Zellic

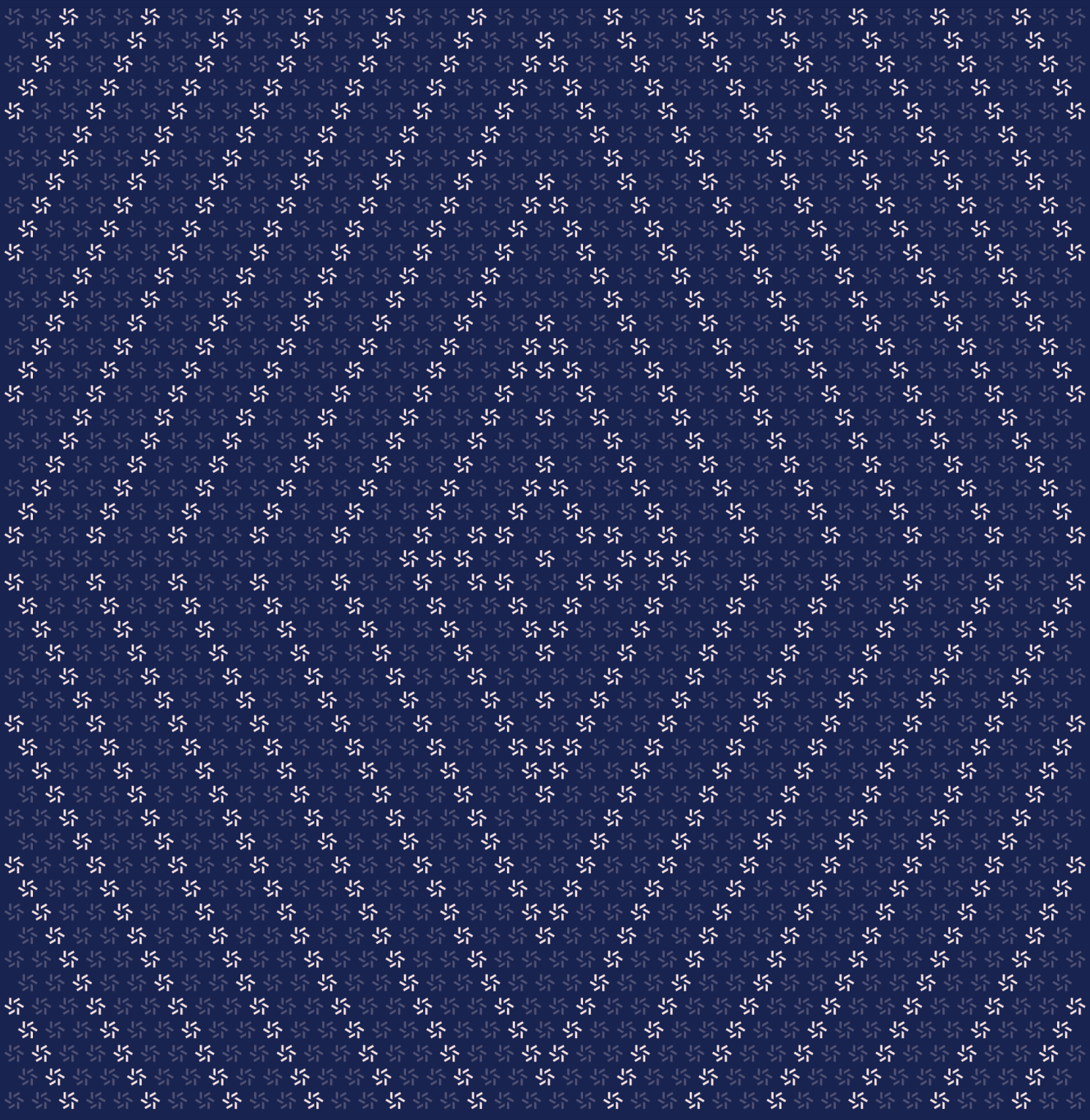
Prepared for
Anish Agnihotri
Ritual

Prepared by
Seunghyeon Kim
Junyi Wang
Zellic

June 27, 2024

Infernet SDK

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 Ritual from June 20th to June 27th, 2024. During this engagement, Zellic reviewed Infernet SDK'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:

- Could a malicious node drain the tokens from another's wallet?
- Could a malicious user make the service DoS?
- Is the `Wallet` designed to handle the tokens well?
- Is the access control for the contracts designed well?

