OpenZeppelin | security

Ironblocks Onchain Firewall Audit

March 26, 2024

Table of Contents

Table of Contents 2

Summary 4

Scope 5

System Overview 6

Schemetic Execution Flow 7

Connecting to Firewall 7

Configuring Policies 8

Security Model and Trust Assumptions 8

Privileged Roles 9

Threat Model 11

High Severity 15

H-01 Risk of Signature Replay Attack in ApprovedCallsPolicy 15

H-02 Risk of Passing Incorrect Context to Firewall Policies 16

Medium Severity 16

M-01 Risk of Non-Reachable postExecution 16

M-02 Confusion With Multiple Modifiers Passing Custom Data to Policies 17

M-03 safeFunctionCall May Expose Unnecessary Future Risks 19

M-04 OnlyEOAPolicy Allows Non-Consumer-Specific Contract Accounts 20

M-05 Nested Protected Functions May Cause Unexpected Reverts 20

M-06 Challenges for Admin Functionality and Logic Initialization in FirewallProxyIntercept 21

Low Severity 23

L-01 Lack of Event Emission for Firewall Setting Changes 23

L-02 Simulation Mode in ApprovedCallsPolicy 23

L-03 try/catch Error(string memory){} in CombinedPoliciesPolicy May Not Catch Call Cases 23

L-04 ForbiddenMethodsPolicy May Block More Transactions Than Intended 24

L-05 Risk of Function Signature Clash With ifAdmin 25

L-06 Policies Can Be Executed When No Longer Approved 25

L-07 Implementation Address of the FirewallProxyIntercept May Not Be Compatible With Etherscan 26

L-08 staticCallCheck Can Be Circumvented 26

L-09 Array Lengths May Mismatch 27

L-10 Limitations of AdminCallPolicy 27

L-11 Missing Docstrings 28

L-12 Use of tx.origin to Validate EOA Sender 28

Notes & Additional Information 29
N-01 Unused Imports 29
N-02 Unnecessary Casts 29
N-03 Lack of Indexed Event Parameter 29
N-04 Non-Explicit Imports Are Used 30
N-05 Using uint Instead of uint256 30
N-06 Missing Named Parameters in Mappings 30
Conclusion 32

Summary

Type	DeFi
Timeline	From 2024-02-05 To 2024-02-27
Languages	Solidity

Total Issues	26 (26 resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	2 (2 resolved)
Medium Severity Issues	6 (6 resolved)
Low Severity Issues	12 (12 resolved)
Notes & Additional Information	6 (6 resolved)

Scope

We audited the ironblocks/onchain-firewall repository at commit 8f4fcf861d53c835ed005db67b8af4037464e0ca.

In scope were the following files:

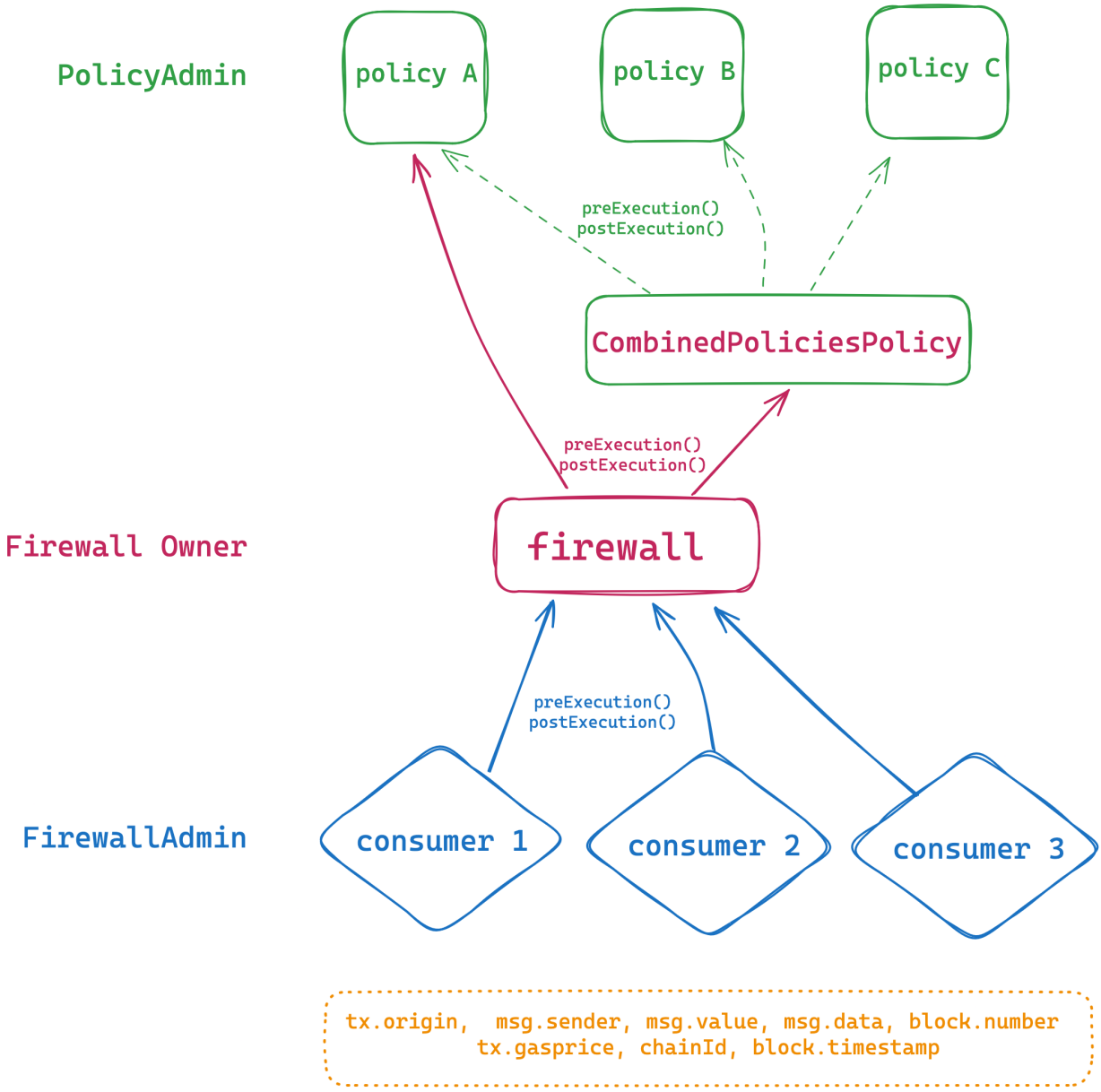
```
contracts
├── interfaces
│   ├── IERC165.sol
│   ├── IFirewall.sol
│   ├── IFirewallConsumer.sol
│   ├── IFirewallPolicy.sol
│   └── IFirewallPrivateInvariantsPolicy.sol
├── policies
│   ├── AdminCallPolicy.sol
│   ├── AllowlistPolicy.sol
│   ├── ApprovedCallsPolicy.sol
│   ├── ApprovedVectorsPolicy.sol
│   ├── BalanceChangePolicy.sol
│   ├── BlocklistPolicy.sol
│   ├── CombinedPoliciesPolicy.sol
│   ├── FirewallPolicyBase.sol
│   ├── ForbiddenMethodsPolicy.sol
│   ├── NonReentrantPolicy.sol
│   └── OnlyEOAPolicy.sol
├── proxy
│   ├── FirewallProxyAdmin.sol
│   ├── FirewallProxyIntercept.sol
│   └── FirewallTransparentUpgradeableProxy.sol
├── Firewall.sol
├── FirewallConsumer.sol
└── FirewallConsumerBase.sol
```

