OtterSec

Fluid Timelock Integration

Security Assessment

Prepared for
Fluid

Prepared by
Otter Audits, LLC

Prepared on
November 6th, 2026

Michał Bochnak embe221ed@osec.io
Thiago Tavares thitav@osec.io
Bartłomiej Wierzbiński dark@osec.io

Table of Contents

1. Executive summary 2
1.1. Overview 2
1.2. Key findings 2
2. Scope 3
3. Findings 4
3.1. Findings summary 4
4. Advisories 5
OS-FLT-ADV-00 Timelock controls lack version gates 6
OS-FLT-ADV-01 Stale markers revoke future guardians 7
OS-FLT-ADV-02 Conflicting roles let guardians block removal 9
OS-FLT-ADV-03 Prescheduled releases bypass custody delays 10
5. Suggestions 11
OS-FLT-SUG-00 Missing oracle timelock lifecycle events 12
OS-FLT-SUG-01 Unnecessarily public address getter function 13
OS-FLT-SUG-02 Guardian removal retains administrator powers 14
OS-FLT-SUG-03 Prescheduled unpauses bypass fresh delays 16
OS-FLT-SUG-04 Stale approvals stop successor campaigns 17
OS-FLT-SUG-05 Frozen TimelockCap makes tickets publicly available 19
OS-FLT-SUG-06 Incomplete capability custody and role revocation 20
OS-FLT-SUG-07 Code clarity 22
A. Vulnerability rating scale 24
B. Procedure 25

© 2026 Otter Audits LLC. All Rights Reserved.

1. Executive summary

1.1. Overview

Fluid engaged OtterSec to conduct a security assessment of the `timelock` program. This assessment was conducted between October 23rd and November 2nd, 2026. For information on our auditing methodology, refer to Appendix B.

1.2. Key findings

We produced 12 findings throughout this audit engagement.

In particular, we identified that the oracle's functions for enabling and disabling the timelock do not enforce the active package version, letting administrators change timelock enforcement through outdated package versions (● OS-FLT-ADV-00). Also, removing a guardian who does not currently hold the guardian role leaves behind a pending removal that never expires and cannot be cancelled, allowing it to later be used to revoke authority granted to that address in a future, legitimate appointment (● OS-FLT-ADV-01).

Additionally, the publisher is initially granted both guardian and admin-canceller authority, and role finalization does not reject this overlap, allowing a malicious holder to use its canceller privileges to block its own removal as guardian (● OS-FLT-ADV-02). Furthermore, capability release proposals may be scheduled before the capability is placed into governance custody, so once the delay elapses, a later deposit can be released to the pre-approved recipient immediately, effectively bypassing the intended custody delay (● OS-FLT-ADV-03).

We also advised emitting dedicated events whenever the oracle's timelock is enabled or disabled to support consistent governance monitoring (● OS-FLT-SUG-00) and restricting the address getter to package-only access while keeping the boolean getter publicly accessible (● OS-FLT-SUG-01). Moreover, we recommended ensuring guardian removal also clears administrator privileges, as a removed guardian holding administrator authority retains the ability to pause instantly (● OS-FLT-SUG-02).

Furthermore, we suggested that new timelock capabilities go straight into protocol custody (● OS-FLT-SUG-05), and highlighted gaps in how capabilities are held and roles are removed, which let privileged parties get around governance delays or block their own removal (● OS-FLT-SUG-06). Finally, we recommended clearer names for the authorization helpers, accurate comments on migration, and unique vault error codes (● OS-FLT-SUG-07).

© 2026 Otter Audits LLC. All Rights Reserved.

2. Scope

The `timelock` update in PR#38 adds governance timelocks across Fluid's five packages, enforcing minimum delays of 12 hours for admin actions and 6 hours for configuration changes. Protected calls require both existing role authorization and a capability-backed ticket issued through scheduled execution. The new fluid-timelock package manages proposals, execution, cancellation, role changes, and upgrades. Emergency pauses and migrations remain immediate, while unpausing and disabling timelocks require delayed approval.

