Ethereum Foundation

BLS Wallet

Smart Contract Security Review

Version: 1.0

July, 2022

Contents

Introduction

Sigma Prime was commercially engaged by the Ethereum Foundation to perform a time-boxed security review of the BLS wallet smart contracts. The review focused solely on the security aspects of the Solidity implementation of the contract, though general recommendations and informational comments are also provided.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the smart contract. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the BLS Wallet smart contracts contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

Outputs of automated testing that were developed during this assessment are also included for reference (in the Appendix: Test Suite).

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the client smart contracts.

Overview

BLS Wallet is a Layer 2 native smart contract wallet which uses BLS signatures and aggregated transactions to reduce gas costs.

As well as implementing aggregated transactions, BLS Wallet enables gasless transactions via their aggregator service. This allows BLS wallet users to send their transactions to a third party who then is paid in non-ETH tokens to broadcast the transaction to the network.

The wallets are upgradeable as well, allowing the user to swap for an updated version of BLS Wallet or their own private version. Account recovery is supported by means of a multi-signature scheme with signers chosen by the user.

BLS Wallets also support multiple operations in the same transaction allowing users to perform operations such as approving and sending an ERC20 in one transaction.

The cryptographic primitives leveraged by BLS Wallet are based upon the prior work of the Hubble project and makes use of the precompiled contracts in Ethereum to reduce complexity and save on gas.

Security Assessment Summary

This review was conducted on the files hosted on the Web3Well `bls-wallet` repository and were assessed at commit ce7f958.

Note: the OpenZeppelin libraries and dependencies were excluded from the scope of this assessment.

The manual code review section of the report is focused on identifying any and all issues/vulnerabilities associated with the business logic implementation of the contracts. This includes their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout). Additionally, the manual review process focused on all known Solidity anti-patterns and attack vectors. These include, but are not limited to, the following vectors: re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers. For a more thorough, but non-exhaustive list of examined vectors, see [1, 2].

To support this review, the testing team used the following automated testing tools:

- Mythril: `https://github.com/ConsenSys/mythril`
- Slither: `https://github.com/trailofbits/slither`
- Surya: `https://github.com/ConsenSys/surya`

Output for these automated tools is available upon request.

Findings Summary

The testing team identified a total of 14 issues during this assessment. Categorized by their severity:

- Critical: 3 issues.
- Medium: 2 issues.
- Low: 4 issues.
- Informational: 5 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the BLS Wallet smart contracts. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the contracts, including gas optimisations, are also described in this section and are labelled as “informational”.

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected contract(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
BLS-01	BLS Rogue Key Attack Allows Executing Arbitrary Transactions	Critical	Open
BLS-02	Operations Are Vulnerable To Signature Replay Via Reentrancy	Critical	Open
BLS-03	Ownership of `ProxyAdmin` May Be Transferred To Any Wallet	Critical	Open
BLS-04	Signatures May Be Replayed Across Different Wallets	Medium	Open
BLS-05	Operation External Calls Can Use Excessive Gas	Medium	Open
BLS-06	Denial Of Service Attacks On Relayer	Low	Open
BLS-07	Insufficient BLS Domain Allows For Signature Replays Under Specific Conditions	Low	Open
BLS-08	Vulnerability In OpenZeppelin Library `4.3.2`	Low	Open
BLS-09	Proof of Possession Signatures Do Not Have Freshness Requirements	Low	Open
BLS-10	Silent Failure When User Submits Invalid Nonce	Informational	Open
BLS-11	Wallet Recovery Break Glass Fields & State Updates	Informational	Open
BLS-12	Custom Wallets May Avoid Paying Relayer Fees	Informational	Open
BLS-13	Potential Gas Saving for Signature Verification	Informational	Open
BLS-14	Miscellaneous `BLSWallet` General Comments	Informational	Open

BLS-01	BLS Rogue Key Attack Allows Executing Arbitrary Transactions		
Asset	`contracts/VerificationGateway.sol`		
Status	**Open**		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

By creating a new wallet while executing the function `processBundle()`, we can leverage the cryptographic attack known as the `rogue public key attack` [3] to commit any arbitrary action including theft of assets.

When processing a bundle of operations it is necessary to verify that each operation was approved by checking the owner's signature. This is done using the BLS signature scheme which enables a single aggregate signature to be checked. This aggregate signature is formed by merging all the signatures related to each operation together.

