Zellic

May 21, 2025

Prepared for
@eboadom
@kyzia551
@brotherlymite
Chaos Labs

Prepared by
Chongyu Lv
Ayaz Mammadov
Zellic

Chaos Labs Edge Agents

Smart Contract Security Assessment

Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Chaos Labs from May 16th to May 20th, 2025. During this engagement, Zellic reviewed Chaos Labs Edge Agents's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Does the A-specific agent admin have influence over other agents?
- Could a denial-of-service attack happen, preventing updates from being processed?
- Are the validations for updates sane and healthy?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.4. Results

During our assessment on the scoped Chaos Labs Edge Agents contracts, we discovered one finding, which was informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of Chaos Labs in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	0
🟧 High	0
🟨 Medium	0
🟩 Low	0
⬜ Informational	1

2. Introduction

2.1. About Chaos Labs Edge Agents

Chaos Labs contributed the following description of Chaos Labs Edge Agents:

> The goal of [the] EdgeAgent system is to make integration of EdgeRiskOracle into a protocol as secure and effortless as possible and trying to create a standardized framework as a middleware. EdgeAgent systems aims to create a common infra so protocols when integrating EdgeRiskOracle do not need to re-invent things, but can instead leverage the EdgeAgent infrastructure.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Chaos Labs Edge Agents Contracts

Type	Solidity
Platform	EVM-compatible

Target	edge-agents-zellic
Repository	https://github.com/bgd-labs/edge-agents-zellic/ ↗
Version	`46e00195b0f8d92655d6860257a111e4c9dcbecb`

Programs	`EdgeAgentHub.sol`
	`EdgeConfigurator.sol`
	`agent/BaseAgent.sol`
	`modules/RangeValidationModule.sol`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of one person-week. The assessment was conducted by two consultants over the course of half a calendar week.

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io

Chad McDonald
Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Chongyu Lv
Engineer
chongyu@zellic.io

Ayaz Mammadov
Engineer
ayaz@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

May 16, 2025	Kick-off call
May 20, 2025	Start of primary review period
May 20, 2025	End of primary review period

3. Detailed Findings

3.1. Potential denial of service in `EdgeAgentHub::execute`

Target	EdgeAgentHub		
Category	Business Logic	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `EdgeAgentHub::execute` function is used to inject updates for agents, which should be run when the `EdgeAgentHub::check` returns true. In this function, a nested for loop is used to iterate over the address array in the `actionData` structure array `address[] markets;`:

```
for (uint256 i = 0; i < actionData.length; i++) {
      // ...
      address[] memory markets = actionData[i].markets;

      // ...
      address[] memory configuredMarkets = _getAgentMarkets(
        config,
        basicConfig.isMarketsFromAgentEnabled,
        basicConfig.agentAddress,
        agentId
      );
      for (uint256 j = 0; j < markets.length; j++) {
        // The Risk Oracle is expected to revert if we query a non-existing
    update.
        // In that case, we simply skip the market
        try

    IRiskOracle(basicConfig.riskOracle).getLatestUpdateByParameterAndMarket(
            updateType,
            markets[j]
          )
        returns (IRiskOracle.RiskParameterUpdate memory update) {
      // ...
```

However, the `EdgeAgentHub::execute` function does not limit the length of the `markets` array and the `configuredMarkets` array. If the length of the `markets` array and the `configuredMarkets` array is too large, the function may revert due to being out of gas.

Impact

When the length of the `address[] markets` array in `struct ActionData` is too large, the `EdgeAgentHub::execute` function cannot be executed to inject updates to the agent.