The source code was delivered to us in a Git repository at `https://github.com/Instadapp/fluid-packages-sui`.

This audit was performed against PR#38.

© 2026 Otter Audits LLC. All Rights Reserved.

3. Findings

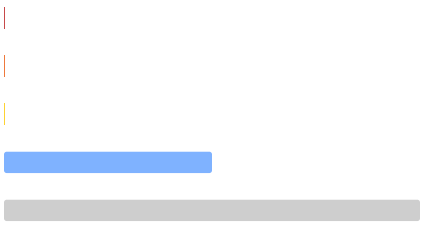
Findings were rated using the criteria in Appendix A.

We split the findings into **advisories** and **suggestions**. Advisories have an immediate impact and should be remediated as soon as possible. Suggestions do not have an immediate impact but will aid in mitigating future vulnerabilities.

Items are tracked with a stable identifier: `OS-FLT-ADV-*` for advisories and `OS-FLT-SUG-*` for suggestions.

3.1. Findings summary

Severity	Count
Critical	0
High	0
Medium	0
Low	4
Informational	8



© 2026 Otter Audits LLC. All Rights Reserved.

4. Advisories

Here, we present a technical analysis of the 4 vulnerabilities we identified during our audit. These vulnerabilities have **immediate** security implications, and we recommend remediation as soon as possible.

ID	Severity	Status	Description
OS-FLT-ADV-00	Low	✓ Resolved	Timelock controls lack version gates
OS-FLT-ADV-01	Low	○ Ack'd	Stale markers revoke future guardians
OS-FLT-ADV-02	Low	○ Ack'd	Conflicting roles let guardians block removal
OS-FLT-ADV-03	Low	○ Ack'd	Prescheduled releases bypass custody delays

© 2026 Otter Audits LLC. All Rights Reserved.

Timelock controls lack version gates

ID	Severity	Status
OS-FLT-ADV-00	Low	✓ Resolved

Description

The oracle's `admin::mint_timelock_cap` and `admin::disable_timelock` functions modify authorization state without calling `registry.assert_version()`, unlike other administrative operations. Consequently, authorized administrators can invoke these functions through outdated package versions, bypassing the active-version restriction when enabling or disabling ticket enforcement.

packages/oracle/sources/module/admin.move

```
public fun mint_timelock_cap(auth: &mut Auth, ctx: &mut TxContext): TimelockCap
{
    auth.assert_has_direct<AdminRole>(ctx);
    auth.enable_timelock(ctx)
}

public fun disable_timelock(auth: &mut Auth, cap: TimelockCap, ctx: &TxContext)
{
    auth.assert_has_direct<AdminRole>(ctx);
    auth.disable_timelock(cap);
}
```

Remediation

Require both functions to accept the oracle registry and call `registry.assert_version()` before modifying timelock state.

Patch

Resolved in 62b7aec.

© 2026 Otter Audits LLC. All Rights Reserved.

Stale markers revoke future guardians

ID	Severity	Status
OS-FLT-ADV-01	Low	Ack'd

Description

In `fluid_timelock::guardian_gov`, `execute_remove_guardian` unconditionally creates a persistent `PendingGuardianRemoval` marker, even when the target does not hold `GuardianRole`. The marker neither expires nor supports cancellation, leaving removal authorization outstanding indefinitely. If the target later becomes a liquidity guardian without holding governance `GuardianRole`, an executor can consume the stale marker through `complete_remove_guardian` to revoke the newly granted authority.