System Overview

The Ironblocks Onchain Firewall is a system of smart contracts designed to enable users to emulate a firewall, thereby protecting their smart contracts through the adoption of template security policies. These policies, to which users can subscribe, are customizable and operated by intercepting the calldata of incoming transactions to a smart contract's function protected by the firewall, both before and after execution.

This mechanism allows users to execute code and impose conditions with the context of the call at both stages, ensuring compliance with certain preset configurations when a protected function is executed. This is achieved through the use of two hooks: `preExecution`, which is run prior to the execution of the call, and `postExecution`, which runs after the call concludes to ensure that any updated states leave the protected contract in a secure state.

Schemetic Execution Flow



Connecting to Firewall

The system uses `Firewall.sol` to allow a consumer contract to subscribe to firewall-approved security policies. The three ways a consumer integrates with the firewall are:

1. Inheriting `FirewallConsumerBase.sol` and adding modifiers such as `firewallProtected` to selected functions.

2. Using `FirewallTransparentUpgradeableProxy.sol` jointly with `FirewallProxyAdmin.sol` to configure and add protection to all non-static function calls that are made to the implementation contract.
3. Making `FirewallProxyIntercept.sol` sit in-between a transparent proxy and an implementation, where the `FirewallProxyIntercept` routes every user call through the firewall.

When the connection to the firewall is set up, security policies can be added or removed over time when deemed fit directly through the `Firewall` smart contract, without having to make changes to the underlying business logic.

Configuring Policies

Each template policy implements a Role-Based Access Control (RBAC) system that grants the `DEFAULT_ADMIN_ROLE` to the deployer, who then grants/revokes the `POLICY_ADMIN_ROLE`. Each `POLICY_ADMIN_ROLE` can authorize consumer and executor status, for example allowing only the firewall to call its state-changing functions or allowing only the rightful consumer to be evaluated. Each policy has its own specific protection logic and thus may require policy-specific configuration. Some policies may require assigning new roles for on-going maintenance. See the next section for a detailed list of policies and their privileged roles.

Security Model and Trust Assumptions

For now, the firewall contracts, including proxies and policies, are meant to be utilized and maintained by projects that wish to build on top of this system. As such, it is the users' responsibility to assign, coordinate, and be secure in all privilege roles involved in the system.

Privileged Roles

Firewall

The deployer of `Firewall.sol` is its owner who can transfer or renounce ownership. Only the owner can `setPolicyStatus`. When a policy has its status as `true`, consumers can subscribe to it. When a policy has status `false`, it cannot be subscribed to by anyone.

Consumers

A `firewallAdmin` needs to be present in all consumers. Depending on how the consumer is set up, the way to change the `firewallAdmin` or the `firewall` address is different. The `firewallAdmin` can subscribe and remove policies, and also set the dry-run status when configuring the policies on the `Firewall`, without needing to connect to it directly.

If inheriting from `FirewallConsumerBase.sol`, the `firewallAdmin` can set the `FirewallAddress` for the consumer. On the connected `Firewall`, the `firewallAdmin` can additionally approve targets on which arbitrary calls can be made from the consumer contract. The incumbent `firewallAdmin` passes the role to their successor in a two-step process.

Using `FirewallProxyAdmin.sol` as the admin of the `FirewallTransparentUpgradeableProxy`, the owner of `FirewallProxyAdmin` can set the connected `Firewall` address as well as the `firewallAdmin` address on the proxy consumer.

Policies

Each instance of a policy is administered by its `POLICY_ADMIN_ROLE`. If a policy's execution hooks are protected by the `isAuthorized` modifier, the consumer and its connected `Firewall` need additional approvals from the `POLICY_ADMIN_ROLE` to successfully run that policy. The `POLICY_ADMIN_ROLE` can approve any address to be an authorized executor or consumer, without them having to be joined via the firewall.

Below is a list of additional configuration required from privileged roles in each policy:

- **AdminCallPolicy.sol** : An `APPROVER_ROLE` is granted to approve calls that can be executed only once within a preset expiration time period. The `APPROVER_ROLE` can update the expiration time period independently.

