Sui Sentinel

Security Assessment

January 29th, 2026 — Prepared by OtterSec

Michał Bochnak	embe221ed@osec.io
Andreas Mantzoutas	andreas@osec.io
Sangsoo Kang	sangsoo@osec.io
Thiago Tavares	thitav@osec.io

Table of Contents

TABLE OF CONTENTS

01 — Executive Summary

Overview

Sui Foundation engaged OtterSec to assess the `sentinel` program. This assessment was conducted between December 6th, 2025 and January 21st, 2026. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 13 findings throughout this audit engagement.

In particular, we identified a critical vulnerability where the function responsible for consuming a prompt accepts signatures from any registered enclave without verifying that it is the authorized enclave (OS-SST-ADV-00), and another high-risk issue concerning the unbounded growth of the agent list, enabling an attacker to register multiple agents to inflate the Agent Registry object until it exceeds the maximum object size limit, resulting in further writes to fail (OS-SST-ADV-02).

Additionally, a call to fund an agent with zero SUI will reset the withdrawal lock timestamp without adding funds, allowing an attacker to block the creator from withdrawing, creating a denial-of-service scenario (OS-SST-ADV-01).

We also recommended refactoring the code to improve functionality and mitigate potential issues (OS-SST-SUG-00), and made suggestions to ensure adherence to coding best practices for better clarity and maintainability (OS-SST-SUG-01). We further advised optimizing the codebase by removing redundant logic, avoiding the emission of unnecessary events, and reclaiming available storage rebates (OS-SST-SUG-02).

We further confirm that all 13 reported issues, including both vulnerabilities and general findings, have been properly addressed and resolved by the team, with zero outstanding issues remaining.

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/sui-sentinel/contracts. This audit was performed against commit 0565a2a.

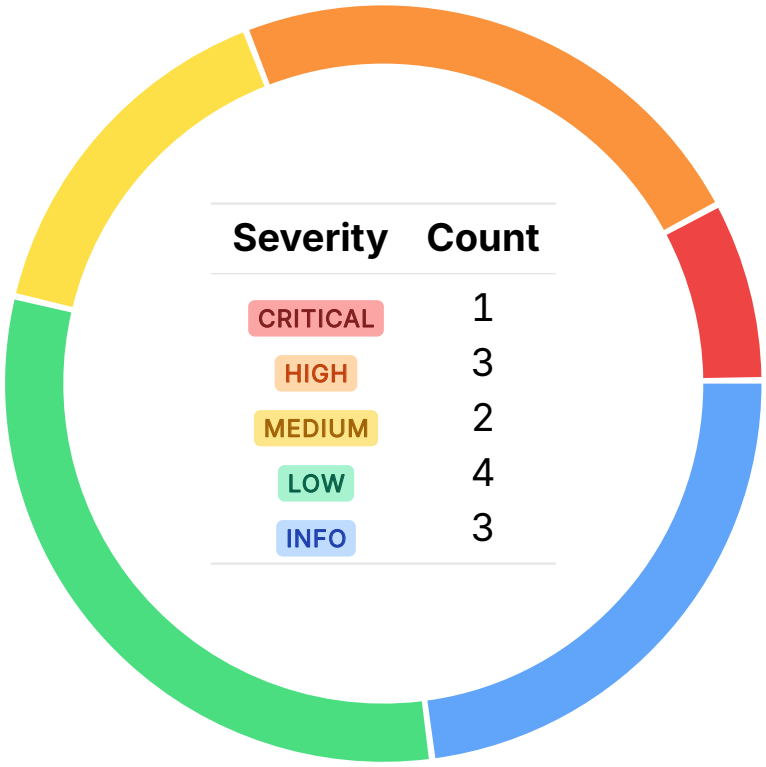
A brief description of the program is as follows:

Name	Description
sentinel	It enables a gamified, on-chain security challenge where Defenders deploy and fund AI “Sentinel” agents, and Attackers pay to attempt social-engineering attacks against them.

03 — Findings

Overall, we reported 13 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings do not have an immediate impact but will aid in mitigating future vulnerabilities.