The aggregate signature is possible because BLS signatures use a type of cryptography that is pairing-friendly. What this means is given a mapping $e : G_1 \times G_2 \longrightarrow G_T$ then for all $P \in G_1$, all $Q \in G_2$ and all $a, b \in F_N$ we have that

$$e(aP, bQ) = e(P, Q)^{ab} \tag{1}$$

In short this enables us to aggregate signatures signing arbitrary messages into a single signature whilst still being able to check the authenticity of each signed message.

Then when calling `getOrCreateWallet()` inside `processBundle()`, it is possible to provide a fake public key that is the negative of a real user's key. If these public keys are then used to confirm identical wallet actions, it is possible to pass the point at infinity (a special type of zero) as the aggregate signature.

The reason this works is because when forming the aggregated public key from public keys pk_1 and pk_2 we have

$$pk_{agg} = pk_1 + pk_2 \tag{2}$$

Then when $pk_2 = -pk_1$ this simplifies to $pk_{agg} = 0$ where 0 is used to represent the point at infinity. We can then simply pass the point at infinity as the aggregated signature σ as

$$\sigma = pk_{agg} \cdot H(m) = 0 \cdot H(m) = 0 \tag{3}$$

where $H(m)$ is our hashed message.

The fake public key and victim's real public key will cancel out allowing this aggregate signature to verify both wallets action without having gained the victim's signature. Because each user's public key is broadcast on-chain when interacting with BLS Wallet and calculating its negative is trivial it is almost certain this issue would be exploited against real users.

Recommendations

Some possible solutions to this issue are:

- Require proof of possession when creating a new wallet, similar to `setSafeWallet()`. The message, unique to each public key, is signed by the private key. Verifying the signature ensures user knows the private key corresponding to the provided public key. Note, currently `setSafeWallet()` will only sign over the wallet address, it should also include the BLS public key in the signed message.

- Message augmentation may also be used to ensure messages from different accounts are unique by setting the public key as the first bytes of each message. This would mean that two messages from different public keys will have different hashes once processed and so will not aggregate together.

The trade-off here is that proof of possession requires an additional transaction to create a wallet before it may be used. Message augmentation, on the other hand, will increase the size of the message data and therefore increase gas costs of verifying each message.

BLS-02	Operations Are Vulnerable To Signature Replay Via Reentrancy		
Asset	`blsWallet/BLSWallet.sol`		
Status	**Open**		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

Operations are vulnerable to signature replay via reentrancy as the nonce is incremented after external calls are made.

To prevent an operation being repeated or operations run in the wrong order, a nonce is attached to each operation. A wallet's nonce is checked after the signature is verified, if the nonce is correct the operation is then executed. Once an operation has been run, successful or not, the nonce is incremented.

However, because the nonce is incremented after an operation has finished it is possible to replay a nonce by reentering `processBundle()` during an operation execution. This means the same signature can be used repeatedly without further authorisation from the wallet owner.

A valid exploitation scenario would consist in a malicious user writing a smart contract that contains a fallback function. The fallback function calls `VerificationGateway.processBundle()` with a preloaded `bundle`.

The malicious user must first store the `operation` with the replayable signature in their malicious contract. To achieve this, they listen to either the mempool or the off-chain bundle aggregator protocol to see the bundle containing a transaction which sends ETH to their malicious contract. Sending ETH to the malicious contract will trigger the fallback function which gives control of execution to the attacker.

Now that the attacker has a signed `operation` which will give them control of execution, they may store the `operation` in the malicious contract. Next the attacker will call `VerificationGateway.processBundle()` where the `bundle` is the single stored `operation`. This will execute the `operation`, transferring ETH to the malicious contract.

The malicious contract's fallback function will be called. Within the fallback function, they may load the `operation` from storage and again call `VerificationGateway.processBundle()`. The `operation` will be processed a second time as the nonce has not been incremented and is therefore still valid. Due to executing the `operation` a second time more funds are transferred to the malicious contract. The malicious contract may optionally repeat this process, transferring more ETH each time until there is minimal balance left in the victim's wallet.

This issue is rated as critical as any bundle with an action that gives control of execution to a third party may be replayed an infinite number of times. As a result any value transfers may be repeated until there is no value left, draining a wallet of its assets.

Recommendations

A possible mitigation of this issue would be to move the `incrementNonce()` call on line [**160**] to before `_performOperation()` on line [**151**]. As the nonce is checked prior to calling `performOperation()` this will not affect the current operation but will increment the nonce for any nested calls.

