 shows how our auditors weigh this information to estimate the severity of a given issue.

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

In this case, we judge the likelihood of a vulnerability as follows in Table 3.2:

Table 3.2: Likelihood Breakdown

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

In addition, we judge the impact of a vulnerability as follows in Table 3.3:

Table 3.3: Impact Breakdown

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

4 Vulnerability Report

In this section, we describe the vulnerabilities found during our audit. For each issue found, we log the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.). Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-DMO-VUL-001	iOS FallbackKeyManager does not require user pr.	Medium	Fixed
V-DMO-VUL-002	Android FallbackKeyManager does not require use	Medium	Fixed
V-DMO-VUL-003	Zero name/address can be registered	Low	Fixed
V-DMO-VUL-004	Initial public key not validated	Low	Intended Behavior
V-DMO-VUL-005	addSigningKey does not check public key validity	Low	Intended Behavior
V-DMO-VUL-006	Unchecked ERC20 token transfer success status	Low	Fixed
V-DMO-VUL-007	User operation signatures have no expiry	Low	Fixed
V-DMO-VUL-008	Inconsistent doc comment on createEphemeralNot	Warning	Fixed
V-DMO-VUL-009	DaimoNameRegistry does not disable initializers	Warning	Fixed
V-DMO-VUL-010	Subclassing OpenZeppelin contracts with upgrada	Warning	Fixed
V-DMO-VUL-011	Consider adding atomic approval change methods	Warning	Acknowledged
V-DMO-VUL-012	Undocumented assumption that notes contract use	Warning	Fixed
V-DMO-VUL-013	fromHex does not validate nonceType	Warning	Fixed
V-DMO-VUL-014	Improve Swift code quality with guard	Info	Fixed
V-DMO-VUL-015	External call to register() can be safely repla...	Info	Fixed

4.1 Detailed Description of Issues

4.1.1 V-DMO-VUL-001: iOS FallbackKeyManager does not require user presence

Severity	Medium	**Commit**	f0dc56d
Type	Access Control	**Status**	Fixed
File(s)	`GenericPasswordStore.swift`		
Location(s)	See description		
Confirmed Fix At	`https://github.com/daimo-eth/daimo/pull/274`		

Elements stored in the Keychain Services using `GenericPasswordStore` have a default level of access which allows apps to access elements while the device is unlocked. Following Apple's documentation for the Keychain Services API, for payment applications like Daimo, it is recommended to require user presence while accessing elements of the keychain. Such configuration would require the user to be prompted for authorization when account keys are retrieved for use in signing transactions. This prevents unauthorized users from accessing account keys if they attempt to do so when the device is unlocked (for example, an unattended child playing around with the device, or theft of the device while it is unlocked).

Impact This issue only affects the `FallbackKeyManager`, which relies completely on `GenericPasswordStore` and thus will not enforce user presence on key retrieval. In contrast, the `SecureEnclaveKeyManager` will require user presence when accessing the Secure Enclave.

If an unauthorized user accesses the application when device is unlocked, they can access elements of the keychain and perform unauthorized actions in the app, such as signing transactions.

Since the most common key manager for most users is the one based on Secure Enclave, this issue will likely only affect users whose phones lack a Secure Enclave and must rely on the fallback method.

Recommendation Modify the keychain access code to require user presence when the element is unlocked. According to Apple's documentation, the queries to the keychain would need to be modified as follows:

First, set up the access control with the following line of code:

```
var error: NSError?
let access = SecAccessControlCreateWithFlags(NULL,  // Use the default allocator.
                                             kSecAttrAccessibleWhenUnlocked,
                                             kSecAccessControlUserPresence,
                                             &error);
```

Use `access` in the subsequent queries:

```
var query: [String: Any] = [ /* ... */
                            kSecAttrAccessControl as String: access,
                             /* ... */ ]
```

This will cause the app to prompt the user for authentication using the available methods (biometry, password, etc) whenever the keychain item is accessed.