Severity	Count
CRITICAL	1
HIGH	3
MEDIUM	2
LOW	4
INFO	3

04 — Vulnerabilities

Here, we present a technical analysis of the vulnerabilities identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-SST-ADV-00	CRITICAL	RESOLVED	`consume_prompt` accepts signatures from any registered enclave without verifying that it is the authorized enclave. Additionally, enclave upgrades may leave multiple valid enclaves active with no defined canonical executor.
OS-SST-ADV-01	HIGH	RESOLVED	A permissionless `fund_agent` call with zero `SUI` will reset the withdrawal lock timestamp without adding funds, allowing an attacker to block the creator from withdrawing.
OS-SST-ADV-02	HIGH	RESOLVED	`agent_list` grows unbounded as a `vector<String>`, enabling an attacker to register multiple agents to inflate the `AgentRegistry` object until it exceeds `max_move_object_size`, resulting in further writes to fail.
OS-SST-ADV-03	HIGH	RESOLVED	Immediate, unbounded updates to agent prompts and costs allow the creator to change behavior or price out users after funds are committed, enabling value capture and denial-of-service.
OS-SST-ADV-04	MEDIUM	RESOLVED	Signatures are not time-bound, allowing attackers to submit a signature at a later time when the agent has accumulated more funds.

OS-SST-ADV-05	MEDIUM	RESOLVED	The agent registration signature is not bound to the creator address, allowing an attacker to front-run the transaction and steal ownership of the agent.
OS-SST-ADV-06	LOW	RESOLVED	If an agent is never funded, `last_funded_timestamp` stays at zero, enabling the creator to immediately withdraw any rewards and bypass the intended lock period.
OS-SST-ADV-07	LOW	RESOLVED	The contract does not validate the `prompt` field in `consume_prompt`, allowing attackers to emit misleading prompt data in the `PromptConsumed` event.
OS-SST-ADV-08	LOW	RESOLVED	Unauthorized parties may create `Cap` objects utilizing the same type `T`, allowing them to register or modify enclaves without permission.
OS-SST-ADV-09	LOW	RESOLVED	Enclave destruction is authorized only by version comparison, allowing a different configuration with a higher version to delete unrelated enclaves.

Possibility to Utilize Arbitrary Enclaves CRITICAL OS-SST-ADV-00

Description

`consume_prompt` fails to verify the relationship between the `Agent` and the `Enclave`. The contract only checks that a signature is valid for a provided Enclave object, thus, as long as the attacker supplies any `Enclave<T>` object with a valid public key, the signature check passes, allowing an attacker to utilize a signature generated by a fake `Enclave` to win. Additionally, when enclave configurations are updated and multiple enclaves are present, it is not clearly defined which enclave will be selected for execution.

Remediation

Ensure that the provided `Enclave` is authorized, and explicitly specify the `Enclave` ID in the protocol configuration to explicitly identify the enclave to be utilized.

Patch

Resolved in e8165ef and 10ee5b7.

Extending Withdrawal Lock through Zero-Value Funding HIGH OS-SST-ADV-01

Description

`fund_agent` is permissionless and accepts zero-value `Coin<SUI>`. Since every call to `fund_agent` unconditionally updates `agent.last_funded_timestamp`, an attacker may pass a zero-value `Coin<SUI>` to reset `last_funded_timestamp` without adding funds. This extends the 14-day withdrawal lock even though no economic value is deposited. Thus, by repeatedly calling `fund_agent`, the attacker may indefinitely block the creator from withdrawing funds by preventing them from calling `withdraw_from_agent`, resulting in a denial-of-service scenario.

>_ contracts/app/sources/sentinel.move RUST

```rust
public fun fund_agent(agent: &mut Agent, payment: Coin<SUI>, clock: &Clock, ctx: &TxContext) {
    let amount = coin::value(&payment);
    let balance_to_add = coin::into_balance(payment);
    balance::join(&mut agent.balance, balance_to_add);

    // Update last funded timestamp
    let current_time = clock::timestamp_ms(clock);
    agent.last_funded_timestamp = current_time;

    let unlock_timestamp = current_time + WITHDRAWAL_LOCK_PERIOD_MS;
    [...]
}
```

Remediation

Implement proper access control logic for `fund_agent`, restricting anyone other than `agent_creator` from calling it.

Patch

Resolved in 72e7e2d.

Unbounded Agent Registry Growth HIGH OS-SST-ADV-02

Description

In `AgentRegistry`, `agent_list` is a `vector<String>` stored on-chain, resulting in `AgentRegistry` object to grow linearly with each new agent registration. An attacker may exploit this by registering a large number of agents, inflating the object's size until it reaches `max_move_object_size`. Once this limit is hit, any further writes to the registry will fail, effectively freezing agent registration.