1.3. Non-goals and Limitations

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

- Prover contract
- 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.

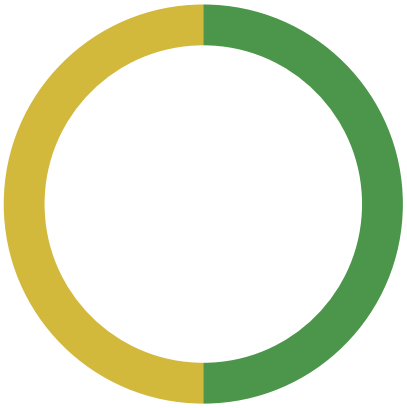
1.4. Results

During our assessment on the scoped Infernet SDK contracts, we discovered two findings. No critical issues were found. One finding was of medium impact and one was of low impact.

Additionally, Zellic recorded its notes and observations from the assessment for Ritual's benefit in the Discussion section (4. ↗) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	1
Low	1
Informational	0



2. Introduction

2.1. About Infernet SDK

Ritual contributed the following description of Infernet SDK:

> Ritual is building a sovereign layer for AI. Our first phase, Infernet, is the first product of its kind to enable developers to access models both on-chain via smart contracts and off-chain. Our next phase, the Ritual chain, will be a sovereign layer custom-built to support AI-native operations and enable a new class of applications at the intersection of crypto and AI.
>
> The Infernet SDK is the on-chain component of Infernet that lets users subscribe to outputs from off-chain compute workloads. Through simple consumer interfaces, out-of-the-box payment rails, async callback consumption, common developer patterns, and more, you can get started using Infernet in under 10 lines of code.

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:

Infernet SDK Contracts

Repository	https://github.com/ritual-net/infernet-sdk ↗
Version	infernet-sdk: `debcd16f37ee9c2203b163ccd6ef87cf2187f14d`
Programs	• consumer/Base.sol • consumer/Callback.sol • consumer/Subscription.sol • pattern/Allowlist.sol • pattern/Delegator.sol • payments/Fee.sol • payments/IProver.sol • payments/Wallet.sol • payments/WalletFactory.sol • utility/Coordinated.sol • Coordinator.sol • EIP712Coordinator.sol • Inbox.sol • Registry.sol
Type	Solidity
Platform	EVM-compatible

2.4. Project Overview

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

Contact Information

The following project manager was associated with the engagement:

Chad McDonald
Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Seunghyeon Kim
Engineer
seunghyeon@zellic.io

Junyi Wang
Engineer
junyi@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

June 20, 2024	Start of primary review period
June 27, 2024	End of primary review period

3. Detailed Findings

3.1. Invalid result delivered to subscriber

Target	Coordinator.sol		
Category	Business Logic	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The Coordinator is responsible for receiving results from computation requests made by the subscriber contracts. The Coordinator then performs the necessary checks, makes the appropriate records and payments, and delivers the result to the subscriber. It supports multiple modes of operation, where the proof check for the delivered result data is performed synchronously and asynchronously, and delivers the result directly or stores it into an `Inbox` as well as other subscriber-configurable modes of operation. In one of the modes of operation, a proof is required, but the checking of the proof is done asynchronously. In this case, the result is delivered immediately to the subscriber contract, regardless of the result of the proof check.

Impact

It is possible for invalid result data to be delivered to a subscriber contract in this mode. The node delivering the invalid results will get slashed, however, increasing the cost of the attack and somewhat mitigating it.