- **BlocklistPolicy.sol** : The `POLICY_ADMIN_ROLE` can set the statuses of a list of blocked addresses for each consumer. This policy can be used by any consumer without authorization.
- **OnlyEOAPolicy.sol** : This policy checks if the sender is the same as `tx.origin` or is from a list of `allowedContracts`. The `POLICY_ADMIN_ROLE` can set the list of allowed contracts. This policy can be used by any consumer without authorization.
- **AllowlistPolicy.sol** : The `POLICY_ADMIN_ROLE` can set the statuses of a list of allowed addresses for each consumer. This policy can be used by any consumer without authorization.
- **ApprovedVectorsPolicy.sol** : This policy allows the `POLICY_ADMIN_ROLE` to approve a sequence of calls, also known as "vectors" or "patterns", that are allowed in **all** authorized consumers. The `POLICY_ADMIN_ROLE` can remove any approved vectors. Note that any approved sequence would necessitate that at least the subsequence is also allowed. For instance, if [A, B, C, D] is allowed, then [A], [A, B], and [A, B, C] must also be allowed.
- **BalanceChangePolicy.sol** : The `POLICY_ADMIN_ROLE` can add tokens to be monitored and set the maximum balance change allowed for each token per consumer. In addition, this role can remove any token from the monitoring list and reset the maximum balance change allowance.
- **ForbiddenMethodsPolicy.sol** : The `POLICY_ADMIN_ROLE` can set a list of forbidden methods for each consumer. This policy can be used by any consumer without authorization.
- **ApprovedCallsPolicy.sol** : This policy allows the execution of a call if all aspects of the call context have been pre-approved. The `SIGNER_ROLE` is granted to sign ordered sequences of approved calls off-chain, each with a designated expiration time. These signed ordered sequences can then be verified only once in the right order on this contract before execution. The `SIGNER_ROLE` can also approve calls on-chain for each `tx.origin`.
- **NonReentrantPolicy.sol** : There is no additional privilege for the `POLICY_ADMIN_ROLE` in this policy.
- **CombinedPoliciesPolicy.sol** : This policy allows consumers to combine policies and override the result of a policy execution outcome. The `POLICY_ADMIN_ROLE` can set the policies to be combined and its allowed combinations of successes and failures.

Threat Model

This section describes the ten most relevant threats to the Firewall system identified during the audits. Some findings may be briefly mentioned as examples.

1. Threat of Misrepresented Call Context

Some call context (e.g., instance `tx.origin`, `msg.sender`, `msg.data` and `block.number`) that has been passed up from the consumer to the policies for evaluation may not represent the correct entities. Here are a few examples where this threat is relevant:

- `msg.sender` and `msg.data` can be overwritten by `_msgSender()` and `_msgData()`.
- In the context of meta-transactions, such as those following `ERC-2771` or `ERC-4337` standard, `tx.origin` can represent a third-party entity, a trusted forwarder, or a bundler. This may not be compatible with policies assuming `tx.origin` to be an EOA sender or a privileged account. Similarly, `msg.sender` may not represent the sender identity.
- In the context of protected `internal` functions, `msg.data` may not represent the actual data needed to evaluate the internal call. This is anticipated by the Ironblocks team with additional protection modifiers, such as `firewallProtectedCustom` and `firewallProtectedSig`, which rely on the users' discretion to pass the right data for evaluation.
- In the context of layer 2 networks, some context, such as `block.number` or `tx.origin` for message passing transactions, may behave differently from Ethereum.

2. Threat of Signature Replay and Malleability

Policies that verify signatures may be subject to this threat, particularly considering the possibility of multiple instances of a policy having a centralized signer. This is observed in the `ApprovedCallsPolicy`, where the verifying contract is not checked. In conjunction with this policy, the `approveCallsViaSignature` can be front-run thus able to DoS the `safeFunctionCall` with the documented initial use case. Furthermore, the size of the signature component `s` is not checked explicitly, subjecting the policy to a potential signature malleability threat.

3. Threat of Multiple Executions

Multiple executions of the same policy may result in unexpected reverts. This threat is present due to the following setup:

- The same policy can be accidentally added by the consumer `firewallAdmin` to the connected Firewall's `subscribedGlobalPolicies` as well as `subscribedPolicies` by signature. Since the global policies will run for all protected functions, the overlap in these may result in multiple executions.
- When a `firewallProtected` function calls another `firewallProtected` function in a nested fashion from the same contract, the same policies will also run multiple times in a nested fashion. This can cause unexpected reverts in policies such as `NonReentrantPolicy`, `BalanceChangePolicy`, or `ApprovedCallsPolicy` if the configuration is not set up appropriately.
- The `CombinedPoliciesPolicy` allows combining a policy with itself multiple times, thereby potentially causing multiple executions.

Since policies called on each protected function are not directly viewable, consumers need to make extra effort to avoid unnecessary policy runs.

4. Threat of Inconsistency

- Although all policies can approve multiple consumers to subscribe, some policies do not support consumer-specific states. For instance, `ApprovedVectorsPolicy` has no mapping specific to a consumer address. `OnlyEOAPolicy` does not allow consumer specific `allowedContracts`. `CombinedPoliciesPolicy` does not allow consumer specific `allowedCombinations`.
- On some policies, the `isAuthorized` modifer is not used. Although, it is understandable that since some execution hooks are `view` functions, it seems unnecessary to use `isAuthorized`. However, this may mean that anyone, without having authorization or being part of the firewall system, can tap into the policies such as `OnlyEOAPolicy` or `BlocklistPolicy` for free.

Inconsistencies in policy consumer support pattern and authorization requirement may become a potential threat for future integration.

5. Threat of Cross-Authorization

- The use of a centralized signer for multiple protocols implies risks (e.g., the risk of signature replay).
- Authorizing several executors, may or may not be the firewall, to interact with a policy.

- Authorizing targets that can be called from `safeFunctionCall` using the consumer's permissions.
- Any contracts implementing the `firewallAdmin() external returns(address)` interface can modify the states on `Firewall.sol`, such as `subscribedPolicies`, without having to connect to that firewall. Although the execution hooks will not work for such contracts without actually connecting to the firewall, it could suggest that the firewall may carry bloated states.