packages/timelock/sources/module/guardian_gov.move

```
public fun execute_remove_guardian(
    caps: &CapStore,
    tl: &mut Timelock,
    tl_auth: &mut Auth,
    id: vector<u8>,
    clock: &Clock,
    ctx: &mut TxContext,
) {
    let ticket = timelock_ops::execute_admin<RemoveGuardian,
    RemoveGuardianParams>(
        caps,
        tl,
        tl_auth,
        id,
        clock,
        ctx,
    );
    let (_, RemoveGuardianParams(guardian)) = tl.consume(ticket, RemoveGuardian
    {});

    if (tl_auth.has_role<GuardianRole>(guardian)) {
        tl_auth.revoke_guardian_role<GuardianRole>(guardian);
        if (tl_auth.has_role<AuthCancellerRole>(guardian)) {
            tl_auth.revoke_guardian_role<AuthCancellerRole>(guardian)
        };
    };
    tl_auth.add_pending_guardian_removal(guardian);
    events::emit_guardian_removal_pending(guardian);
}
```

© 2026 Otter Audits LLC. All Rights Reserved.

Remediation

Bind pending removals to a specific guardian appointment and invalidate them when a new appointment supersedes that authorization.

Patch

Fluid acknowledged this issue.

© 2026 Otter Audits LLC. All Rights Reserved.

Conflicting roles let guardians block removal

ID	Severity	Status
OS-FLT-ADV-02	Low	Ack'd

Description

`fluid_timelock::auth::init` grants both `GuardianRole` and `AdminCancellerRole` to the publisher, and `finalize_roles` does not reject this conflicting membership. Although the guardian-specific cancellation path blocks cancellation of `RemoveGuardian`, the holder can use its admin-canceller authority to cancel that operation directly. This allows a malicious holder to obstruct its own removal.

Remediation

Enforce mutual exclusivity during role assignment and finalization.

Patch

Fluid acknowledged this issue stating that these roles cannot be assigned to the same address after deployment.

© 2026 Otter Audits LLC. All Rights Reserved.

Prescheduled releases bypass custody delays

ID	Severity	Status
OS-FLT-ADV-03	Low	Ack'd

Description

The per layer `schedule_disable_timelock` functions do not verify that the corresponding `TimelockCap` is currently held in `CapStore`, nor do they bind the operation to a specific capability ID or custody generation. An authorized proposer can therefore schedule a disable operation before the layer's timelock is enabled.

The 12-hour delay can elapse while no capability is held. If the layer is subsequently enabled during the operation's execution window, an authorized executor can immediately execute the previously approved operation. Execution removes and burns the newly installed capability, disabling the layer.

If setup has already been completed, this is particularly consequential because the consumed `SetupCap` prevents the layer from being re-enabled through the current implementation. This is another instance of stale approvals remaining executable across changes to the state underlying their authorization, similar to pre-scheduled unpause operations.

Remediation

Record the expected capability ID or deposit generation and require an exact match at execution. Also require that the relevant capability is currently held.

Patch

Fluid acknowledged this issue.

© 2026 Otter Audits LLC. All Rights Reserved.

5. Suggestions

Here, we present a discussion of the 8 suggestions we identified during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and may result in security issues in the future.

ID	Severity	Status	Description
OS-FLT-SUG-00	Informational	✓ Resolved	Missing oracle timelock lifecycle events
OS-FLT-SUG-01	Informational	✓ Resolved	Unnecessarily public address getter function
OS-FLT-SUG-02	Informational	○ Ack'd	Guardian removal retains administrator powers
OS-FLT-SUG-03	Informational	○ Ack'd	Prescheduled unpauses bypass fresh delays
OS-FLT-SUG-04	Informational	○ Ack'd	Stale approvals stop successor campaigns
OS-FLT-SUG-05	Informational	○ Ack'd	Frozen TimelockCap makes tickets publicly available
OS-FLT-SUG-06	Informational	○ Ack'd	Incomplete capability custody and role revocation
OS-FLT-SUG-07	Informational	✓ Resolved	Code clarity

© 2026 Otter Audits LLC. All Rights Reserved.