BLS-03	Ownership of `ProxyAdmin` May Be Transferred To Any Wallet		
Asset	`blsWallet/VerificationGateway.sol`		
Status	**Open**		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

The function `walletAdminCall()` is used by BLS wallets to call functions in `ProxyAdmin`. However, it may be hijacked to transfer ownership of `ProxyAdmin` to the wallet.

`walletAdminCall()` propagates calls from a `BLSWallet` to `ProxyAdmin`. The intention is to allow a wallet to call `ProxyAdmin.upgrade()` or `ProxyAdmin.changeProxyAdmin()`. Each of the functions listed in `ProxyAdmin.sol` have `TransparentUpgradeableProxy` as the first parameter.

`walletAdminCall()` checks that the first parameter of the call to `ProxyAdmin` is the `BLSWallet` (i.e. `msg.sender`). The check ensures that a wallet cannot change the admin or implementation logic of other wallets, only itself.

However, there are no checks on which `ProxyAdmin` functions are being called, hence a user may call `transferOwnership()` which `ProxyAdmin` inherits from `Ownable`. The first parameter of `transferOwnership()` is `address newOwner`. Since the first parameter of a `ProxyAdmin` call must be a `BLSWallet` the `newOwner` field will be the wallet.

As `VerificationGateway` owns `ProxyAdmin`, this call passes the `onlyOwner` modifier and ownership is transferred to the calling `BLSWallet` contract.

This is poses a critical risk as `ProxyAdmin` is the same for all wallets using the `VerificationGateway`. Once the ownership is hijacked by a malicious `BLSWallet` instance, it would be possible to call the admin functions of the `ProxyAdmin` with arbitrary data. These functions include the ability to change the implementation address of the `BLSWallet` that the proxy points to. Changing the implementation contract would allow the attacker to call arbitrary functions on the wallet and drain all assets.

Recommendations

To prevent this issue, there are two mitigations in `walletAdminCall()` which limit the functions that can be called in `ProxyAdmin`. The restrictions are to either blacklist the `Ownable` functions or whitelist the `ProxyAdmin` functions.

BLS-04	Signatures May Be Replayed Across Different Wallets		
Asset	`blsWallet/VerificationGateway.sol`		
Status	**Open**		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

Users can change the key pair that their wallet uses if they call `setBLSKeyForWallet()`. If a user swaps to a new wallet and then updates this new wallet to use their public key from the old wallet, then all operations signed by the old wallet are now valid for the new wallet.

While the nonces must still match for the new wallet's operations, this can allow a third party to grief or steal from the user by carrying out old operations on the user's new wallet without their authorisation.

The cause of this issue is that signatures do not sign over a specific wallet. Signatures will sign over the nonce, bundle and chain ID.

When a public key is added to a different wallet, the nonce may be reset. The other fields of the tuple (nonce, bundle, chain ID) will be unchanged and therefore we may replay the previous signatures.

Recommendations

This vulnerability could be fixed by including the wallet address in the signed data of each operation. This way the signature can only be used with the wallet it was intended for.

An alternative solution is to only allow a public key to be set once. This can be done by storing a mapping of the public key hash to a boolean value which is set to `true` the first time a public key is used. To prevent other users from claiming a public key without knowing the private key, this solution would require a Proof of Possession scheme to be implemented.

BLS-05	Operation External Calls Can Use Excessive Gas		
Asset	`blsWallet/VerificationGateway.sol`		
Status	**Open**		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

Once a wallet's operation is run in `VerificationGateway`, control is handed over to `BLSWallet` via the calls `wallet.performOperation()` and `wallet.nonce()`.

```
267 (
        bool success,
269     bytes[] memory resultSet
    ) = wallet.performOperation(bundle.operations[i]);
```

`wallet` is an untrusted address which may use an arbitrary amount of gas. No gas limit is set on either of the external calls and thus the default of `gasRemaining * 63 / 64` will be used. Any unused gas may continue to be used by the calling functions.

The external call may contain logic that maximises gas usage if the `tx.origin` is a third party relayer in addition to some modifiable conditions. If the modifiable conditions are changed between the estimate by the relayer and execution of `processBundle()`, this could lead to the relayer paying significantly more in gas costs than expected.

An attacker may exploit this causing the relayer to pay for gas intensive operations.