Here is a proof of concept:

```
diff --git a/src/contracts/EdgeAgentHub.sol b/src/contracts/EdgeAgentHub.sol
index 49dba60..98c0132 100644
--- a/src/contracts/EdgeAgentHub.sol
+++ b/src/contracts/EdgeAgentHub.sol
@@ -1,6 +1,7 @@
 // SPDX-License-Identifier: BUSL-1.1
 pragma solidity ^0.8.27;

+import {console2} from 'forge-std/console2.sol';
 import {EnumerableSet}
    from 'openzeppelin-contracts/contracts/utils/structs/EnumerableSet.sol';

 import {IRiskOracle} from './dependencies/IRiskOracle.sol';
@@ -29,6 +30,7 @@ contract EdgeAgentHub is EdgeConfigurator, IEdgeAgentHub {
   function check(
     uint256[] memory agentIds
   ) public view virtual returns (bool, ActionData[] memory) {
+    console2.log("in EdgeAgentHub::check ,,, agentIds.length:  ",
    agentIds.length);
     ActionData[] memory actionData = new ActionData[](agentIds.length);
     uint256 actionCount;

@@ -101,6 +103,7 @@ contract EdgeAgentHub is EdgeConfigurator, IEdgeAgentHub {
     for (uint256 i = 0; i < actionData.length; i++) {
       uint256 agentId = actionData[i].agentId;
       address[] memory markets = actionData[i].markets;
+      console2.log("in EdgeAgentHub::execute ,,, markets.length:  ",
    markets.length);

       AgentConfig storage config = $.config[agentId];
       BasicConfig memory basicConfig = config.basicConfig;
@@ -119,6 +122,7 @@ contract EdgeAgentHub is EdgeConfigurator, IEdgeAgentHub {
         basicConfig.agentAddress,
         agentId
       );
+      console2.log("in EdgeAgentHub::execute ,,, configuredMarkets.length:  ",
    configuredMarkets.length);

       for (uint256 j = 0; j < markets.length; j++) {
         // The Risk Oracle is expected to revert if we query a non-existing
    update.
diff --git a/tests/EdgeAgentHub.t.sol b/tests/EdgeAgentHub.t.sol
index 1adc26c..64e8834 100644
--- a/tests/EdgeAgentHub.t.sol
+++ b/tests/EdgeAgentHub.t.sol
@@ -1,6 +1,7 @@
 // SPDX-License-Identifier: BUSL-1.1
 pragma solidity ^0.8.0;

+import "forge-std/console2.sol";
 import {Test, Vm} from 'forge-std/Test.sol';
 import {Strings} from 'openzeppelin-contracts/contracts/utils/Strings.sol';
 import {OwnableUpgradeable} from 'openzeppelin-contracts-
    upgradeable/contracts/access/OwnableUpgradeable.sol';
@@ -358,6 +359,51 @@ contract EdgeAgentHub_Test is Test {
     _edgeHub.execute(actions);
   }

+  function test_DOS_execute_1() public{
+    uint256 agentId = _registerAgent();
+    // vm.prank(AGENT_ADMIN);
+    // _edgeHub.setAgentAsPermissioned(agentId, true);
+
+    address[] memory langMarketsArray = new address[](1);
+    for(uint256 i=0;i<langMarketsArray.length;i++){
+      address tmp_newMarket = address(uint160(5)); // should be Market
    address ,,, in allowedMarkets: markets
+      _addUpdateToRiskOracle(tmp_newMarket);
+      langMarketsArray[i] = tmp_newMarket;
+    }
+    IEdgeAgentHub.ActionData[] memory actions
    = new IEdgeAgentHub.ActionData[](1);
+    actions[0] = IEdgeAgentHub.ActionData({agentId: agentId, markets:
    langMarketsArray});
+    // vm.expectRevert(IEdgeAgentHub.NoActionCanBePerformed.selector);
+
+    uint256 gasBefore = gasleft();
+    _edgeHub.execute(actions);
+    uint256 gasAfter = gasleft();
+    uint256 gasUsed = gasBefore - gasAfter;
+    console2.log("in test_DOS_execute_1 ,,, gasUsed: ",gasUsed);
+  }
+
+  function test_DOS_execute_100() public{
+    uint256 agentId = _register_100_Agent();
+    // vm.prank(AGENT_ADMIN);
+    // _edgeHub.setAgentAsPermissioned(agentId, true);
+    address[] memory langMarketsArray = new address[](100);
+    for(uint256 i=0;i<langMarketsArray.length;i++){
+      address tmp_newMarket = address(uint160(0x10000+i)); // should be
    Market address ,,, in allowedMarkets: markets
+      _addUpdateToRiskOracle(tmp_newMarket);
+      langMarketsArray[i] = tmp_newMarket;
+    }
+    IEdgeAgentHub.ActionData[] memory actions
    = new IEdgeAgentHub.ActionData[](1);
+    address[] memory tmp_markets = new address[](1);
+    tmp_markets[0] = address(0x10000);
+    actions[0] = IEdgeAgentHub.ActionData({agentId: agentId, markets:
    tmp_markets});
+    // vm.expectRevert(IEdgeAgentHub.NoActionCanBePerformed.selector);
+
+    uint256 gasBefore = gasleft();
+    _edgeHub.execute(actions);
+    uint256 gasAfter = gasleft();
+    uint256 gasUsed = gasBefore - gasAfter;
+    console2.log("in test_DOS_execute_100 ,,, gasUsed: ",gasUsed);
+  }
+
   function test_restrictedMarketsNotInjected_marketsFromAgentEnabled()
    public {
     uint256 agentId = _registerAgent();
     address newMarket = address(105);
@@ -662,6 +708,32 @@ contract EdgeAgentHub_Test is Test {
     );
     vm.stopPrank();
   }
+  function _register_100_Agent() internal returns (uint256 agentId){
+    address[] memory markets = new address[](100);
+    for(uint256 i=0;i<100;i++){
+      markets[i] = address(uint160(0x10000+i));
+    }
+
+    vm.startPrank(HUB_OWNER);
+    agentId = _edgeHub.registerAgent(
+      IEdgeConfigurator.AgentRegistrationInput({
+        agentAddress: address(_agent),
+        riskOracle: address(riskOracle),
+        admin: AGENT_ADMIN,
+        agentContext: abi.encode(TARGET_CONTRACT),
+        isAgentEnabled: true,
+        isAgentPermissioned: false,
+        isMarketsFromAgentEnabled: false,
+        expirationPeriod: 1 days,
+        minimumDelay: 0,
+        updateType: UPDATE_TYPE,
+        allowedMarkets: markets,
+        restrictedMarkets: new address[](0),
+        permissionedSenders: new address[](0)
+      })
+    );
+    vm.stopPrank();
+  }

   function _addUpdateToRiskOracle(address market) internal {
     _addUpdateToRiskOracle(market, UPDATE_TYPE);
```