4.1.2 V-DMO-VUL-002: Android FallbackKeyManager does not require user presence

Severity	Medium	**Commit**	f0dc56d
Type	Access Control	**Status**	Fixed
File(s)	`FallbackKeyManager.kt`		
Location(s)	createSigningPrivkey()		
Confirmed Fix At	`https://github.com/daimo-eth/daimo/pull/289`		

Similar to V-DMO-VUL-001, this issue allows messages to be signed without user presence in cases where `FallbackKeyManager` is used. This happens because no authentication requirement is set for `KeyGenParameterSpec.Builder` in the `createSigningPrivkey` method, nor is the user prompted for authentication in the `sign()` method.

```
internal fun createSigningPrivkey(accountName: String) {
  var params = KeyGenParameterSpec.Builder(accountName, KeyProperties.PURPOSE_SIGN or
     KeyProperties.PURPOSE_VERIFY)
    .setAlgorithmParameterSpec(ECGenParameterSpec("secp256r1"))
    .setDigests(KeyProperties.DIGEST_SHA256)
    .build()

  var keyPairGenerator = KeyPairGenerator.getInstance(KeyProperties.KEY_ALGORITHM_EC,
     KEYSTORE_PROVIDER)
  keyPairGenerator.initialize(params)
  keyPairGenerator.generateKeyPair()
}
```

Snippet 4.1: Definition of `createSigningPrivkey()`

Impact Note that the `FallbackKeyManager` is only used if the call to `ExpoEnclaveModule.hasBiometrics()` returns false. In general, this will occur on phones with API level 29 or lower, as the combination `BIOMETRIC_STRONG | DEVICE_CREDENTIAL` is not supported on those levels (see documentation). On API level 30 or above, the `FallbackKeyManager` will only be used if the user does not have any biometrics or device credentials enabled.

```
internal fun hasBiometrics(): Boolean {
  val biometricManager = BiometricManager.from(context)
  return biometricManager.canAuthenticate(BiometricManager.Authenticators.
    BIOMETRIC_STRONG or BiometricManager.Authenticators.DEVICE_CREDENTIAL) ==
    BiometricManager.BIOMETRIC_SUCCESS
}
```

Snippet 4.2: Definition of `hasBiometrics()`

When the `FallbackKeyManager` is used, unauthorized users may sign messages through the `FallbackKeyManager` while the phone is unlocked. Although these users cannot authenticate through biometric inputs, they should still be prompted for authentication through other means.

Recommendation Due to the instability of the `BiometricPrompt` API across API levels 28-30, there are several potential ways in which this issue can be addressed:

- In `FallbackKeyManager.sign()`, use the `createConfirmDeviceCredentialIntent()` method of `KeyguardManager` to launch an activity that prompts the user for their credentials (PIN, password, etc.), following the steps in the documentation. Note that the method is deprecated in API level 29 but has not been removed.
  Enforcing user authentication at the application level, rather than on the keystore object itself, may increase the complexity of the code, however.
- For API level 29 or lower, if additional security on the `FallbackKeyManager` keys is desired, the developers may want to consider constructing the parameters in `createSigningPrivkey()` in the following way:

```
var params = KeyGenParameterSpec.Builder(accountName, KeyProperties.PURPOSE_SIGN
     or KeyProperties.PURPOSE_VERIFY)
      .setAlgorithmParameterSpec(ECGenParameterSpec("secp256r1"))
      .setDigests(KeyProperties.DIGEST_SHA256)
      .setUserAuthenticationRequired(true)
      .setUserAuthenticationValidityDurationSeconds(some_seconds_as_int)
      .build()
```

  As noted in the documentation, setting required user authentication with a positive validity duration will cause a use of the key to throw an error, unless the user has been previously authenticated through a lock screen.
  It is important to also note that `setUserAuthenticationRequire(true)` may cause the key to be invalidated when new biometrics (e.g., fingerprints) are enrolled, locking users out of their accounts. However, setting a positive validity duration will not cause the key to be invalidated by biometric enrollment, as documented in the related method `setInvalidatedByBiometricEnrollment()`.
- For API level 28 or 29, the developers could modify `hasBiometrics()` to allow Class 3 biometrics (e.g., fingerprints) to be used if they are available, e.g.:

```
var authFlags = BiometricManager.Authenticators.BIOMETRIC_STRONG or
    BiometricManager.Authenticators.DEVICE_CREDENTIAL
if (Build.VERSION_SDK_INT <= Build.VERSION_CODES.Q) {
  // the combination BIOMETRIC_STRONG | DEVICE_CREDENTIAL is unsupported on API
    level < 30
  authFlags = BiometricManager.Authenticators.BIOMETRIC_STRONG
}
return biometricManager.canAuthenticate(authFlags) == BiometricManager.
    BIOMETRIC_SUCCES
```