Missing oracle timelock lifecycle events

ID	Severity
OS-FLT-SUG-00	Informational

Description

Oracle's timelock lifecycle functions do not emit `LogTimelockEnabled` or `LogTimelockDisabled`, unlike the other four protocol packages. This omission prevents event-based indexers and monitoring systems from directly detecting changes to Oracle's timelock enforcement.

Remediation

Emit corresponding package-specific events when enabling or disabling the timelock to provide consistent governance observability.

Patch

Resolved in e516792.

© 2026 Otter Audits LLC. All Rights Reserved.

Unnecessarily public address getter function

ID	Severity
OS-FLT-SUG-01	Informational

Description

In `fluid_liquidity::structs`, both `get_address_bool_value` and `get_address_bool_addr` are publicly exposed, even though it is only necessary for `get_address_bool_value` to be `public` as `get_address_bool_addr` used only within `fluid_liquidity`. This unnecessarily expands the public API and creates an avoidable compatibility obligation, without providing additional functionality.

packages/liquidity/sources/types/structs.move

```
1  public fun get_address_bool_addr(self: &AddressBool): address {
2      self.addr
3  }
4
5  public fun get_address_bool_value(self: &AddressBool): bool {
6      self.value
7  }
```

Remediation

Restrict `get_address_bool_addr` to `public(package)` while retaining `public` visibility for `get_address_bool_value`.

Patch

Resolved in 3977e39.

© 2026 Otter Audits LLC. All Rights Reserved.

Guardian removal retains administrator powers

ID	Severity
OS-FLT-SUG-02	Informational

Description

`RemoveGuardian` revokes only the explicit Liquidity guardian membership through `fluid_timelock::guardian_gov::complete_remove_guardian` calling `update_guardians(false)`.

packages/timelock/sources/module/guardian_gov.move

```
public fun complete_remove_guardian(
    caps: &CapStore,
    tl_auth: &mut Auth,
    liquidity_auth: &mut LiquidityAuth,
    liquidity_registry: &Registry,
    guardian: address,
    ctx: &TxContext,
) {
    [...]
    liquidity_admin::update_guardians(
        liquidity_auth,
        liquidity_registry,
        liq_ticket,
        vector[structs::new_address_bool(guardian, false)],
        ctx,
    );
    events::emit_guardian_removal_completed(guardian);
}
```

However, `fluid_liquidity::auth::get_is_guardian` also recognizes Liquidity administrators, so a target retaining `AdminRole` continues to qualify as a guardian.

packages/liquidity/sources/state/auth.move

```
/// True iff `addr` holds an explicit `GuardianRole` record.
public fun get_is_guardian_record(self: &Auth, addr: address): bool {
    self.roles.contains(RoleKey<GuardianRole> { owner: addr })
}
```

Consequently, guardian removal can report completion while the address retains instant pause authority through its administrator privileges. Fully revoking this authority requires a separate governance action to revoke `AdminRole`, as deleting the guardian record alone is insufficient.

© 2026 Otter Audits LLC. All Rights Reserved.

Remediation

Require removal to verify that effective guardian authority is cleared, including separately revoking any retained Liquidity administrator role.

Patch

Fluid acknowledged this suggestion.

© 2026 Otter Audits LLC. All Rights Reserved.

Prescheduled unpauses bypass fresh delays

ID	Severity
OS-FLT-SUG-03	Informational

Description

`fluid_timelock::liquidity_gov::schedule_unpause_global` does not inspect pause state, allowing an admin proposer to queue an unpause while the relevant scope is healthy. After its 12-hour delay elapses, the approval remains executable throughout the seven-day grace window without being bound to a particular pause occurrence. If a guardian subsequently calls `fluid_liquidity::admin::pause_global` to pause that scope, an executor who also holds Liquidity `AdminRole` can immediately execute the matured unpause via `execute_unpause_global`. This lifts a newly imposed emergency restriction without waiting for a fresh delay after the guardian's intervention. The same pattern affects global, token, and user unpauses, allowing pre-scheduled approvals to weaken the intended emergency-response window.