6. Threat of Special Execution Flow

Some firewall policies may not work as expected when protecting functions with the following execution flow:

- Use of assembly returns will cause the post-execution hook to be non-reachable when used with `FirewallConsumerBase`.
- Variation in loop length requires specific approval for each length when integrating with `ApprovedCallsPolicy`.
- Multiple calls of the same data in one transaction will not work with `AdminCallsPolicy`.

7. Threat of Excessive Restriction

The policies are considered security add-ons to the underlying consumer. However depending on the policy design, it may also be overly restrictive and may impede the day-to-day flow of the business logic. For instance, the `ApprovedCallsPolicy` can put very strict restriction on the underlying contract as only the last `callCash` in the array with `tx.origin` can be executed. There is only one `SIGNER` who needs to sign hashes for all consumers, all `tx.origin`, and all calls, which may become a bottleneck in the operation flow.

8. Threat of Selector Clash

- The `ApprovedVectorsPolicy` uses the function selector to generate a hash with the previous sequence to validate the call vectors. Since this policy is not specific to each consumer, it is possible to have clashes with function selectors from all subscribed consumers. Note that once a selector is approved in a vector, any function with the same selector will be approved too.
- There is a risk of function selector clashing between transparent proxies and implementation contracts.

9. Threat of Mis-integration

Below are some potential risks when integrating with the firewall:

- Risk associated with approving target contracts in the `FirewallConsumerBase`.
- Risk of incorrect context due to the wrong choice of modifier (e.g., within internal call contexts).
- Risk when updating the proxy setup: for instance, some challenges when calling admin functions in `FirewallProxyIntercept` and possible overriding of the `FirewallProxyIntercept` slot in the proxy when updating the logic.
- Risk of uncertainty regarding which policies a consumer is actively subscribed to as modifiers do not clearly indicate the policies to which the consumer has subscribed.
- Risk of overwriting approved calls in the `ApprovedCallsPolicy` when executions and approvals go out of sync as the `approveCallsViaSignature` can overwrite non-executed `approvedCalls`.

We recommend projects integrating with the firewall system to have a separate audit before deployment.

10. Threat of Malicious Consumer

Since the policies called on each protected function are not easy to keep track of from the consumers' contracts, it is potentially possible for malicious consumers to trick users via the facade of firewall protection. Consider the example where a contract subscribes to a slippage protection policy. In this case, the user might assume it is safe to interact with the contract. However, a malicious admin could front-run a user's transaction and alter the contract's logic by unsubscribing from the slippage protection, leaving the user's transaction without protection. This could break the guarantees that users supposedly think they have.

High Severity

H-01 Risk of Signature Replay Attack in `ApprovedCallsPolicy`

The `ApprovedCallsPolicy` mandates that a transaction to a consumer must be approved on-chain prior to execution. Only users holding the `SIGNER_ROLE` are authorized to approve transactions, and there are two methods to achieve this approval. The first method is for a user with the `SIGNER_ROLE` to approve the call directly on-chain by invoking `approveCalls()`. The second method involves obtaining approval for the calls with an off-chain signature, then invoking `approveCallsViaSignature()` and supplying the signature. To verify the signature's validity, the contract generates a `messageHash` comprising of:

- `bytes32 [] callHashes`, the hashes of a sequence of calls
- `uint256 expiration`, the expiration date of the calls
- `address txOrigin`, the account that initiated the calls
- `uint nonce`, the nonce associated with `txOrigin`
- `block.chainid`, used to avoid replay attacks

The nonce of `txOrigin` is incremented each time a signature is utilized, preventing the reuse of the signature within the same contract. However, the issue emerges when multiple `ApprovedCallsPolicy` instances from the same protocol share the same `SIGNER_ROLE`, enabling a user to reuse the same signature across each different policy instance. Although the nonce should be unique, it is conceivable that overlaps occur since all nonces start at 0. The Ironblocks team also noted that in the protocol's initial phase, the `SIGNER_ROLE` could be centralized and held by them.

A replay attack could also occur within a policy different from `ApprovedCallsPolicy` that shares the same `messageHash` structure in the future. In addition, in this setup, the value `s` in the signature is not checked. Although this oversight does not subject this specific policy to signature malleability attacks, it is generally advised as a good practice to verify this component of the signature to mitigate risks in future versions.

To prevent a signature from being reused across different contracts, consider including the verifying address to be part of the `messageHash` (e.g., following the style of EIP-712

standard). Regarding signature malleability, consider always checking the value of `s`, see for example the OpenZeppelin ECDSA implementation.

Update: *Resolved in pull request #6 at commit 90cd9b0.*

H-02 Risk of Passing Incorrect Context to Firewall Policies

The on-chain firewall system is designed to support meta-transactions. The FirewallConsumerBase contract, inheriting from `Context.sol`, allows consumers to override `_msgSender()` and `_msgData()` within the current execution context. This base contract, along with its modifiers, is responsible for managing the information passed to the `Firewall` contract.

However, an issue arises because the modifiers, particularly `firewallProtected`, utilize the raw `msg.sender` and `msg.data` instead of the potentially overridden `_msgSender()` or `_msgData()` for policy evaluation. If the context is overridden, for example, to support meta-transactions, incorrect values could be transmitted through the firewall, leading to false alarms or the passage of invalid transactions.

Further note that `tx.origin` is used extensively in policies, such as in AdminCallPolicy and ApprovedCallsPolicy, which may not be compatible with consumers that utilize meta-transactions.

Consider ensuring that modifiers use the correct context by utilizing `_msgSender()` and `_msgData()`. Moreover, if some policies are not designed to support meta-transactions, it is crucial to explicitly state this in the documentation, leaving no room for ambiguity.

Update: *Resolved in pull request #13 at commit 74a5635.*

Medium Severity

M-01 Risk of Non-Reachable `postExecution`

When inheriting the `FirewallConsumerBase`, any protected function using modifiers `firewallProtected`, `firewallProtectedCustom`, `firewallProtectedSig`, and `invariantProtected` will not reach the `postExecution` hook if the execution flow

reaches an inline assembly `return` in the function body. However when using `FirewallTransparentUpgradeableProxy`, the `firewallProtected _internalDelegate` can execute the `postExecution` after an early inline assembly `return`.