  Note that this will also require the same authentication parameters to be passed to the `KeyGenParameterSpec.Builder` in `BiometricsKeyManager.createSigningPrivkey()`.
  One caveat of adding support like this is that if the user is currently accessing a key through the `FallbackKeyManager` and then decides to add a strong biometric, then the `BiometricsKeyManager` will always be used to generate the key in the future. Since the key generation parameters in `FallbackKeyManager` do not require user authentication, the existing key will not require user authentication when it is used to sign messages in `BiometricsKeyManager`.

Developer Response The developers noted that they only intend to support phones with API level 28 and above, and they documented the intended behavior of the expo module across different devices in the following GitHub issue: `https://github.com/daimo-eth/daimo/issues/288`

The developers will add the `createConfirmDeviceCredentialIntent()` prompt to the `FallbackKeyManager`. The developers do not intend to apply the other recommendations due to the complexity involved. Instead, the developers will force the `FallbackKeyManager` to be used on API levels 28-29 and the `BiometricsKeyManager` to be used on API levels 30 and above.

4.1.3 V-DMO-VUL-003: Zero name/address can be registered

Severity	Low	**Commit**	f0dc56d
Type	Data Validation	**Status**	Fixed
File(s)	`DaimoNameRegistry.sol`		
Location(s)	registerName()		
Confirmed Fix At	`https://github.com/daimo-eth/daimo/pull/268`		

The `DaimoNameRegistry` contract maintains a one-to-one relation between names (of maximum length 32 bytes) and addresses. This relation is represented by the `_names` and `_addrs` mappings, where a name/address pair `(n, a)` is registered if and only if `_names[a] == n` and `_addrs[n] == a`. Any address may invoke the `register()` method to add a unique name/address pair to the contract.

If `_names[a]` or `_addrs[n]` is set to zero or the zero bytes, then conceptually the pair `(n, a)` is considered to be unregistered. However, the `register()` method allows registering a pair where one component is zero and the other is nonzero.

```
/// Enforces uniqueness. Doesn't do any validation on name. The app
/// validates names both for claiming and lookup, so there's no advantage
/// to registering an invalid name onchain (will be ignored / unusable).
function register(bytes32 name, address addr) public {
    require(_addrs[name] == address(0), "NameRegistry: name taken");
    require(_names[addr] == bytes32(0), "NameRegistry: addr taken");
    _addrs[name] = addr;
    _names[addr] = name;
    emit Registered(name, addr);
}

/// Registers msg.sender under a given name.
function registerSelf(bytes32 name) external {
    this.register(name, msg.sender);
}
```

Snippet 4.3: Definition of the relevant functions in `DaimoNameRegistry`

Impact If either `(n, 0)` or `(0, a)` is registered for nonzero `n` or `a`, then the relation will no longer be a one-to-one relation. This may violate assumptions made by third-party applications that use the `resolveAddr()` and `resolveName()` methods. Specifically, the two methods treat the return value zero as a special value representing a missing entry. If, for example, the pair `(n, 0)` is registered, then `resolveAddr(n)` will return `address(0)`, which would indicate that `(n, 0)` is "missing" despite it being registered.

Furthermore, the issue may cause a denial-of-service problem in the `forceRegister()` method. The `owner` of the registry can call `forceRegister()` in order to override a name/address pair. This is implemented by (1) clearing any entry with a nonzero `_addrs[name]`; and then (2) calling `register()`. However, if `(name, address(0))` is already registered for some nonzero name, then `prevAddr` will be zero (so the existing entry will not be cleared), and the call to `register()` will always revert when checking that `_names[address(0)]` is zero.

Because `register()` does not have any access controls, it would be very easy for the zero address to be registered, allowing one name to be irrevocably denied to future registrations.

```
/// Looks up the address for a given name, or address(0) if missing.
function resolveAddr(bytes32 name) external view returns (address) {
    return _addrs[name];
}

/// Looks up the name for a given address, or bytes32(0) if missing.
function resolveName(address addr) external view returns (bytes32) {
    return _names[addr];
}

/// Allow owner to override a claimed name.
/// The purpose of this is to prevent name-squatting early on.
/// Eventually, ownership of the NameRegistry will be burned to ossify.
function forceRegister(bytes32 name, address addr) external onlyOwner {
    address prevAddr = _addrs[name];
    if (prevAddr != address(0)) {
        assert(_names[prevAddr] == name); // invariant
        _names[prevAddr] = bytes32(0);
        _addrs[name] = address(0);
    }
    register(name, addr);
}
```

Snippet 4.4: Affected functions in `DaimoNameRegistry`

Recommendation Require both of the `name` and `addr` function parameters to be nonzero in `register()` and `forceRegister()`.