packages/timelock/sources/module/liquidity_gov.move

```
public fun schedule_unpause_global(
    caps: &CapStore,
    tl: &mut Timelock,
    tl_auth: &Auth,
    action: PauseAction,
    salt: vector<u8>,
    clock: &Clock,
    ctx: &mut TxContext,
): vector<u8> {
    timelock_ops::schedule_admin<UnpauseGlobal, _>(
        caps,
        tl,
        tl_auth,
        UnpauseGlobalParams(action),
        salt,
        clock,
        ctx,
    )
}
```

Remediation

Bind each unpause proposal to the scope's current paused state and pause nonce, invalidating it whenever a new pause occurs.

Patch

Fluid acknowledged this suggestion.

© 2026 Otter Audits LLC. All Rights Reserved.

Stale approvals stop successor campaigns

ID	Severity
OS-FLT-SUG-04	Informational

Description

`fluid_timelock::rewards_gov::schedule_stop_rewards` stores only the manager `ID` in `StopRewardsParams`, without identifying the campaign approved for termination. A stop proposal scheduled for campaign `A` can remain pending while campaign `B` is queued and promoted through permissionless `fluid_lending_rewards::admin::transition_to_next_rewards`.

packages/timelock/sources/module/rewards_gov.move

```
/// Schedules `StopRewards<T>` on the auth timelock (6h).
public fun schedule_stop_rewards(
    caps: &CapStore,
    tl: &mut Timelock,
    tl_auth: &Auth,
    manager_id: ID,
    salt: vector<u8>,
    clock: &Clock,
    ctx: &mut TxContext,
): vector<u8> {
    timelock_ops::schedule_auth<StopRewards, _>(
        caps,
        tl,
        tl_auth,
        StopRewardsParams(manager_id),
        salt,
        clock,
        ctx,
    )
}
```

While the proposal remains executable, `fluid_timelock::rewards_gov::execute_stop_rewards` forwards the stale approval to `fluid_lending_rewards::admin::stop_rewards`, which operates on the current schedule. Although `rewards_calcs::assert_stoppable` rejects a live queued window, it accepts `B` after promotion to the current campaign. An authorized executor can therefore terminate `B` without a stop proposal specifically targeting it, and without an externally held `RewardsWriterCap`.

Remediation

Bind each stop proposal to the intended campaign identifier or schedule version and reject execution if it no longer matches.

© 2026 Otter Audits LLC. All Rights Reserved.

Patch

Fluid acknowledged this suggestion.

© 2026 Otter Audits LLC. All Rights Reserved.

Frozen TimelockCap makes tickets publicly available

ID	Severity
OS-FLT-SUG-05	Informational

Description

Each protocol exposes a public `mint_timelock_cap` function that records a newly created `TimelockCap` and returns it to the calling top-tier administrator. Because `TimelockCap` has `key` and `store`, the caller can invoke `public_freeze_object` instead of depositing it into the canonical `CapStore`.

The frozen capability becomes an immutable public object. Anyone can then pass `&TimelockCap` to the protocol's public `timelock_ticket` function and obtain a valid ticket for the recorded capability. Protocol operations still verify the transaction sender's native role, so this does not grant privileges to arbitrary users. However, every account already holding an applicable administrator or configuration role can obtain the public ticket and call protected operations directly, bypassing the intended timelock delay.

Freezing also prevents the capability from being moved into `CapStore` or consumed by `disable_timelock`. Because the protocol continues recording its ID, a replacement capability cannot be minted. The protocol is therefore left with a permanently public timelock factor and cannot restore canonical custody through the normal lifecycle.

Remediation

Make capability creation and canonical storage inseparable and raw minting not publicly return the capability. Also, require `&mut TimelockCap` in `timelock_ticket`.

Patch

Fluid acknowledged this suggestion.

© 2026 Otter Audits LLC. All Rights Reserved.