Remediation

Remove `agent_list` since agent is already stored in a registry table.

Patch

Resolved in 59f6b5e.

Failure to Timelock Cost and Agent Prompt Updates HIGH OS-SST-ADV-03

Description

Both `update_agent_prompt` and `update_agent_cost` in `sentinel` allow the agent creator to make immediate, unilateral changes after users have already committed funds.

`update_agent_cost` allows the creator to increase the cost per message instantly and without any upper bound. After sufficient rewards accumulate, the creator may set an extremely high cost, preventing any further interactions with the agent. This effectively bricks the agent, blocking attackers from competing to claim the reward.

>_ *contracts/app/sources/sentinel.move* RUST

```rust
/// Update agent cost per message (only by creator)
public fun update_agent_cost(agent: &mut Agent, new_cost: u64, ctx: &TxContext) {
    assert!(agent.creator == ctx.sender(), ENotAuthorized);
    agent.cost_per_message = new_cost;
}
```

Similarly, `update_agent_prompt` allows the agent creator to modify the system prompt immediately and without any delay. Because the system prompt directly influences how the agent responds, a creator can wait until attackers have committed funds and then update the prompt to make the agent trivially solvable or deterministically *fail*, ensuring the creator (or a colluding address) may win and extract the accumulated rewards.

>_ *contracts/app/sources/sentinel.move* RUST

```rust
/// Update agent system prompt (only by creator)
public fun update_agent_prompt(agent: &mut Agent, new_prompt: String, ctx: &TxContext) {
    assert!(agent.creator == ctx.sender(), ENotAuthorized);
    agent.system_prompt = new_prompt;
}
```

Remediation

Enforce a timelock on cost updates and prompt updates in `update_agent_cost` and `update_agent_prompt`, respectively. Also, add protocol-level bounds on `cost_per_message` in `update_agent_cost`.

Patch

Resolved in 04f3adb and 432b856.

Absence of Signature Expiration Checks MEDIUM OS-SST-ADV-04

Description

The contract accepts enclave signatures without enforcing any timestamp or expiry checks, allowing them to remain valid indefinitely. Thus, a user may obtain a valid signature and delay submission until the agent accumulates more funds. Once submitted, the stale signature still verifies and will drain newly added rewards.

Remediation

Ensure signatures expire after a certain period of time.

Patch

Resolved in e8165ef.

Front-Running Agent Registration MEDIUM OS-SST-ADV-05

Description

`RegisterAgentResponse` does not include the creator address, so the enclave's signature is not bound to a specific caller. Because the signed payload only includes `agent_id`, `cost_per_message`, and `system_prompt`, any user may frontrun a `register_agent` call and submit their own `register_agent` transaction utilizing the same fields to effectively steal the `agent_id` and become the agent's creator, enabling them to update the prompt and the cost per message as they wish.

>_ *contracts/app/sources/sentinel.move* RUST

```rust
public struct RegisterAgentResponse has copy, drop {
    agent_id: String,
    cost_per_message: u64,
    system_prompt: String,
    is_defeated: bool
}
```

Remediation

Include the intended creator address in `RegisterAgentResponse` and verify that it matches `ctx.sender` during `register_agent`.

Patch

Resolved in 72e7e2d.

Potential to Bypass Withdrawal Lock Period LOW OS-SST-ADV-06

Description

When an agent is created, `last_funded_timestamp` is left at its default value (0) in `register_agent` unless `fund_agent` is explicitly called. As a result, if the agent later receives funds through other flows (for example, if someone submits a request for attack), the withdrawal logic in `withdraw_from_agent` becomes unsafe.

>_ *contracts/app/sources/sentinel.move* RUST

```
public fun register_agent<T>(
    registry: &mut AgentRegistry,
    agent_id: String,
    [...]
    enclave: &Enclave<T>,
    ctx: &mut TxContext,
) {
    [...]
    let agent = Agent {
        id: object::new(ctx),
        agent_id,
        creator,
        cost_per_message,
        system_prompt,
        balance: balance::zero(),
        last_funded_timestamp: 0, // Initialize to 0
    };
    [...]
}
```

The function computes `time_since_last_funding` by subtracting the `agent.last_funded_timestamp` from `current_time` to determine if it met the `WITHDRAWAL_LOCK_PERIOD_MS`. However, with `last_funded_timestamp = 0`, this check trivially passes. This allows the agent creator to immediately withdraw newly acquired funds, bypassing the intended lock period.