The following policies in scope are affected by this issue when used with the `FirewallConsumerBase`.

- In the `NonReentrantPolicy`, the `hasEnteredConsumer` state will not be reset back to `false`, thereby preventing any further legitimate calls from the same consumer, effectively disabling this policy for that consumer.
- In the `ForbiddenMethodsPolicy`, the revert upon entering a forbidden method resides in the `postExecution` hook. Thus, the call into an forbidden method will not be rejected without entering the `postExecution`.
- In the `BalanceChangePolicy`, the `postExecution` reads the consumerLastBalance to compute the change of balance in monitored tokens, reverting if the difference exceeds the maximum allowed. If the `postExecution` is not reached, the policy will not revert on calls exceeding a maximum balance change.
- In the `CombinedPoliciesPolicy`, the `postExecution` updates the `currenResult` from executing all subscribed policies, thus the execution results may not be accurate.

Consider heavily documenting this risk for firewall consumers to avoid unexpected policy behaviors.

Update: *Resolved in pull request #16 at commit eee15f4. The Ironblocks team stated:*

> *Added inline documentation to each modifier in order to alert developers to this risk.*

M-02 Confusion With Multiple Modifiers Passing Custom Data to Policies

Risk of Passing Arbitrary Data

In `FirewallConsumerBase`, the `firewallProtectedCustom` modifier and `firewallProtectedSig` can pass arbitrary user input byte arguments, which may not be `msg.data`, to policies for evaluation. This behavior is inconsistent with the documentation of

`Firewall` which states that policy contracts have full context of the call. In the case of misuse, this can cause the wrong data to be passed to a policy for evaluation.

Confusion In Approving Internal Calls

In spite of the apparent flexibility, an important use case for `firewallProtectedSig` is to protect internal calls as demonstrated in the `SampleConsumerInternals` contract where the argument for `firewallProtectedSig` is the 4-byte signature of the protected `internal` function. This is a fine strategy as otherwise the same `msg.data` from the parent external call would be used for policy evaluation for the internal call. However, the extended flexibility may require special considerations from policy privileged roles to operate correctly when used in conjunction with the `ApprovedCallsPolicy` and the `ApprovedVectorsPolicy`. For the sake of brevity, we only consider `firewallProtectedSig` in the following. Similar arguments apply to `firewallProtectedCustom` as well.

- When one uses `firewallProtected` on an `internal` function, the `msg.data` from the evoked external call is passed to the policies as the call context. When a consumer subscribes to the `ApprovedCallsPolicy`, it is necessary for the `SIGNER_ROLE` to approve the internal call using the same `msg.data` and `msg.value` for the callHash, thereby approving the exact same callHash twice, once for the external call and once for the evoked internal call.

- When the consumer chooses instead to protect an internal call using the `firewallProtectedSig` with the argument as the protected function signature, then the data used in `approvedCallsPolicy.preExecution` will be just the 4-byte function signature without any argument encoded. Thus, the `SIGNER_ROLE` needs to sign a `callHash` that consists of only the `funcSig` instead of encoding arguments like usual external calls. As such, any arguments will be valid for the internal call.

Both of these cases require the privileged roles from `policies` to be aware of the implementations and use different models for approving data depending on the visibility of the function and also depending on the protected modifier. This can be confusing for the SIGNER or `POLICY_ADMIN_ROLE`, particularly, if they are not maintained by the same team. This risk also applies to the `ApprovedVectorsPolicy`.

Consider heavily documenting the correct usage of each modifier in relation to relevant policies, for both `internal` or `external` functions. In addition, explicitly spell out the potential risks of certain configurations to discourage misuse for both consumer integration as well as policy administration.

Update: *Resolved in pull request #17 at commit 5ba418e. The Ironblocks team stated:*

> *Added comments to the two modifiers.*

M-03 `safeFunctionCall` May Expose Unnecessary Future Risks

The `safeFunctionCall` allows anyone (except via meta-transaction) to execute any data on an approved target, set by the firewallAdmin, with `this` consumer contract as `msg.sender`.

As documented, `safeFunctionCall` is written with a particular use case in mind: to allow simultaneous transaction approval via `approveCallsViaSignature` on the target `ApprovedCallsPolicy` and execution of the approved transaction.

This function may present the following risks.

- There is a front-run risk with the above use case where the payload can be intercepted and the signature can be used to call `approveCallsViaSignature` directly, thereby causing denial of service (DoS) for a legitimate caller of the `safeFunctionCall` function.
- The consumer may not subscribe to the `ApprovedCallsPolicy`, thereby rendering the initial use case unnecessary for some consumers who are nonetheless exposed to future risks.
- The approved `target` need not be a firewall policy. By allowing the `firewallAdmin` to approve arbitrary `target`, the privilege of the `firewallAdmin` role extends beyond firewall-related activities.
- Since the caller to the approved `target` is `address(this)`, any special privilege granted to this consumer contract can be accessed by anyone using this function. For instance, if the consumer contract has been granted some ERC-20 token approval by the `target`, anyone could call `transferFrom` on the `target` to get the approved amount.
- Depending on the approved `target`, this function may potentially be utilized as a convenient wrapper for some underlying business logic in the future. It is a good practice to adhere to separation of concerns, keeping the `FirewallConsumerBase` only for firewall-related logic and minimizing the possibilities to interfere in the underlying business logic.

Consider imposing further restrictions on the approved targets, limiting the privilege of the `firewallAdmin`, and thoroughly documenting the risks involved.

