Kofi Finance

Security Assessment

May 10th, 2025 — Prepared by OtterSec

Andreas Mantzoutas	andreas@osec.io
Robert Chen	r@osec.io

Table of Contents

Appendices

© 2025 Otter Audits LLC. All Rights Reserved.

01 — Executive Summary

Overview

Kofi engaged OtterSec to assess the `kofi-finance-contracts` program. This assessment was conducted between April 17th and May 5th, 2025. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 11 findings throughout this audit engagement.

In particular, we identified multiple high-risk vulnerabilities affecting staking reward integrity and supply accounting. In the `rewards_manager` module, staking rewards are minted to the `kAPT` vault without updating the `virtual_balance`, causing all rewards (except fees) to become permanently locked, thereby preventing stakers from receiving profits (OS-KOF-ADV-00). Additionally, the `update_rewards` function fails to account for `minting_fees` when computing `total_kapt`, resulting in excess token minting and eventual depegging as the kAPT supply surpasses the actual staked APT (OS-KOF-ADV-01). Moreover, a medium-severity issue arises from rounding errors in APT-to-share conversions within the `delegation_pool`, causing the protocol to absorb small inconsistencies that may accumulate over time (OS-KOF-ADV-02).

We also recommended codebase modifications to eliminate redundant code, avoid hardcoded values, and improve overall efficiency, while ensuring adherence to coding best practices (OS-KOF-SUG-00).

© 2025 Otter Audits LLC. All Rights Reserved.

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/wagmitt/kofi-finance-contracts. This audit was performed against 75ed16e.

A brief description of the program is as follows:

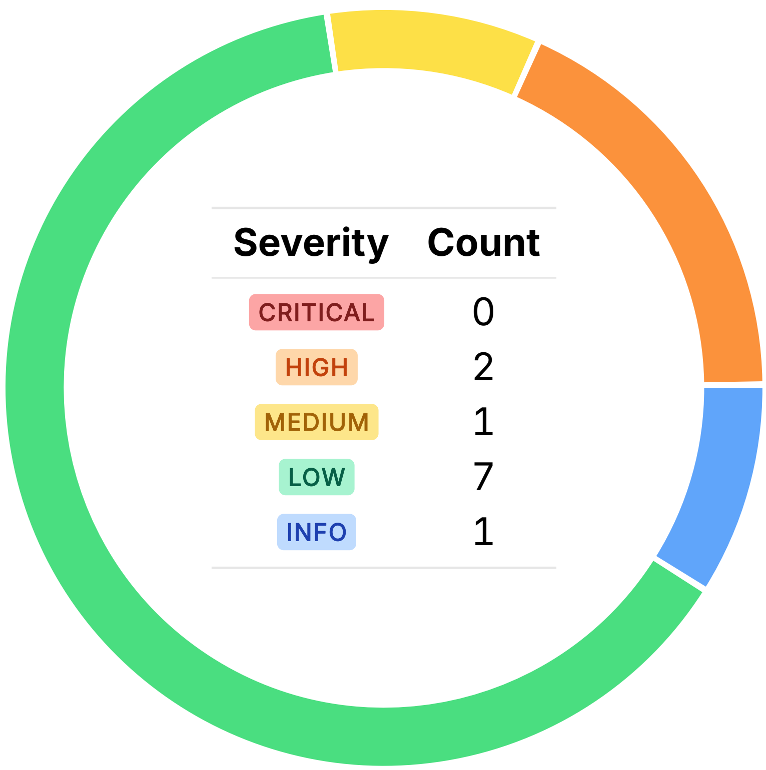
Name	Description
kofi-finance-contracts	A liquid staking protocol for Aptos, allowing users to stake APT tokens and receive liquid staking derivatives (kAPT and stkAPT).

© 2025 Otter Audits LLC. All Rights Reserved.

03 — Findings

Overall, we reported 11 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	0
HIGH	2
MEDIUM	1
LOW	7
INFO	1

© 2025 Otter Audits LLC. All Rights Reserved.

04 — Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we 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.

© 2025 Otter Audits LLC. All Rights Reserved.