Here is the result:

```
[PASS] test_DOS_execute_1() (gas: 576252)
Logs:
  in EdgeAgentHub::execute ,,, markets.length:  1
  in EdgeAgentHub::execute ,,, configuredMarkets.length:  1
  in test_DOS_execute_1 ,,, gasUsed:  54229

[PASS] test_DOS_execute_100() (gas: 26899962)
Logs:
  in EdgeAgentHub::execute ,,, markets.length:  1
  in EdgeAgentHub::execute ,,, configuredMarkets.length:  100
  in test_DOS_execute_100 ,,, gasUsed:  69448
```

Recommendations

Consider limiting the length of the `configuredMarkets` array and the `struct ActionData` array. This ensures that a single agent will not revert due to being out of gas when executing `EdgeAgentHub::execute`.

Remediation

This issue has been acknowledged by Chaos Labs.

Chaos Labs provided the following comment,

> We think the extra gas consumed is accepted and should not cause out of gas in realistic cases.

The team responded that the likely upperbound of the number of markets would be 100, given such an upperbound, testing has shown that the estimated gas used with 100 markets is around

~200,000 gas. This seems to be an acceptable amount indicating that even more markets would still be functional.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Failed updates during `execute` are not tracked or made public

During a call to the function `execute` — responsible for finally gathering and processing risk-oracle updates — failed validations or updates from risk oracles are not emitted as events or cause a revert of any type, if there was a single valid action. While not explicitly a security boundary, in the context of risk oracles, this could be problematic if a risk oracle is forgotten or slips through due to operational issues and does not correctly post risk updates. This could result in issues with liquidations and bad debt. However, it is also important to note that omitting events could save on gas costs.

4.2. Test suite

The test suite adequately covers most of the possible scenarios in the business logic. Most of the testing is unit tests that enforce certain core invariants. There is no fuzz testing or end-to-end testing on the testnet; this seems reasonable for the scope of this business logic as it is fairly straightforward and does not contain complex structure or byte parsing.

5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1. Module: BaseAgent.sol

Function: `inject(uint256 agentId, bytes agentContext, IRiskOracle.RiskParameterUpdate update)`

Abstract base contract to be inherited by the agents to do agent specific validation and injection.