4.1.4 V-DMO-VUL-004: Initial public key not validated

Severity	Low	**Commit**	f0dc56d
Type	Data Validation	**Status**	Intended Behavior
File(s)	`DaimoAccount.sol`		
Location(s)	initialize()		
Confirmed Fix At	N/A		

The `DaimoAccount.initialize()` method takes as an argument a NIST P-256 public key to use as the initial key. However, it assumes that the key is valid and does not perform any validation on the key. SEC 1 version 2.0, section 3.2.2 [PDF] recommends that "it is either necessary or desirable for an entity using an elliptic curve public key to receive an assurance that the public key is valid."

While we note that the typical usage scenario for a `DaimoAccount` is to construct it with one of the APIs or applications provided by Daimo (arguably a "trusted party" that ensures that the public key is valid), a third-party application that initializes a `DaimoAccount` may not necessarily ensure that the provided public key is valid.

```
function initialize(
    uint8 slot,
    bytes32[2] calldata key,
    Call[] calldata initCalls
) public virtual initializer {
    keys[slot] = key;
    numActiveKeys = 1;
```

Snippet 4.5: Relevant lines in `initialize()`

Impact If the initial signing key is not a valid P-256 public key, then the account creator will be permanently locked out of the account. Any gas, tokens, or other valuables that the account has been initialized with will be locked.

Recommendation Validate that the initial key is a valid public key for curve P-256. To avoid implementation mistakes, it is recommended to reuse the validation logic in `addSigningKey()` (esp. after applying the recommendation in V-DMO-VUL-005).

Developer Response The developers noted that they are making the following trade-off:

> The validity of *public* keys is an invariant that we do not think makes sense to enforce in the on-chain contract.
>
> The similar but actual invariant we would like to maintain for the user/owner of the account is that signing keys authorised to the account are ones the user owns the private key of / can produce valid signatures from (note that besides simply being valid *public* keys, this requires that the user must also be able to sign using the key via the app). This is something that the application enforces on behalf of the user off-chain.

Note also that since the transactions for signing key changes will always originate from the user themselves, key validity seems like a "application level" invariant to enforce, rather than on-chain contract level.

Thus, we would prefer not to add additional complexity/gas cost to enforce suggested weaker invariant that actually does not add sufficient real guarantee for the user of the app.

4.1.5 V-DMO-VUL-005: addSigningKey does not check public key validity

Severity	Low	**Commit**	f0dc56d
Type	Data Validation	**Status**	Intended Behavior
File(s)	`DaimoAccount.sol`		
Location(s)	addSigningKey()		
Confirmed Fix At	N/A		

The `addSigningKey()` method is used to add a ECDSA public key (on curve P-256) to the account. The `key` must be provided as a point `(x, y)` on the curve P-256 in affine coordinates. However, `addSigningKey()` only checks that `x` is nonzero and does not fully validate the given public key. The "Elliptic Curve Public Key Validation Primitive" procedure described in SEC 1 Ver. 2.0 [PDF], Section 3.2.2.1 includes the following steps that are *not* in the implementation of `addSigningKey()`:

- Checking that `x` and `y` are valid integer representations of elements of $\{F\}_p$ (i.e., by checking that `x` and `y` are in the inclusive range `[0, p - 1]`).
- Checking that `(x, y)` is a point on curve P-256.

```
function addSigningKey(uint8 slot, bytes32[2] memory key) public onlySelf {
    require(keys[slot][0] == bytes32(0), "key already exists");
    require(key[0] != bytes32(0), "new key cannot be 0");
    require(numActiveKeys < maxKeys, "max keys reached");
    keys[slot] = key;
    numActiveKeys++;
    emit SigningKeyAdded(this, slot, key);
}
```

Snippet 4.6: Implementation of `addSigningKey()`

Note that there is also a step that checks that `(x, y)` is a scalar multiple of the base point of the curve; however, this check can be skipped for curve P-256, as the the base point is a generator for the curve.