Recommendations

A method of solving this issue would be to add an extra field to each operation stating the amount of gas to attach to `wallet.performOperation()`. Furthermore, there should be a limit on the gas passed to `wallet.nonce()`. Consider allowing the relayer to pass this as a function parameter. The relayer would then know ahead of time what the maximum gas cost of each operation will be.

Note that there should also be an additional check to ensure there is sufficient gas remaining in the contract to pass the required gas to the wallet. This will add protection from a relayer who may deliberately send less gas than required to the wallet by manipulating the transaction `gasLimit`.

BLS-06	Denial Of Service Attacks On Relayer		
Asset	`blsWallet/VerificationGateway.sol`		
Status	**Open**		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

There are three possible attacks a wallet owner may use to cause a relayer's `processBundle()` transaction to revert.

Relayers gather a list of operations which may be collected from external parties and process them on-chain for a fee. It is possible to cause the relayer's `processBundle()` transaction to revert. Since a relayer will be paid by actions within a `processBundle()` operation the relayer will not receive funds if the transaction reverts.

It is therefore possible for wallet owners to perform griefing attacks on a relayer by causing their transaction to revert. The wallet attacker may perform some of these attacks after the relayer's transaction is already in the mempool by front-running it with a higher gas price.

The first possible method for getting a `processBundle()` transaction to revert is to cause `new TransparentUpgradeableProxy{salt: publicKeyHash}(...)` to revert on line [293]. Contract creation will revert if the address is already in use. The address will already be in use if the public key was used previously by a wallet but the wallet has changed their public key via `setSafeWallet()`. A wallet owner may change their public key at any time by calling `recoverWallet()`.

The second method for causing a process bundle operation to fail is for the node to have a custom wallet which reverts on `wallet.nonce()` or `wallet.performOperation()`. Wallet owners may upgrade the logic at any time via `walletAdminCall()`.

Finally, a wallet owner may use $\frac{63}{64}$ of the remaining gas causing the transaction to run out of gas when processing the remainder of the bundle. Since, the wallet makes external calls to arbitrary addresses, the amount of gas used by these external calls is dynamic.

Recommendations

The first issue may be resolved by using Proof of Possession and creating public keys prior to calling `processBundle()`. Therefore, `new TransparentUpgradeableProxy{salt: publicKeyHash}(...)` will not be called.

For the second issue, consider using a `try-catch` statement for calls to an untrusted `wallet` addresses within `processBundle()`, such as `wallet.nonce()` and `wallet.performOperation()`.

Finally, add a `gas` parameter to each operation which dictates how much gas must be sent to the wallet. This is similar to the recommendation in BLS-05. Additionally, have a fixed gas stipend for calling `wallet.nonce()`.

BLS-07	Insufficient BLS Domain Allows For Signature Replays Under Specific Conditions		
Asset	`blsWallet/VerificationGateway.sol`		
Status	**Open**		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The choice of `BLS_DOMAIN` is insufficient to provide full protection from replays across different contracts and chains. The `BLS_DOMAIN` is `keccak256(abi.encodePacked(uint32(0xfeedbee5)))`.

The first issue relates to Proof of Possession signatures in `safeSetWallet()`. The message to be signed for Proof of Possession is simply the wallet address. It is therefore possible to replay this signature across multiple `VerificationGateway` contracts and chains. That is because there is no information to distinguish different blockchains or other `VerificationGateway` contracts deployed on the same chain.

This issue partially exists in the function `verify()` which validates the signature in a bundle. Each message will have `chainId` appended to prevent replay across chains. However, it does not chain code which prevent signatures from being replayed on other `VerificationGateways`.

A third related issue is that the same domain is being used for Proof of Possession signatures as for bundle signatures. An overlap between a bundle message and a Proof of Possession message would allow a signature to be malleable in the message. This is not currently exploitable as the bundle messages and Proof of Possession messages will have different lengths and therefore a hash collision is unlikely due to the security guarantees of Keccak256.

Recommendations

We recommend updating the `BLS_DOMAIN`.

First, both the `chainId` and `address(this)` (the `VerificationGateway`) should be hashed into the `BLS_DOMAIN`. This will have the added gas saving that `chainId` does not need to be appended to each message.

Second, there should be a separate domain for Proof of Possession messages and bundle messages.

Other recommendations are to also include a `version` and a `name` fields to allow protocol upgrades. Consider using the EIP712 domain as inspiration and additional information.