Update: *Resolved in pull request #15 at commit 0539c1f. The Ironblocks team stated:*

> *This pull request solves the issue with ERC-165, requiring any payload target to conform to a new interface. This means that no pre-existing contracts (tokens, NFTs, vaults, etc.) should be callable via `safeFunctionCall`. Regarding the DDoS, in the worst-case scenario, users' transactions can be delayed by one block, at a cost to the attacker and no cost to the user. This is because afterwards, the users' transaction will be approved and they can interact with the contract without using `safeFunctionCall` since the attacker already approved it. If this is not satisfactory, we can replace a revert either in the policy or the consumer to fully resolve the issue.*

M-04 `OnlyEOAPolicy` Allows Non-Consumer-Specific Contract Accounts

The `OnlyEOAPolicy` not only allows EOAs to interact with a protected function on the consumer, any contract account can also be added to the `allowedContracts` variable. The `allowedContracts` do not differentiate between consumers. As a result, a contract allowed by one consumer is considered a valid caller for all consumers subscribed to this policy.

This causes confusion as users may suppose that `OnlyEOAPolicy` only allows an EOA caller and reverts on non-EOA callers. To achieve the effect of allowing exceptions to the onlyEOA rule, one can combine this policy with `AllowlistPolicy` using `CombinedPoliciesPolicy`. It is a better strategy as the `AllowlistPolicy` can have different allowed accounts for different consumers.

Consider removing the `allowedContracts` state and keep it as a single-purpose policy or renaming it to reflect its intended purpose. Consider also enhancing the documentation on how to combine different policies for the desired effect.

Update: *Resolved in pull request #5 at commit 948288f.*

M-05 Nested Protected Functions May Cause Unexpected Reverts

- A firewall-protected function calling another firewall-protected function in the same contract will cause a revert in the NonReentrantPolicy. This is because the

`hasEnteredConsumer` is specific to the consumer address and will be set to `true` before it enters the first protected function and reverts when it reaches the inner protected function.

- In the `BalanceChangePolicy`, the `preBalance` is read at the beginning of each function call. It is possible that the intermediate change of balance in the inner call may exceed the maximum allowance but the final balance change will not.
- In the `ApprovedCallsPolicy` as well as the `ApprovedVectorsPolicy`, any protected inner function calls are required to be approved in advance in a specified manner. This also includes approval for each loop length and any further nesting.

Although the `CombinedPoliciesPolicy` permits combining the same policy more than once and thus can override certain combination of results from the same policy, it may not be the desired behavior. A similar behavior is observed in the `Firewall` contract that a consumer could be subscribed to a policy globally and also on a function level, which could create conflicts between the repeated policies.

Consider documenting the recommended and discouraged patterns for the relevant policies.

Update: *Resolved in pull request #18 at commit d7a6355. The Ironblocks team stated:*

> *Updated comments for the relevant policies.*

M-06 Challenges for Admin Functionality and Logic Initialization in `FirewallProxyIntercept`

The on-chain firewall contracts provide a set of proxy contracts designed to integrate firewall capabilities into the proxy layer. The `FirewallTransparentUpgradeableProxy` contract inherits from OpenZeppelin's `TransparentUpgradeableProxy` contract with additional firewall functionality. If a protocol opts to implement the firewall at the proxy layer instead of utilizing OpenZeppelin's `TransparentUpgradeableProxy`, they could use `FirewallTransparentUpgradeableProxy`. Another proxy contract, `FirewallProxyIntercept`, is designed for scenarios in which the protocol has already been deployed and there is a desire to incorporate the firewall at the proxy level. This contract acts as an intermediary. It receives `delegatecall`s from the proxy and subsequently forwards them to the logic contract via `delegatecall`.

However, the `FirewallProxyIntercept` encounters specific challenges. As is known, the Transparent Proxy pattern differentiates between functions only callable by the admin and

other functions. Consequently, the admin can only invoke admin-specific functions, while non-admin users are restricted to accessing the logic contract's functions. The admin's interaction with a function in the logic is limited to using `upgradeToAndCall`, which allows specifying the function details in the logic contract they wish to call, typically reserved for invoking the `initialize` function after an upgrade.

The `FirewallProxyIntercept` contract includes some only admin functions (e.g., the `initialize` function). However, due to the admin's inability to interact with the implementation directly, these functions cannot be directly invoked. To circumvent this limitation, the admin must employ the `upgradeToAndCall` method, retaining the intercept proxy's address as-is, and encode the call to the admin function within the `data` field. When using `UpgradeToAndCall` to change firewall settings, the only event emitted is `event Upgraded(address indexed implementation)`. This event will be emitted at each change of firewall settings with the same address and thus appear confusing to any off-chain monitoring setup, without emitting the real change in firewall settings.

Another challenge arises when the admin also needs to initialize the underlying logic contract after an upgrade. One way to upgrade the underlying logic contract is via the `FirewallProxyIntercept`'s `initialize` function, which does not support immediate execution of an initialization call. This gap could leave the logic contract uninitialized for a brief moment, presenting some potential risk.

All these changes inherent in the dual role of the `FirewallProxyIntercept`, both as a proxy and an implementation, make the correct usage susceptible to unintended errors. A user might mistakenly attempt to upgrade the logic with `upgradeToAndCall` and unintentionally overwrite the address of the `FirewallProxyIntercept`.

Consider documenting each step that users must take to overcome these challenges, as they may not be intuitive for the users and could lead to problems.

Update: *Resolved in pull request #19 at commit 231e2ea. The Ironblocks team stated:*

> *Added comments about the difference from the standard proxy behavior.*

Low Severity

L-01 Lack of Event Emission for Firewall Setting Changes

No event is emitted when the `firewall` is updated or the `firewallAdmin` is updated in the following contracts:

- FirewallProxyIntercept.sol
- FirewallTransparentUpgradeableProxy.sol
- FirewallConsumerBase.sol

Consider adding event emission to these critical state changes to the firewall setting reflecting both the old and new values for off-chain monitoring.

Update: *Resolved in pull request #14.*

L-02 Simulation Mode in `ApprovedCallsPolicy`

In `ApprovedCallsPolicy`, the variable IS_EXECUTING_SIMULATION_SLOT can be read in `_is_executing_simulation()`. However, it cannot be set, thereby always returning `false` for all on-chain transactions. Although this variable provides a way for the `SIGNER_ROLE` to override local states and simulate a sequence of calls off-chain before approving them, a simulation effect can be achieved by other ways (e.g., in a forked environment).