Impact If a user of the account adds an invalid public key and removes all other public keys, then all users will be permanently locked out of the account.

Recommendation Validate that the given `key` is a valid ECDSA public key on curve P-256 by also including the checks described above.

Developer Response See the developer response in V-DMO-VUL-004.

4.1.6 V-DMO-VUL-006: Unchecked ERC20 token transfer success status

Severity	Low	**Commit**	f0dc56d
Type	Logic Error	**Status**	Fixed
File(s)	`DaimoEphemeralNotes.sol`		
Location(s)	createNote(), claimNote()		
Confirmed Fix At	`https://github.com/daimo-eth/daimo/pull/267`		

When the `EphemeralNotes` contract is used to create or claim a note, the contract will call `ERC20.transferFrom()` or `ERC20.transfer()`, respectively. These transfer methods may return a boolean value indicating whether the transfer is successful; however, this value is not checked.

```
function createNote(address _ephemeralOwner, uint256 _amount) external {
    // ...
    token.transferFrom(msg.sender, address(this), _amount);
}

function claimNote(
    address _ephemeralOwner,
    bytes memory _signature
) external {
    // ...
    token.transfer(msg.sender, note.amount);
}
```

Snippet 4.7: Relevant lines in `createNote()` and `claimNote()`

```
function transfer(address _to, uint256 _value) public returns (bool success)

function transferFrom(address _from, address _to, uint256 _value) public returns (
    bool success)
```

Snippet 4.8: Function signatures of `transfer` and `transferFrom` in the ERC-20 specification.

Impact An ephemeral note may be successfully created even if the `transferFrom()` call in `createNote` returns false (i.e., transfer unsuccessful). A user may be able to claim such a note and transfer tokens out of the notes contract, even if no tokens were transferred in during the creation of the note.

Recommendation Use OpenZeppelin's `SafeTransfer` library, which will check the return value and correctly handle ERC-20 token contracts that do not return a boolean success value.

4.1.7 V-DMO-VUL-007: User operation signatures have no expiry

Severity	Low	**Commit**	f0dc56d
Type	Replay Attack	**Status**	Fixed
File(s)	`DaimoAccount.sol`		
Location(s)	_validateUseropSignature()		
Confirmed Fix At	`https://github.com/daimo-eth/daimo/pull/270`		

The signatures of user operations for a `DaimoAccount` consists of 65 bytes in the following format:

- A 1 byte index indicating which public key of the `DaimoAccount` to use to validate the signature.
- A 32 byte integer containing the `r` component of the ECDSA signature.
- A 32 byte integer containing the `s` component of the ECDSA signature.

Crucially, this signature does not include information that indicates the time period over which the signature is valid, meaning that any signature is valid forever. This could increase the risk of replay attacks. For example, if a user operation is submitted to a bundler's mempool, the bundler decides not to include the user operation (e.g., for reasons such as insufficient gas), and the user no longer wants the user operation to be executed as a result, then a malicious actor may be able to resubmit the operation at a later point in the future.

Impact Each EIP-4337 user operation has a `nonce` that is used to protect against replay attacks (see here for details). Specifically, the nonce consists of a pair `(key, sequence)` such that the 64-bit `sequence` must be strictly increasing among all user operations with the same `key`.

However, the `nonce` values generated by the `daimo-userop` package always have `sequence` set to 0, meaning that the signatures generated by `daimo-userop` will not be able to use the replay attack protection built into EIP-4337. This may expose `DaimoAccounts` to the attack scenario described above.

Recommendation Include a timestamp in the signature that indicates when the user operation expires, and change the `_validateSignature()` method to reject the signature if the current time is after the expiry time.

Developer Response In response, developers stated:

> I agree it's a good feature and we'll add it.
>
> I don't think it's accurate to call this a replay attack. A bundler can't include an op twice, or replay it on a different chain—it can only delay an op, including it onchain later than intended.
>
> EOA transactions (which, unlike 4337 userops, don't have a built-in `validUntil` mechanism) are similarly delayable.

The auditors agree that the user op cannot be executed twice by the bundler or executed again on a different chain. However, it is the *submission* to the mempool that is being replayed in the scenario described in this issue.