Remediation

Initialize `last_funded_timestamp` to the current time (at which the registration is done) at agent registration so that the withdrawal lock applies even before the first explicit funding.

Patch

Resolved in 72e7e2d.

Unauthenticated Prompt in Events LOW OS-SST-ADV-07

Description

In the current flow, the enclave signature only covers the `ConsumePromptResponse` fields. The `prompt` string is not validated, enabling an attacker to supply an arbitrary `prompt` value when calling `sentinel::consume_prompt`, which will consequently be emitted by the `PromptConsumed` event. This compromises the integrity of the `PromptConsumed` event.

Remediation

Verify the correct prompt is supplied in `consume_prompt`.

Patch

Resolved in ae5d5ad.

Unauthorized Capability Creation Risk LOW OS-SST-ADV-08

Description

`enclave::new_cap` currently allows anyone with access to a type `T` to create a `Cap<T>`. This is risky because the `Cap` object grants authority to create and modify `EnclaveConfig` and register new enclaves. A malicious actor may re-utilize the same type `T` as a legitimate module to generate unauthorized caps, enabling them to create enclave configurations or register enclaves without permission.

>_ *contracts/enclave/sources/enclave.move* RUST

```rust
/// Create a new `Cap` using a `witness` T from a module.
public fun new_cap<T: drop>(_: T, ctx: &mut TxContext): Cap<T> {
    Cap {
        id: object::new(ctx),
    }
}
```

Remediation

Require a one-time witness from the module.

Patch

Resolved in 10ee5b7.

Improper Enclave Destruction Authorization Logic LOW OS-SST-ADV-09

Description

`enclave::destroy_old_enclave` only checks that the supplied configuration has a higher version, without verifying that the enclave was created from that configuration. Because multiple `EnclaveConfig` objects may exist, an attacker may create a different configuration, increment its version, and utilize it to destroy an unrelated enclave. This breaks configuration lineage and allows unauthorized enclave deletion. Also, outdated enclaves may still be utilized and be deleted by anyone.

>_ *contracts/enclave/sources/enclave.move* RUST

```rust
public fun destroy_old_enclave<T>(e: Enclave<T>, config: &EnclaveConfig<T>) {
    assert!(e.config_version < config.version, EInvalidConfigVersion);
    let Enclave { id, .. } = e;
    id.delete();
}
```

Remediation

Bind each enclave to its creating configuration ID and verify it during destruction. Additionally, restrict enclave destruction to the enclave owner.

Patch

Resolved in 10ee5b7.

05 — General Findings

Here, we present a discussion of general findings identified during our audit. While these findings do not pose an immediate security impact, they represent anti-patterns and may result in security issues in the future.

ID	Description
OS-SST-SUG-00	Recommendation to refactor the code to improve functionality and mitigate potential issues.
OS-SST-SUG-01	Suggestions to ensure adherence to coding best practices for better clarity and maintainability.
OS-SST-SUG-02	The codebase may be optimized by removing redundant logic, avoiding the emission of unnecessary events, and reclaiming available storage rebates.

Code Refactoring OS-SST-SUG-00

Description

1. Currently within `sentinel`, there are no length checks for the `prompt` and `system_prompt` parameters. This may result in arbitrarily large `prompt` and `system_prompt`, enabling gas griefing, oversized event emissions, and excessive payloads. Thus, it will be appropriate to add length validation for `prompt` in `consume_prompt`, and for `system_prompt` in `register_agent` and `update_agent_prompt`.
2. `sentinel::request_attack` utilizes `tx_context::epoch` which yields a low-entropy nonce that is shared across all `attacks` within the same epoch. Since the epoch length is 1 day, switching to `sui::random` will produce a high-entropy nonce, ensuring uniqueness, thereby improving security.

   >_ contracts/app/sources/sentinel.move RUST

   ```
   public fun request_attack([...]): Attack {
       [...]
       // Generate nonce using TxContext epoch for uniqueness
       let nonce = tx_context::epoch(ctx);
       [...]
   }
   ```

3. Currently in `sentinel::consume_prompt`, `PromptConsumed` derives success indirectly from `(score > 95 || success)` condition, which may diverge from the enclave's explicit verdict. Utilize the `success` value that is passed in as an argument to `consume_prompt`.