Incomplete capability custody and role revocation

ID	Severity
OS-FLT-SUG-06	Informational

Description

1. `fluid_timelock::role_gov::complete_setup` verifies custody of protocol timelock capabilities but not package upgrade capabilities. Consequently, externally retained upgrade capabilities may allow package upgrades outside timelocked governance even after setup completes.

packages/timelock/sources/module/role_gov.move

```
public fun complete_setup(auth: &mut Auth, caps: &CapStore, setup:
SetupCap) {
  assert_all_timelocks_enabled(caps);
  auth.complete_setup(setup);
}
```

2. `fluid_timelock::role_gov::schedule_grant_role<GuardianRole>` does not reject existing role holders, allowing a guardian with `AdminProposerRole` to pre-schedule grants to itself. If a queued grant remains executable after removal, it can restore the guardian's authority, undermining the removal unless outstanding grants are cancelled or invalidated.

packages/timelock/sources/module/role_gov.move

```
public fun schedule_grant_role<R>(
    caps: &CapStore,
    tl: &mut Timelock,
    auth: &Auth,
    addr: address,
    salt: vector<u8>,
    clock: &Clock,
    ctx: &mut TxContext,
): vector<u8> {
    auth::assert_is_role<R>();
    assert_no_role_conflict<R>(auth, addr);
    timelock_ops::schedule_admin<GrantRole<R>, _>(
        caps,
        tl,
        auth,
        GrantRoleParams(addr),
        salt,
        clock,
        ctx,
    )
}
```

© 2026 Otter Audits LLC. All Rights Reserved.

3. A non-guardian `AdminCancellerRole` holder can cancel its own scheduled revocation through `fluid_timelock::role_gov::cancel<AdminCancellerRole, RevokeRoleParams>`. Because cancellation does not exclude operations revoking the caller's authority, the holder can repeatedly veto attempts to remove it. Additional honest cancellers cannot override this veto, allowing a malicious holder to obstruct its own removal and retain cancellation powers.

   packages/timelock/sources/module/role_gov.move

```
public fun cancel<Role, Params: store + drop>(
    caps: &CapStore,
    tl: &mut Timelock,
    auth: &Auth,
    id: vector<u8>,
    ctx: &mut TxContext,
) {
    caps.assert_version();
    tl.cancel<Role, Params>(auth, id, ctx);
}
```

4. `fluid_lending::admin::set_rewards_schedule` authorizes changes through the current `RewardsWriterCap`, without independently enforcing the rewards manager's delay. An external holder of that capability can therefore update or clear rewards schedules directly, bypassing the manager's delayed execution path. This bypass is prevented only while the currently recognized writer capability remains sealed inside the manager.

Remediation

1. Require setup completion to verify that all required package upgrade capabilities are held in the capability store.
2. Reject grants to existing role holders and invalidate pending guardian grants when the target is removed.
3. Prevent admin cancellers from vetoing their own revocation by enforcing a non-cancellable removal path.
4. Require the current rewards-writer capability to be sealed in the manager.

Patch

Fluid acknowledged these suggestions.

© 2026 Otter Audits LLC. All Rights Reserved.

Code clarity

ID
OS-FLT-SUG-07

Severity
Informational

Description

1. Authorization helper names in `auth.move` modules do not clearly distinguish their enforcement scope. For example, in `fluid_lending::auth`, `assert_role` checks both registry version and role membership, whereas `assert_has_role` checks only membership. This similarity makes it easy to select the wrong helper. Rename helpers consistently to precisely state their application.

packages/lending/sources/state/auth.move

```
public(package) fun assert_has_role<C: store + drop>(self: &Auth, ctx:
&TxContext) {
    assert!(self.has_role<C>(ctx.sender()),
    errors::f_token_unauthorized());
}

public(package) fun assert_role<C: store + drop>(
    lending_registry: &LendingRegistry,
    self: &Auth,
    ctx: &TxContext,
) {
    lending_registry.assert_version();
    self.assert_has_role<C>(ctx);
}
```