4.1.8 V-DMO-VUL-008: Inconsistent doc comment on createEphemeralNote

Severity	Warning	**Commit**	f0dc56d
Type	Maintainability	**Status**	Fixed
File(s)	`daimo-userop/src/index.ts`		
Location(s)	createEphemeralNote()		
Confirmed Fix At	`https://github.com/daimo-eth/daimo/pull/276`		

A documentation comment on the `DaimoOpSender.createEphemeralNote()` function implies that the method will send a transaction that includes a call to the `approve()` function of the ERC20 token of the note. However, the user operation constructed by `createEphemeralNote()` does not include any such calls to `approve()`.

```
/**
 * Creates an ephemeral note with given value.
 * Infinite-approves the notes contract first, if necessary.
 * Returns userOpHash.
 **/
public async createEphemeralNote(
  ephemeralOwner: `0x${string}`,
  amount: `${number}`,
  opMetadata: DaimoOpMetadata
) {
```

Snippet 4.9: The documentation comment on `createEphemeralNote()`

Impact If the comment is incorrect, it may mislead a developer into unintentionally inserting excessive ERC20 approval calls. If the implementation is incorrect, the transaction may be reverted due to missing approvals.

Recommendation Ensure that the comment and the implementation are consistent.

Developer Response The developers noted that the comment is outdated; the initial deployment of the account already includes an `approve()` call, so that the `daimo-userop` package can assume that the ephemeral notes contract is already approved.

4.1.9 V-DMO-VUL-009: DaimoNameRegistry does not disable initializers

Severity	Warning	**Commit**	f0dc56d
Type	Logic Error	**Status**	Fixed
File(s)	`DaimoNameRegistry.sol`		
Location(s)	See description		
Confirmed Fix At	`https://github.com/daimo-eth/daimo/pull/278`		

The `DaimoNameRegistry` contract does not declare a constructor. However, it subclasses from OpenZeppelin's `Initializable` contract, whose documentation mentions the following:

> Avoid leaving a contract uninitialized.
>
> An uninitialized contract can be taken over by an attacker. This applies to both a proxy and its implementation contract, which may impact the proxy. To prevent the implementation contract from being used, you should invoke the `_disableInitializers` function in the constructor to automatically lock it when it is deployed:

Recommendation Declare a constructor and have it call `_disableInitializers()`.

4.1.10 V-DMO-VUL-010: Subclassing OpenZeppelin contracts with upgradable proxies

Severity	Warning	**Commit**	f0dc56d
Type	Logic Error	**Status**	Fixed
File(s)	`DaimoNameRegistry.sol`		
Location(s)	See description		
Confirmed Fix At	`https://github.com/daimo-eth/daimo/pull/278`		

The `DaimoNameRegistry` contract is meant to be deployed through an upgradeable proxy. However, it subclasses from the normal OpenZeppelin contracts rather than the upgradeable versions. This may result in initialization problems when initializing or deploying the `DaimoNameRegistry`.

Impact Currently, this should have no impact. The `DaimoNameRegistry` subclasses from `Ownable` and `Initializable`. The constructor of the `Ownable` contract will set the owner to the `msg.sender` (i.e., the account deploying the contract). The `init()` method of `DaimoNameRegistry` will similarly set the owner to the `msg.sender`. Consequently, the logic contract's owner will be set to the deployer of the logic contract, and the proxy contract's owner will be set to the sender of the `init()` call.

```
/**
 * @dev Initializes the contract setting the deployer as the initial owner.
 */
constructor() {
    _transferOwnership(_msgSender());
}
```

Snippet 4.10: Constructor of `Ownable` in OpenZeppelin 4.8.1

```
function init() public initializer {
    _transferOwnership(msg.sender);
}
```

Snippet 4.11: Definition of `DaimoNameRegistry.init()`

However, if the `DaimoNameRegistry` is modified to subclass additional OpenZeppelin contracts that have constructor code, then the initialization code of those contracts may not be executed correctly when the `DaimoNameRegistry` is deployed through a proxy.

Recommendation

- To avoid future bugs, subclass from the `OwnableUpgradeable` contract of `https://github.com/OpenZeppelin/openzeppelin-contracts-upgradeable` instead of `Ownable`.
- In `init()`, invoke the `__Ownable_init()` (or `__Ownable_init_unchained()`) methods instead of manually calling `_transferOwnership()`.