Recommendations

Redesign this mode of operation according to protocol needs such that it does not have the above problem.

Remediation

This issue has been acknowledged by Ritual.

Ritual provided the following response:

> For applications with strict trust assumptions, eager verification prevents invalid results from being delivered to a consumer because proof verification is a strict precondition to compute output delivery. For other applications, lazy verification allows (1) immediate consumption of outputs, (2) reduced gas overhead, (3) the flexibility to asynchronously verify response proofs

in the future. Subscriptions that are lazily verified still maintain the guarantee that any payment is only processed on proof verification. This ensures that malicious nodes are still slashed.

Ritual acknowledges the issue but considers this flexibility a part of the intended functionality of the system. Additional documentation has been added to highlight this concern.

3.2. Redundancy number increments for invalid data

Target	Coordinator.sol		
Category	Business Logic	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The Coordinator is responsible for receiving results from computation requests made by the subscriber contracts. The Coordinator then performs the necessary checks, makes the appropriate records and payments, and delivers the result to the subscriber. The Coordinator supports a redundancy count, which means subscribers can request multiple compute results for the same query from distinct nodes. When submitting data with an invalid proof, the node is slashed, but the redundancy count is still incremented.

Impact

Potentially, the subscriber could end up with less redundancy than requested. We note that the subscriber is compensated in this case from the slashed funds of the misbehaving nodes.

Recommendations

Either redesign the protocol to not increment the redundancy count for invalid data or require that all subscribers handle this situation in prominent documentation.

Remediation

This issue has been acknowledged by Ritual.

Ritual provided the following response:

> The Infernet SDK allows consumers to request redundancy in their compute requests. For example, an on-chain application consuming an LLM output may choose to set a higher redundancy to account for non-deterministic discrepancies across individual runs. This redundancy number makes no assumptions to the validity of returned data.
>
> For applications with strict trust assumptions, eager proof verification of compute data, downstream proof verification in consumer contracts during callback receipt, or a combination of both, avoids invalid data altogether.

In the rare case that a malicious node explicitly chooses to perform a denial of service attack by reducing available redundancy, they are materially slashed with the consumer being compensated for this reduced redundancy.

Ritual acknowledges the issue and has added additional documentation to highlight this concern.

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. We recommend that `SafeERC20` be used

In order to support more ERC-20 tokens safely, we recommend that `SafeERC20` functions be used for interacting with ERC20 contracts. ERC20 contracts have a very diverse range of API expectations, and unsafe handling of them could lead to loss of funds. Safely handling ERC20 is best done with a library like `SafeERC20`.

4.2. We recommend user pitfalls be prominently documented

Ritual is an intricate protocol requiring care from API users to avoid vulnerabilities. Some of these points of caution include the following:

- The prover must be a trusted entity.
- A whitelist is required if a prover is not used and the subscriber does not verify the message manually.
- If the official base classes are not used, the subscriber must check the caller of the callback functions.

Although Ritual's in-line documentation is quite thorough and easy to understand, we feel that the nature of the protocol warrants an additional emphasis on API–user-facing documentation.

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: Coordinator.sol

Function: `cancelSubscription(uint32 subscriptionId)`

This function is for canceling a specific subscription.

Inputs

- `subscriptionId`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: The owner of the given `subscriptionId` must be same as the function caller.
  - **Impact**: The subscription ID to cancel.

Branches and code coverage

Intended branches

- The selected subscription's `activeAt` must be max of the type `uint32`.
  - ☑ Test coverage

Negative behavior

- Revert when the function caller is not the owner of the selected subscription.
  - ☑ Negative test

Function: `createSubscription(string containerId, uint32 frequency, uint32 period, uint16 redundancy, bool lazy, address paymentToken, uint256 paymentAmount, address wallet, address prover)`

This function creates a subscription using the given inputs.

Inputs

- `containerId`

- **Control**: Completely controlled by the caller.
- **Constraints**: None.
- **Impact**: The container ID for the subscription.