2. The comments on `migrate` in `fluid_liquidity::registry` and `fluid_vaults::registry` incorrectly describe their administrative wrappers as permissionless. Migration is role-gated, so these comments misrepresent who can advance the registry version and may mislead deployment or upgrade procedures. Update both comments to reflect the actual authorization requirements without changing the migration logic.

packages/liquidity/sources/state/registry.move

```
/// Bumps the object to the current package version after a package
upgrade.
/// Aborts if already current.
/// Permissionless at the `admin.move` entry point: only moves `version`
forward.
public(package) fun migrate(self: &mut Registry) {
    assert!(self.version < constants::version(),
    errors::already_migrated());
    let old_version = self.version;
    self.version = constants::version();
    events::emit_log_migrate(old_version, self.version);
}
```

© 2026 Otter Audits LLC. All Rights Reserved.

packages/vaults/sources/state/registry.move

```
public(package) fun migrate(self: &mut VaultRegistry) {
    assert!(self.version < constants::version(),
    errors::vault_already_migrated());
    let old_version = self.version;
    self.version = constants::version();
    events::emit_log_migrate(old_version, self.version);
}
```

3. The vault's error definitions introduces conflicting error codes. `vault_admin_only_migrators()` and `vault_price_quote_required()` share error code `6078`, while `vault_admin_timelock_ticket_required()` and `vault_same_col_debt_type()` both use `6079`. These collisions make numeric-code-based decoding ambiguous, potentially resulting in misidentification of failures. Assign unique codes to each error.

Remediation

Incorporate the above-mentioned suggestions.

Patch

1. Fluid acknowledged this suggestion.
2. Resolved in e4ba0dc.
3. Resolved in b0d11a9.

© 2026 Otter Audits LLC. All Rights Reserved.

Appendix 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.

Severity	Description
Critical	Vulnerabilities that immediately result in a loss of availability, consensus divergence, or exploitable memory corruption with minimal preconditions.
High	Vulnerabilities that may result in consensus divergence or denial of service but require specific transaction or network conditions to exploit.
Medium	Vulnerabilities that may result in incorrect behavior or state divergence under specific configurations or edge cases.
Low	Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk.
Informational	Best practices to mitigate future security risks. These are classified as general findings.

Resolution status is one of ✓ Resolved, ○ Ack'd, or × To-do.

© 2026 Otter Audits LLC. All Rights Reserved.

Appendix B.

Procedure

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

The design review asks whether the program's architecture and operating assumptions support its intended behavior. We study the inputs it accepts, the decisions it makes, the information shared between components, and the services or participants it trusts. We then consider how it behaves when those inputs are unexpected or conflicting. The goal is to find conditions that can produce an incorrect decision, inconsistent state, or loss of service.

Design problems often arise from interactions rather than one defective function. Two components may each process a value correctly but interpret it differently. A failure may also leave one component ready to continue while another treats the same operation as incomplete. Local checks can look correct in both cases, so we follow the complete operation across component boundaries.

The implementation review asks whether the code matches the design. We trace untrusted data from its entry point to the point where it affects behavior, checking the assumptions made along the way. The review covers validation, error handling, state updates, concurrency, persistence, resource use, and cleanup. Its exact focus depends on the target's language, runtime, and architecture rather than a fixed list of bug classes.

The boundary between design and implementation is not exact. Missing validation may be a local coding error, while the same omission across several components may reflect an unclear rule. We classify a finding by its root cause, but judge its impact from the resulting behavior and the conditions needed to trigger it.

We begin by reading the target as a team and agreeing on its main components, trust boundaries, and expected behavior. Auditors then review different paths through the system and compare notes where those paths meet. We reproduce suspected issues with focused tests or traces when the setup allows. Before reporting a finding, another auditor reviews its trigger, impact, and proposed remediation.

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

© 2026 Otter Audits LLC. All Rights Reserved.