4.1.11 V-DMO-VUL-011: Consider adding atomic approval change methods

Severity	Warning	**Commit**	f0dc56d
Type	Frontrunning	**Status**	Acknowledged
File(s)	`daimo-userop/src/index.ts`		
Location(s)	DaimoOpSender		
Confirmed Fix At	N/A		

The `DaimoOpSender` class contains a method `erc20approve()` that is used to construct and send a user operation that will call the `approve()` method on the ERC20 token of the `DaimoOpSender`. In scenarios where a user wishes to increase an existing allowance, the `erc20approve()` method may be vulnerable to ERC20 approval frontrunning attacks. There are no functions or methods on `DaimoOpSender` that allow a user to atomically increase or decrease an existing allowance.

Impact If a user sends a user operation that sets the approval amount with the intention of increasing an existing allowance, then recipient of the approval can frontrun the transaction to actually spend more tokens than they should be intended to. For example, suppose Alice has already approved Bob to use X tokens, and then Alice sends an `approve` call to change the approval to X+Y. If Bob sees the `approve` call, he can then spend the existing allowance X before the `approve` is finalized. Afterwards, he will be approved for a new amount X+Y, which he can then spend for a grand total of 2*X+Y tokens spent.

Recommendation If this issue is applicable, then `DaimoOpSender` should be extended with functionality to create user operations that atomically increase/decrease allowance. This may require the Daimo contracts to be modified to include methods that implement the allowance increases/decreases.

Developer Response The developers noted that increasing/decreasing allowance is not within their intended usage scenarios, but they will keep this issue in mind. Furthermore, they noted:

> We've removed the `erc20approve` function altogether, it was unused.
>
> DaimoOpSender constructs userops for each action supported in-app. The remaining functions are all used.

4.1.12 V-DMO-VUL-012: Undocumented assumption that notes contract uses the same token

Severity	Warning	**Commit**	f0dc56d
Type	Maintainability	**Status**	Fixed
File(s)	`daimo-userop/src/index.ts`		
Location(s)	DaimoOpSender		
Confirmed Fix At	`https://github.com/daimo-eth/daimo/pull/283`		

The `DaimoOpSender` class stores the address of the on-chain `DaimoEphemeralNotes` contract as well as the address of an ERC-20 token. Several methods on `DaimoOpSender`, such as `createEphemeralNote()`, implicitly assume that this ERC-20 token is the same one used in the `DaimoEphemeralNotes`. However, this assumption is not documented anywhere in the file.

Impact When ERC-20 token amounts are used as parameters of the methods of `DaimoOpSender`, they are assumed to be in the units of the token and have to be converted to `uint256` values for EVM calls. If the ERC-20 token does not match that of the notes contract, then (1) the wrong token will used for the note; and (2) a mismatch in the decimals may cause the wrong amount to be sent.

```
public async createEphemeralNote(
  ephemeralOwner: `0x${string}`,
  amount: `${number}`,
  opMetadata: DaimoOpMetadata
) {
  const parsedAmount = parseUnits(amount, this.tokenDecimals);
  console.log(`[OP] create ${parsedAmount} note for ${ephemeralOwner}`);

  const op = this.opBuilder.executeBatch(
    [
      {
        dest: this.notesAddress,
        value: 0n,
        data: encodeFunctionData({
          abi: Contracts.ephemeralNotesABI,
          functionName: "createNote",
          args: [ephemeralOwner, parsedAmount],
        }),
      },
    ],
    opMetadata
  );
```

Snippet 4.12: Example of how the token will be used.

Recommendation Clearly document this assumption in the code.