- `frequency`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Frequency for the subscription.
- `period`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Period for the subscription.
- `redundancy`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Redundancy for the subscription.
- `lazy`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Lazy for the subscription.
- `paymentToken`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Payment token address for the subscription.
- `paymentAmount`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Payment amount for the subscription.
- `wallet`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Wallet address for the subscription.
- `prover`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Prover address for the subscription.

Function: `deliverCompute(uint32 subscriptionId, uint32 deliveryInterval, bytes input, bytes output, bytes proof, address nodeWallet)`

This function delivers container compute responses for the selected subscription.

Inputs

- `subscriptionId`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: The subscription matching the given ID must exist on the array `subscriptions`.
  - **Impact**: An ID of the subscription.
- `deliveryInterval`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: This value must match with the return value of `getSubscriptionInterval(subscription.activeAt, subscription.period)`.
  - **Impact**: Subscription `interval` to deliver.
- `input`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Off-chain input recorded by Infernet node.
- `output`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Off-chain container output.
- `proof`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Container execution proof or arbitrary metadata.
- `nodeWallet`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Address of the node wallet.

Branches and code coverage

Intended branches

- Subscription must be fulfilled with the right input values.
  - ☑ Test coverage

Negative behavior

- Revert when the subscription does not exist.
  - ☑ Negative test
- Revert if subscription is not yet active.
  - ☑ Negative test
- Revert if not processing the current interval.
  - ☑ Negative test

- Revert if interval is larger than frequency.
  - ☑ Negative test
- Revert if redundancy requirements for this interval have been met.
  - ☑ Negative test

Function: `finalizeProofValidation(uint32 subscriptionId, uint32 interval, address node, bool valid)`

This function is for finalizing and deleting the `proofRequest`. This is expected to be executed after the proof validation by the prover.

Inputs

- `subscriptionId`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: The subscription matching the given ID must exist on the array `subscriptions`.
  - **Impact**: An ID of the subscription.
- `interval`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Interval of subscription for which proof validation was requested.
- `node`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Node within the specified interval for which proof validation was requested.
- `valid`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Validity of the proof.

Branches and code coverage

Intended branches

- Transfer the payment token to the node wallet if the proof is valid.
  - ☑ Test coverage

Negative behavior

- Revert when the `proofRequest` does not exist.
  - ☑ Negative test

- Slash the node when the proof is invalid.
  - ☑ Negative test

5.2. Module: EIP712Coordinator.sol

Function: `createSubscriptionDelegatee(uint32 nonce, uint32 expiry, Subscription sub, uint8 v, byte[32] r, byte[32] s)`

This function verifies the `sub.owner` with ECDSA and creates a subscription using the given info when the verification succeeds.

Inputs

- `nonce`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Nonce for the digest used for verifying using the ECDSA.
- `expiry`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Expiry time stamp for the given function call.
- `sub`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Subscription to add.
- `v`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: Must be valid for the ECDSA verification.
  - **Impact**: The value `v` for the ECDSA.
- `r`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: Must be valid for the ECDSA verification.
  - **Impact**: The value `r` for the ECDSA.
- `s`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: Must be valid for the ECDSA verification.
  - **Impact**: The value `s` for the ECDSA.

Branches and code coverage

Intended branches

- Return existing `subscriptionId` when the subscription exists.
  - ☑ Test coverage
- The given subscription is added when the ECDSA verification is passed.
  - ☑ Test coverage

Negative behavior

- Revert when the expiry is reached.
  - ☑ Negative test
- Revert when the given subscription's owner address is different than the recovered address with the ECDSA.
  - ☑ Negative test

6. Assessment Results

At the time of our assessment, the reviewed code was deployed to the Ethereum Mainnet.

During our assessment on the scoped Infernet SDK contracts, we discovered two findings. No critical issues were found. One finding was of medium impact and one was of low impact.

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.