Remediation

Incorporate the above refactors.

Patch

1. #1 resolved in 432b856.
2. #2 resolved in e8165ef.
3. #3 resolved in ae5d5ad and 432b856.

Code Maturity OS-SST-SUG-01

Description

1. The function name for `enclave::deploy_old_enclave_by_owner` implies version-based safety checks, but it allows the owner to delete the enclave unconditionally without verifying whether it is outdated. Thus, it may be renamed to `destroy_enclave_by_owner` for clarity.

>_ contracts/enclave/sources/enclave.move RUST

```
public fun deploy_old_enclave_by_owner<T>(e: Enclave<T>, ctx: &mut TxContext) {
    assert!(e.owner == ctx.sender(), EInvalidOwner);
    let Enclave { id, .. } = e;
    id.delete();
}
```

2. Utilize a configurable score threshold instead of hardcoding it to 95 in `sentinel::consume_prompt` to improve maintainability.

>_ contracts/app/sources/sentinel.move RUST

```
public fun consume_prompt<T>([...]) {
    [...]
    if (score > 95 || success) {
        let agent_balance = balance::value(&agent.balance);
        [...]
    }
    [...]
}
```

3. Utilize an `AgentCap` to centralize authorization around a capability object rather than utilizing `Attack.admin` field.
4. Add `!is_withdrawal_unlocked` in `request_attack` to prevent a useless request.

Remediation

Implement the above-mentioned suggestions.

Patch

1. #1 resolved in 10ee5b7.
2. #2 resolved in ae5d5ad.
3. #3 resolved in 72e7e2d.
4. #4 resolved in 4b53f75.

Code Optimization OS-SST-SUG-02

Description

1. In `sentinel::withdraw_from_agent`, the logic for checking if 14 days have passed since last funding may be replaced with an assertion check on `is_withdrawal_unlocked` to avoid unnecessary code duplication.

>_ *contracts/app/sources/sentinel.move* RUST

```rust
/// Withdraw funds from agent (only by creator, enforces 14-day lock from last funding)
public fun withdraw_from_agent(
    agent: &mut Agent,
    amount: u64,
    clock: &Clock,
    ctx: &mut TxContext
): Coin<SUI> {
    assert!(agent.creator == ctx.sender(), ENotAuthorized);
    assert!(balance::value(&agent.balance) >= amount, EInsufficientBalance);

    // Check if 14 days have passed since last funding
    let current_time = clock::timestamp_ms(clock);
    let time_since_last_funding = current_time - agent.last_funded_timestamp;
    assert!(
        time_since_last_funding >= WITHDRAWAL_LOCK_PERIOD_MS,
        EWithdrawalLocked
    );
    [...]
}
```

2. After `sentinel::consume_prompt` completes, the `Attack` object has no further purpose and should be destroyed instead of just marking it as utilized. Deleting it will reclaim the Sui storage rebate and prevent unnecessary on-chain state growth.
3. Setter functions should first compare the new value against the current value before applying updates. This avoids emitting meaningless events when no state change occurs.
4. Simplify the logic by storing only creator fee and protocol fee, and automatically sending the remaining amount to the agent, instead of storing all three values and performing additional remainder calculations.

Remediation

Modify the codebase to include the above optimizations.

Patch

1. #1 resolved in 432b856.
2. #2 resolved in 432b856.
3. #3 resolved in 0a667ff.
4. #4 resolved in 550dc2c.

A — Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the General Findings.

CRITICAL

Vulnerabilities that immediately result in a loss of user funds with minimal preconditions.

Examples:

- Misconfigured authority or access control validation.
- Improperly designed economic incentives leading to loss of funds.

HIGH

Vulnerabilities that may result in a loss of user funds but are potentially difficult to exploit.

Examples:

- Loss of funds requiring specific victim interactions.
- Exploitation involving high capital requirement with respect to payout.

MEDIUM

Vulnerabilities that may result in denial of service scenarios or degraded usability.

Examples:

- Computational limit exhaustion through malicious input.
- Forced exceptions in the normal user flow.

LOW

Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk.

Examples:

- Oracle manipulation with large capital requirements and multiple transactions.

INFO

Best practices to mitigate future security risks. These are classified as general findings.

Examples:

- Explicit assertion of critical internal invariants.
- Improved input validation.

B — Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that others may have missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.