4.1.13 V-DMO-VUL-013: fromHex does not validate nonceType

Severity	Warning	**Commit**	f0dc56d
Type	Data Validation	**Status**	Fixed
File(s)	daimo-userop/src/nonce.ts		
Location(s)	DaimoNonceMetadata.fromHex()		
Confirmed Fix At	https://github.com/daimo-eth/daimo/pull/284		

The `DaimoNonceMetaData.fromHex()` function parses a `DaimoNonceMetadata` from a hex string. The first byte of the hex string corresponds to a `nonceType`, which corresponds to one of the entries in the `DaimoNonceType` enum. However, the `nonceType` is not checked to be a valid `DaimoNonceType` value.

```
public static fromHex(hexMetadata: Hex): DaimoNonceMetadata {
  assert(hexMetadata.length === 16 + 2);
  const nonceType = parseInt(hexMetadata.slice(2, 4), 16);
  const identifier = BigInt("0x" + hexMetadata.slice(4)) as bigint;
  return new DaimoNonceMetadata(nonceType, identifier);
}
```

Snippet 4.13: Implementation of `fromHex()`

```
export enum DaimoNonceType {
  Send = 0,
  CreateNote = 1,
  ClaimNote = 2,
  RequestResponse = 3,
  AddKey = 4,
  RemoveKey = 5,
  MAX = 255, // At most one byte
}
```

Snippet 4.14: Definition of `DaimoNonceType`

Impact If `fromHex()` is given a hex string that has an invalid nonce type value as its first byte, then `fromHex()` will still successfully return a `DaimoNonceMetadata`. This could potentially lead to subtle errors in code that uses `fromHex()`.

Recommendation Insert an assertion or other check that ensures that the `nonceType` is valid.

4.1.14 V-DMO-VUL-014: Improve Swift code quality with guard

Severity	Info	**Commit**	f0dc56d
Type	Maintainability	**Status**	Fixed
File(s)	`SecureEnclaveKeyManager.swift`, `FallbackKeyManager.swift`		
Location(s)	See description		
Confirmed Fix At	`https://github.com/daimo-eth/daimo/pull/265`		

There are two nil checks and subsequent unwrapping in `SecureEnclaveKeyManager.swift` and `FallbackKeyManager.swift` that could be replaced with a `guard` statement, as recommended for readability and maintainability. The usage of `guard` clarifies intentions regarding the state of a variable and places the exit conditions at the beginning of the function.

```
public func fetchPublicKey(accountName: String) throws -> String? {
    let readSigningPrivkey: SecureEnclave.P256.Signing.PrivateKey? = try self.store.
    readKey(account: accountName)
    if readSigningPrivkey == nil {
        return nil
    }
    let signingPrivkey = readSigningPrivkey!
    return signingPrivkey.publicKey.derRepresentation.hexEncodedString()
}
```

Snippet 4.15: The check at the beginning of `fetchPublicKey` in `SecureEnclaveKeyManager.swift` could be rewritten using a `guard`.

```
public func fetchPublicKey(accountName: String) throws -> String? {
    let readSigningPrivkey: P256.Signing.PrivateKey? = try self.store.readKey(account
    : accountName)
    if readSigningPrivkey == nil {
        return nil
    }
    let signingPrivkey = readSigningPrivkey!
    return signingPrivkey.publicKey.derRepresentation.hexEncodedString()
}
```

Snippet 4.16: The check at the beginning of `fetchPublicKey` in `FallbackKeyManager.swift` could be rewritten using a `guard`.

Impact There is no security impact.

Recommendation To improve maintainability, rewrite the highlighted functions using `guard` statements.

4.1.15 V-DMO-VUL-015: External call to register() can be safely replaced with internal call

Severity	Info	**Commit**	f0dc56d
Type	Gas Optimization	**Status**	Fixed
File(s)	`DaimoNameRegistry.sol`		
Location(s)	registerSelf()		
Confirmed Fix At	`https://github.com/daimo-eth/daimo/pull/268`		

The `DaimoNameRegistry` contract that has a method `registerSelf()` that effectively serves as an alias to the `register()` method but with the `address` argument set to `msg.sender`. Unexpectedly, the `registerSelf()` method is implemented as an *external* call to the `register()` method rather than as an internal call. As the `register()` method does not use any information from the calling context, the external call can be safely replaced with an internal call to reduce gas consumption.

```
function register(bytes32 name, address addr) public {
    require(_addrs[name] == address(0), "NameRegistry: name taken");
    require(_names[addr] == bytes32(0), "NameRegistry: addr taken");
    _addrs[name] = addr;
    _names[addr] = name;
    emit Registered(name, addr);
}

/// Registers msg.sender under a given name.
function registerSelf(bytes32 name) external {
    this.register(name, msg.sender);
}
```

Snippet 4.17: Definition of `register()` and `registerSelf()`

Recommendation Replace `this.register(...)` with `register(...)` in the definition of `registerSelf`.