Inputs

- `agentId`
    - **Control**: Full.
    - **Constraints**: Must be a valid agent ID.
    - **Impact**: The agent to which the update will be injected.
- `agentContext`
    - **Control**: Full.
    - **Constraints**: None.
    - **Impact**: The agent-specific context needed for processing the update.
- `update`
    - **Control**: None.
    - **Constraints**: N/A.
    - **Impact**: The standardized risk update object to be processed.

Branches and code coverage

Intended branches

- Only the EdgeAgentHub should be allowed to call this.
    - ☑ Test coverage

Function call analysis

- `this._processUpdate(agentId, agentContext, update)`

- **What is controllable?** `agentId` and `agentContext`.
- **If the return value is controllable, how is it used and how can it go wrong?** N/A.
- **What happens if it reverts, reenters or does other unusual control flow?** N/A.

5.2. Module: EdgeAgentHub.sol

Function: `check(uint256[] agentIds)`

The `check()` method is called to check if for an agent, an update can be injected.

Inputs

- `agentIds`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: The list of `agentIds` for which updates should be checked.

Branches and code coverage

Intended branches

- Skip the market if the risk oracle is expected to revert when querying a nonexisting update.
  - ✓ Test coverage
- Stop collecting data if the maximum batch size has been reached, to protect against gas overflow on execution.
  - ✓ Test coverage
- Validate if the update is not expired.
  - ✓ Test coverage
- Validate if the `updateId` has not been executed before.
  - ✓ Test coverage
- Validate if the minimum delay has passed since the last update for an agent and market.
  - ✓ Test coverage

Function: `execute(ActionData[] actionData)`

The `execute()` method is called to inject/push the update from the EdgeRiskOracle into the protocol.

Inputs

- `actionData`
    - **Control**: Arbitrary.
    - **Constraints**: Should call the `check()` method first to check if for an agent, an update can be injected.
    - **Impact**: Actions contain the list of `ActionData` for which updates can be injected.

Branches and code coverage

Intended branches

- Skip the market if the risk oracle is expected to revert when querying a nonexisting update.
    - ☑ Test coverage
- Skip if an agent is disabled or the user is not authorized.
    - ☑ Test coverage
- Check that the market from the update corresponds to the configured markets on the agent.
    - ☑ Test coverage
- Validate if the update is not expired.
    - ☑ Test coverage
- Validate if the `updateId` has not been executed before.
    - ☑ Test coverage
- Validate if the minimum delay has passed since the last update for an agent and market.
    - ☑ Test coverage
- Check that the market from the update corresponds to the configured markets on the agent.
    - ☑ Test coverage

Negative behavior

- ☑ Revert if `hasValidActions` is false.

5.3. Module: EdgeConfigurator.sol

Function: `registerAgent(AgentRegistrationInput input)`

This is the principal function to register an agent; it sets all the states for an agent.

Inputs

- `input`
    - **Control**: Full.
    - **Constraints**: Several sanity checks in for the agent state.
    - **Impact**: The input.

Branches and code coverage

Intended branches

- Set all the relevant states.
    - ☑ Test coverage
- Agents' IDs sequentially increase.
    - ☑ Test coverage

5.4. Module: RangeValidationModule.sol

Function: `validate(address edgeHub, uint256 agentId, RangeValidationInput input)`

This method will be called by the EdgeAgentHub to do custom validation/checks for one's agent. The `validate()` method passes the following params from the hub that will be used for validation: `agentId`, `agentContext`, and the `update` struct from the risk oracle. The `validate()` method will be called per update only after it passes generic validation of the EdgeAgentHub. As a protocol, if the update from one's risk oracle passes their agent-specific validation, it would need to return true and false otherwise.

Inputs

- `edgeHub`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Address of the `edgeHub` contract of the agent.

- `agentId`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: The ID of the agent configured on the `edgeHub`.
- `input`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Values to validate range.

Branches and code coverage

Intended branches

- If the config is not set for a specific market, fall back to the default configuration.
  - ☑ Test coverage

6. Assessment Results

During our assessment on the scoped Chaos Labs Edge Agents contracts, we discovered one finding, which was informational in nature.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.