Since the simulation does not affect any present on-chain logic in the `ApprovedCallsPolicy`, consider removing it to improve code clarity.

Update: *Resolved in pull request #4.*

L-03 `try/catch Error(string memory){}` in `CombinedPoliciesPolicy` May Not Catch Call Cases

In `CombinedPoliciesPolicy`, the result from executing each subscribed policy is set to be `true` if the `preExecution` does not revert with an error string. The result is updated to

`false` if the `postExecution` reverts with an error string. The entire result sequence is compared to the allowed combinations. The `CombinedPoliciesPolicy: Disallowed combination` error will be returned if it does not match any allowed combination.

Note that the returned error string is unused. More importantly, if an execution reverts without any error message (e.g., a policy reverting with `require(success)` or `assembly {revert(offset, size)}`) the `try/catch Error(string memory){}` hook will not be able to catch it. The transaction will, in fact, revert even though such a revert is an explicitly-allowed failure in the `isAllowedCombination`.

For the policies in scope, all reverts contain a string error message. To prevent such cases for future policy implementation, consider using `try/catch {}` instead of `try/catch Error(string memory){}` to catch all types of failures.

Update: *Resolved in pull request #3.*

L-04 `ForbiddenMethodsPolicy` May Block More Transactions Than Intended

In `ForbiddenMethodsPolicy`, `tx.origin`, `block.number`, and `tx.gasprice` are used to remember the context when a forbidden method is entered and the policy reverts in the `postExecution` hook when the same context is detected. This context may apply to more transactions than intended.

`tx.origin`:

In the context of EIP-4337 Account Abstraction or the presence of a relayer, the `tx.origin` may be coming from a bundler or a trusted forwarder that sends out multiple transactions in the same block. On some L2 networks, if it is an L1→L2 transaction, the `tx.origin` is set to some special addresses on L2 (e.g., Optimism), thereby covering more transactions than intended.

`block.number`:

On some L2 network, the `block.number` may remain constant for some time until it is synced with the L1 `block.number`. For instance, Arbitrum's `block.number` is updated to sync with the L1 `block.number` approximately every minute. In this context, `block.timestamp` is recommended over `block.number`.

`tx.gasprice`:

Gas price can be the same for different transactions in the same block on the Ethereum network. Here are some examples in the Ethereum block 19232990 where the third, fourth, fifth, and sixth transaction all have the same gas price of `0.000000022986457525 ETH` according to the Etherscan user interface.

The combination of the above scenarios suggests that the `ForbiddenMethodsPolicy` can block more transactions than intended. As such, consider documenting the risks involved and strategies for a workaround, and using `block.timestamp` over `block.number` if intending to support L2 networks.

Update: *Resolved in pull request #12 and pull request #28 at commit aeb8f9f with additional comments.*

L-05 Risk of Function Signature Clash With `ifAdmin`

The `ifAdmin` modifier is deprecated due to the risk of function signature clash in the OpenZeppelin Contracts library's `v4.9.3` release, which is the dependency version used in this project. The `ifAdmin` modifier is used in the FirewallTransparentUpgradeableProxy and the FirewallProxyIntercept contracts to guard `firewall`- and `firewallAdmin`-related functions. This increases the risk of function signature clash with implementation contracts.

Consider allowing a clean separation of admin functionality and user calls to minimise the risk of signature clash. See, for instance, the `v5.0` release of the TransparentUpgradeableProxy.

Update: *Resolved in pull request #23 at commit 3fb999b.*

L-06 Policies Can Be Executed When No Longer Approved

The owner of the `Firewall` contract can set policy status and only approved policies can be subscribed by consumers.

In the situation where a policy is compromised, the owner of the `Firewall` contract can set the policy status to `false` to disapprove the policy, preventing further subscription to the compromised policy. However, existing consumers of that compromised policy are still executing it unless each consumer's `firewallAdmin` removes the policy individually. This creates a window where consumers can execute disapproved policies, subject to risks of a vulnerable policy.

Consider documenting this risk clearly and recommending consumer admins to monitor subscribed policies' status. Alternatively, consider giving an option to the firewall owner to disable any disapproved policies temporarily until further actions from the consumer admin.

Update: *Resolved in pull request #22 at commit e56b1e6. The Ironblocks team stated:*

> *Added comments on this for functions that subscribe consumers to policies.*

L-07 Implementation Address of the `FirewallProxyIntercept` May Not Be Compatible With Etherscan

The "eip1967.firewall.intercept.implementation" storage slot contains the underlying business logic of the implementation address for the `FirewallProxyIntercept` contract. However, it may be incompatible with block explorers such as etherscan, that point the implementation address to the `eip1967.proxy.implementation`. In such a case, the `TransparentUpgradeableProxy` will point to `FirewallProxyIntercept` and the `FirewallProxyIntercept` will point to the same slot on the intercept, set in the constructor, which may not always be the same as that in the `FIREWALL_INTERCEPT_IMPLEMENTATION_STORAGE_SLOT`.

To ensure that users can seamlessly work with this contract, consider documenting this potential mismatch.

Update: *Resolved in pull request #21 at commit d924e9f. The Ironblocks team stated:*

> *Added comment about the limitation of interoperability with common block explorers.*

L-08 `staticCallCheck` Can Be Circumvented

The `FirewallTransparentUpgradeableProxy` intends to skip `view` functions with the `_isStaticCall` check. A `staticcall` will result in the `staticCallCheck` function to revert, thus returning `true` when it happens. This strategy can be circumvented using a low-level `call` to a `view` function and triggering the `pre` and `post` hooks on subscribed policies, thereby rendering this check ineffective.

To avoid unexpected results for users integrating with this contract, consider documenting this specific case.

Update: *Resolved in pull request #24 at commit 3711619. The Ironblocks team stated:*

> *Added a comment regarding the limitations of static call checks.*

L-09 Array Lengths May Mismatch

In `CombinedPoliciesPolicy`, the policy administrator can set the allowed combinations and policies using an `address[]` array for the policies and a double boolean array `bool[][]` for the allowed combinations.