BLS-08	Vulnerability In OpenZeppelin Library `4.3.2`		
Asset	`blsWallet/BLSWallet.sol`		
Status	**Open**		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `BLSWallet.sol` contract imports the OpenZeppelin library at version `4.3.2` to make use of the `initializer` modifier. Version 4.3.2 of this library contained a bug relating to the `initializer` modifier. The vulnerability in the `initializer` modifier is that it does not prevent reentrancy.

This issue is raised as low risk as the `initializer` modifier is only used once in `BLSWallet.initialize()`. Since the function does not make any external calls, reentrancy is not possible. However, as this is an upgradeable contract, future upgrades to the contract may change the `initialize()` function, potentially leading to vulnerable code.

Recommendations

It is recommended to update the OpenZeppelin library to the most recent version, at the time of writing that is `4.7.1`.

BLS-09	Proof of Possession Signatures Do Not Have Freshness Requirements		
Asset	`blsWallet/VerificationGateway.sol`		
Status	**Open**		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The signature in `safeSetWallet()` is replayable. That is, it is possible to re-use a signature for any length of time once a signature is generated.

The message signed in `safeSetWallet()` is the `wallet` address. Since the message does not have a freshness requirement it is possible to replay these signatures at any point in the future.

While these signatures may be replayable due to the lack of freshness, both `setBLSKeyForWallet()` and `setPendingBLSKeyForWallet()` require the `msg.sender` to be the wallet whose public key is being set. Thus, it is only possible to replay the key signature if the wallet calls one of these functions. The related impact and likelihood are therefore rated as low.

Recommendations

To mitigate this issue we recommend adding a freshness check. An example of a freshness check is to include a timestamp in the message that is signed, then check the timestamp is within a reasonable limit say 1 hour of the current block timestamp.

BLS-10	Silent Failure When User Submits Invalid Nonce
Asset	`blsWallet/VerificationGateway.sol`
Status	Open
Rating	Informational

Description

When a user submits an operation to be processed through the `VerificationGateway.sol`, they submit a nonce that prevents the same operation being replayed or run in the wrong order if they are submitting several operations. If this submitted nonce does not match the wallet's current expected nonce then the operation will not be run.

This failure to run an operation is not recorded or broadcasted to the user in a clear way and so a user may continue to resubmit the transaction unchanged or be unaware that the operation did not run.

Recommendations

Below are some possible methods of making nonce rejection clearer to the end user:

- Emit an event when an incorrect nonce means a transaction is not run. This can then be detected off-chain allowing the user to be notified.
- Return the failure to run the operation along with `successes` and `results` returned by function `processBundle()`.

BLS-11	Wallet Recovery Break Glass Fields & State Updates
Asset	`blsWallet/VerificationGateway.sol`
Status	Open
Rating	Informational

Description

The function `recoverWallet()` contains a number of secret fields such as `salt` and the sender address. Since all transactions on-chain are public and front-runnable, these fields may be known and used by attackers when submitting transactions on-chain. This issue is raised as informational as the function requires the `msg.sender` to be the recoverer address, therefore access control is sufficient to prevent exploitation.

After `recoverWallet()` has been called and a new public key set, the field `BLSWallet.recoveryHash` remains unchanged. It will no longer be possible to use this `recoveryHash` since it incorporates a hash of the public key, which is modified in `recoverWallet()`.

Recommendations

Ensure the development team are aware of the dangers of using break glass fields on a public blockchains.

Since the previous `recoveryHash` is invalid there are three options to handle this:

1. Zero `BLSWallet.recoveryHash` so it can be immediately reset in `setRecoveryHash()`;
2. Have a `newRecoveryHash` parameter in `recoverWallet()` and update `BLSWallet.recoveryHash`; OR
3. Leave the invalid `recoveryHash` and therefore enforce the delay in `pendingRecoveryHashTime` before a new one may be set.

BLS-12	Custom Wallets May Avoid Paying Relayer Fees
Asset	`blsWallet/VerificationGateway.sol`
Status	**Open**
Rating	Informational

Description

There is an issue known to the development team where custom wallets may avoid paying relayer fees.

Relayers submit transactions in the form of operations on-chain on behalf of a wallet. Relayers will receive funds from the wallet by executing actions included in an operation. The relayer will decide if the operation is profitable by performing a virtual execution of the operation and comparing total value held by the relayer before and after the transaction.

An operation will be executed by the wallet through the external call `wallet.performOperation()`. Since each individual action is processed within the wallets execution control, they may skip execution of certain actions, such as any actions that transfer fees to the relayer.