The data structure `bool[][]` for the input `_allowedCombinations` allows boolean arrays of different lengths. For example, `[[true],[true, false], [true, false, true]]` passed as an `_allowedCombinations` will succeed regardless of the length of `_policies`. Each inner array is hashed together as a key used to compare with the hash of the resulting array in `postExecution` which is always a boolean array of the same length as the policies. An accidental mistake when setting allowed combinations may result in unintended reverts.

To ensure correct behavior, consider enforcing that the lengths of the inner arrays in `_allowedCombinations` match the length of `_policies`.

Update: *Resolved in pull request #2 at commit 411530b.*

L-10 Limitations of `AdminCallPolicy`

The `AdminCallPolicy` is a policy that requires a third party with the `APPROVER_ROLE` to approve any admin calls. When a call is approved, the hash of the call together with the `block.timestamp` at approval is stored in a mapping. The hash of the call is determined by five parameters, and once the call is approved, the user has one day to execute it. However, a challenge arises if the admin intends to execute multiple calls within the same transaction using the same call hash, as this would not be feasible.

Consider modifying the system to allow the admin to execute the same call hash multiple times within the same transaction, thereby covering all potential use cases.

Update: *Resolved in pull request #27 at commit c8e2e24. The Ironblocks team stated:*

> *We decided to leave this as-is and instead document this limitation as a known trade-off, thereby balancing security and usability (we do not foresee a practical use case*

> *where an admin would need to call the same function multiple times in a single transaction for a function that is protected by notary approvals).*

L-11 Missing Docstrings

Throughout the codebase, there are multiple instances of code that is missing docstrings, for example, but not limited to:

- The `APPROVER_ROLE` state variable in `AdminCallPolicy.sol`.
- The `expirationTime` state variable in `AdminCallPolicy.sol`.
- The `adminCallHashApprovalTimestamp` state variable in `AdminCallPolicy.sol`.
- The `preExecution` function and in the `postExecution` function in different policies.

Consider thoroughly documenting all functions (and their parameters) that are part of any contract's public API. Functions implementing sensitive functionality, even if not `public`, should be clearly documented as well. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

Update: *Resolved in pull request #25 at commit 988f358.*

L-12 Use of `tx.origin` to Validate EOA Sender

In the `OnlyEOAPolicy`, the sender is considered an EOA if `sender == tx.origin`. If the transaction is coming via a trusted forwarder as in the ERC-2771 meta-transaction, it is possible that the sender is an EOA but not equal to `tx.origin`.

Consider documenting this specific case and/or try handling this validation differently when it comes to a meta-transaction.

Update: *Resolved in pull request #26 at commit b065d73. The Ironblocks team stated:*

> *Added a comment to the EOA policy.*

Notes & Additional Information

N-01 Unused Imports

In `FirewallProxyAdmin.sol`, there are imports that are duplicated and unused:

- The import `import "@openzeppelin/contracts/utils/Address.sol";`.
- The import `import "../interfaces/IFirewall.sol";`.
- The import `import "../interfaces/IFirewallConsumer.sol";`.

Consider removing unused imports to improve the overall clarity and readability of the codebase.

Update: *Resolved in pull request #7.*

N-02 Unnecessary Casts

In `BalanceChangePolicy.sol`, there are unnecessary casts:

- The `address(consumer)` cast
- The `address(consumer)` cast

To improve the overall clarity, intent, and readability of the codebase, consider removing unnecessary casts.

Update: *Resolved in pull request #8.*

N-03 Lack of Indexed Event Parameter

In `Firewall.sol`, the `PolicyStatusUpdate` event does not have indexed parameters.

To improve the ability of off-chain services to search and filter for specific events, consider indexing event parameters.

Update: *Resolved in pull request #9.*

N-04 Non-Explicit Imports Are Used

The use of non-explicit imports in the codebase can decrease code clarity and may create naming conflicts between locally-defined and imported variables. This is particularly relevant when multiple contracts exist within the same Solidity files or when inheritance chains are long.

Throughout the codebase, global imports are being used. Following the principle that clearer code is better code, consider using named import syntax (`import {A, B, C} from "X"`) to explicitly declare which contracts are being imported. Some examples are:

- The import "./FirewallPolicyBase.sol"; import in `AdminCallPolicy.sol`.
- The import "@openzeppelin/contracts/token/ERC20/IERC20.sol"; import in `BalanceChangePolicy.sol`.
- The import "@openzeppelin/contracts/utils/Address.sol"; import in `FirewallConsumerBase.sol`.
- The import "@openzeppelin/contracts/utils/Context.sol"; import in `FirewallConsumerBase.sol`.
- The import "./interfaces/IFirewall.sol"; import in `FirewallConsumerBase.sol`.

Update: *Resolved in pull request #10.*

N-05 Using `uint` Instead of `uint256`

Throughout the codebase, there are instances of `uint` being used as opposed to `uint256`. In favor of explicitness, consider replacing all instances of `uint` with `uint256`. Some examples of this are:

- In line 21 of `AdminCallPolicy.sol`
- In line 43 of `ApprovedCallsPolicy.sol`
- In line 37 of `BalanceChangePolicy.sol`

Update: *Resolved in pull request #11.*

N-06 Missing Named Parameters in Mappings

Since Solidity 0.8.18, developers can utilize named parameters in mappings. This means mappings can take the form of `mapping(KeyType KeyName? => ValueType ValueName?)`. This updated syntax provides a more transparent representation of a mapping's purpose.

Throughout the codebase, there are multiple mappings without named parameters. Consider adding named parameters to the mappings to improve the readability and maintainability of the codebase.

Update: *Resolved in pull request #20.*

Conclusion

Two high-severity issues were identified among various medium and lower-severity issues. We emphasize the need for a more comprehensive documentation on the protocol overall, particularly regarding the specifics of the smart contracts, some of which were described in the found issues. The Ironblocks team consistently responded to our inquiries made during the audit and throughout the two threat modeling sessions which helped us better understand the system.