Wallet implementations may be updated via `recoverWallet()` or `walletAdminCall()`. It is possible for a malicious wallet to update the implementation contract after the relayer has signed the transaction before it is mined on-chain. This may be done by front-running the `processBundle()` transaction with a `recoverWallet()` transaction that has a higher gas price.

This issue is rated as informational as it is an issue known to the development team. It would also vary in profitability based on the gas fees an attacker would pay to front-run a `processBundle()` transaction and the fees paid to the relayer.

Recommendations

Consider allowing relayers to reject executing operations where the bytecode of the wallet is not the default setup. That is, if the wallet address has bytecode which is not equivalent to `TransparentUpgradeableProxy` and the implementation logic address of the `TransparentUpgradeableProxy` matches `VerificationGateway.blsWalletLogic`.

An alternate mitigation is to allow a `processBundle()` operation to perform balance checks for a list of native or ERC20 tokens. Requiring the relayer balance of one or many tokens to increase by a fixed amount will ensure the relayer has been paid for their services.

BLS-13	Potential Gas Saving for Signature Verification
Asset	`blsWallet/lib/hubble-contracts/contracts/lib/BLS.sol`
Status	**Open**
Rating	Informational

Description

When verifying public keys, signatures and messages are all related to the same key pair. BLS wallet uses a precompiled Ethereum contract at address `0x08`. This precompiled contract will fail calls where the BLS signature fails to validate. BLS Wallet uses the Hubble libraries, which are based on a prior work. The library determines the gas cost of these calls prior to calling. This enables them to tell the difference between a call failing due to insufficient gas and an invalid signature.

With BLS Wallet the out-of-gas failure is not an issue as the bundle would fail verification and stop running. This means that it is possible to remove the `precompileGasCost` on line [**117**] and replace its use with `gas()` on line [**121**]. The assembly operation `gas()` will return the remaining gas and therefore attach all available gas.

On testing this replacement code, it was found to be about 3,000 gas cheaper on calls using valid and invalid signatures. However, there is an edge case trade off in that public keys which are not on the curve or not in the subgroup G2 become significantly more expensive. For these cases, $\frac{63}{64}$ of the remaining gas will be burnt. The cause for the gas increase is that the precompiled contracts do the equivalent of a Solidity `assert()` meaning that all gas sent with the call is burnt on this type of failure.

As these calls are performed by the aggregator service, edge cases like these can be checked for off-chain prior to submission and can be avoided.

Recommendations

Consider weighing the potential gas savings against the extra costs for the listed edge-cases.

BLS-14	Miscellaneous `BLSWallet` General Comments
Asset	`contracts/*`
Status	Open
Rating	Informational

Description

This section details miscellaneous findings discovered by the testing team that do not have direct security implications:

1. **BLSwallet.sol**
   (a) Replace the magic number 604800 with `1 weeks` for readability on line [**72**], line [**82**] and line [**91**].
   (b) Several functions are marked as `public` but could be restricted to `external`. These include `setRecoveryHash()`, `setTrustedGateway()`, `recover()`, `setProxyAdminFunctionHash()`, `setAnyPending()`, `performOperation()`, `_performOperation()` and `clearApprovedProxyAdminFunctionHash()`.
   (c) `a.contractAddress` variable in line [**181**] and line [**184**] can also be an `EOA`. Consider renaming this variable to `a.targetAddress`.
2. **VerificationGateway.sol**
   (a) No zero address checks on [**68-69**] for constructor arguments.
   (b) The variable `selectorOffset` on line [**162**] could be set as a global constant to reduce gas usage.
   (c) The condition `(blsGateway != address(0))` in the require statement in line [**223**] is unnecessary and can be safely removed because `address(0)` has zero code size.
   (d) Typo: `wallletAddressSignature` should be `walletAddressSignature` on line [**314**].
   (e) `processBundle()` doesn't check for the zero public key.
3. **Create2Deployer.sol**
   (a) Functions `addressFrom()` and `deploy()` can be changed to being external visibility to save gas when calling them.
4. **lib/BLS.sol**
   (a) The function `isZeroBLSKey()` can be changed to being external visibility.

Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing the suggestions above.

Appendix A Test Suite

A non-exhaustive list of tests were constructed to aid this security review and are provided alongside this document. The `brownie` framework was used to perform these tests and the output is given below.

```
test_variable_cost                           PASSED  [2%]
test_deploy                                  PASSED  [5%]
test_mapToPoint                              PASSED  [8%]
test_isOnCurveG1                             PASSED  [11%]
test_isOnCurveG2                             PASSED  [14%]
test_isValidSignature                        PASSED  [17%]
test_sqrt                                    PASSED  [20%]
test_inverse                                 PASSED  [22%]
test_hashToField                             PASSED  [25%]
test_expandMsgTo96                           PASSED  [28%]
test_ModexpInverse                           PASSED  [31%]
test_ModexpSqrt                              PASSED  [34%]
test_bn254                                   PASSED  [37%]
test_setup_protocol                          PASSED  [40%]
test_wallet_created                          PASSED  [42%]
test_recover_wallet                          PASSED  [45%]
test_set_new_bls_key                         PASSED  [48%]
test_set_bls_key_for_wallet_zero_public_key  PASSED  [51%]
test_send_eth_to_eoa                         PASSED  [54%]
test_process_bundle_action_fails             PASSED  [57%]
test_mint_approve_transfer_from              PASSED  [60%]
test_wallet_admin_call                       PASSED  [62%]
test_wallet_admin_call_wrong_wallet          PASSED  [65%]
test_wallet_admin_call_steal_ownership       XFAIL   [68%]
test_set_trusted_bls_gateway                 PASSED  [71%]
test_process_bundle_wrong_nonce              PASSED  [74%]
test_process_bundle_wrong_signature          PASSED  [77%]
test_process_bundle_mismatch_length          PASSED  [80%]
test_process_bundle_zero_publicKey           XFAIL   [82%]
test_verify                                  PASSED  [85%]
test_verify_multiple                         PASSED  [88%]
test_repeat_nonce                            XFAIL   [91%]
test_rogue_key_attack                        XFAIL   [94%]
test_set_new_wallet_PoP_replay               XFAIL   [97%]
test_new_wallet_signature_replay             XFAIL   [100%]
```

Appendix B Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

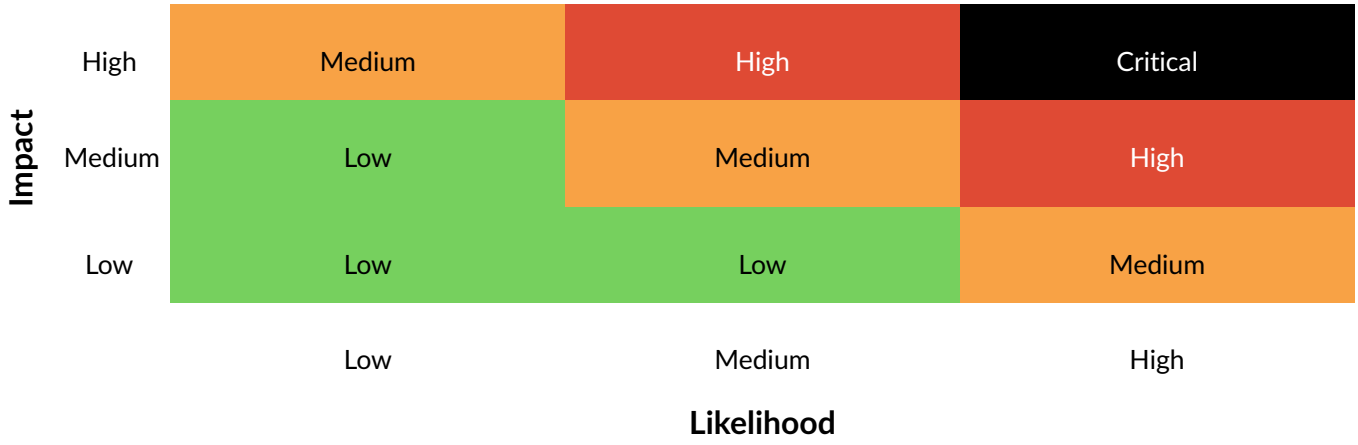

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References

[1] Sigma Prime. Solidity Security. Blog, 2018, Available: https://blog.sigmaprime.io/solidity-security.html. [Accessed 2018].

[2] NCC Group. DASP - Top 10. Website, 2018, Available: http://www.dasp.co/. [Accessed 2018].

[3] Dan Boneh C. Manu Drijvers C. Gregory Neven. Compact multi-signatures for smaller blockchains. 2018. Available: https://eprint.iacr.org/2018/483.pdf.