ID	Severity	Status	Description
OS-KOF-ADV-00	HIGH	RESOLVED ⊘	`rewards_manager::update_rewards` fails to update the `virtual_balance`, causing staking rewards to become locked.
OS-KOF-ADV-01	HIGH	RESOLVED ⊘	`update_rewards` miscalculates staking rewards by not accounting for minting fees, leading to double-minting of `kAPT`.
OS-KOF-ADV-02	MEDIUM	RESOLVED ⊘	Rounding errors in `delegation_pool` operations may create discrepancies in stake amounts during delegation and undelegation, resulting in potential value loss for the pool.
OS-KOF-ADV-03	LOW	RESOLVED ⊘	A strict assertion in `update_rewards` causes staking operations to fail during temporary imbalances between total `kAPT` and total staked APT.
OS-KOF-ADV-04	LOW	RESOLVED ⊘	The `math::ratio` function applies inconsistent scaling in its conversion logic, leading to potential imbalances between `kAPT` and `stkAPT`.
OS-KOF-ADV-05	LOW	RESOLVED ⊘	The `ratio <= RATIO_MAX` assertion in `math::ratio` can permanently lock protocol liquidity if the ratio naturally grows beyond the limit.
OS-KOF-ADV-06	LOW	RESOLVED ⊘	`math::from_shares` assigns a minimum of 1 share, even if the calculated amount is 0, leading to protocol overpayment during unstaking.
OS-KOF-ADV-07	LOW	RESOLVED ⊘	Removing validators may result in irretrievable stake, risking temporary protocol insolvency.
OS-KOF-ADV-08	LOW	RESOLVED ⊘	Unaccounted staking fees during delegation cause the buffer vault to burn more than it mints, gradually depleting its balance and risking protocol stability.
OS-KOF-ADV-09	LOW	RESOLVED ⊘	`delegation_manager::withdraw_stake` assumes that `withdrawn_amount` is always greater than the minimum threshold, risking unexpected aborts if this condition is not met.

© 2025 Otter Audits LLC. All Rights Reserved.

Failure to Distribute Staking Rewards HIGH OS-KOF-ADV-00

Description

`rewards_manager::update_rewards` handles staking reward calculations and distributions during epoch changes. A manager fee is deducted, and the remaining APT rewards are minted as `kAPT` and deposited into the vault via the `minting_manager::mint_to_vault` function.

>_ *rewards_manager.move* RUST

```rust
public fun update_rewards() acquires RewardState {
    ...
    // Mint remaining rewards (after fee) to vault
    let rewards_after_fee = (new_rewards as u128) - (fee_amount as u128);
    mint_to_vault(rewards_after_fee as u64);
    ...
}
...
fun mint_to_vault(amount: u64) {
    let vault_address = vault::get_kapt_vault_address();
    kAPT_coin::mint(vault_address, amount);
}
```

However, while `kAPT` coins are successfully minted, the `virtual_balance`, which tracks deposited `kAPT` and determines the exchange rate, is not updated. This oversight permanently locks staking rewards, preventing distribution to stakers.

Remediation

Modify `mint_to_vault` to also update `virtual_balance` when new rewards are issued to the vault.

Patch

Fixed in e9c9c89.

© 2025 Otter Audits LLC. All Rights Reserved.

kAPT Double Minting HIGH OS-KOF-ADV-01

Description

`rewards_manager::update_rewards` determines new staking rewards by comparing the total active staked APT, retrieved from the delegation pools, against the circulating `kAPT` supply and collected management fees. Any amount not yet minted as `kAPT` and not part of the fees is considered staking rewards, and the corresponding `kAPT` is issued to the vault.

>_ rewards_manager.move RUST

```rust
public fun update_rewards() acquires RewardState {
    ...
    // Get total staked APT and kAPT supply
    let total_staked = get_staked_apt();
    let total_kapt = (kAPT_coin::total_supply() as u128)
        + (state.collected_fees as u128);
    ...
    let new_rewards = (total_staked as u128) - total_kapt;
    ...
    // Mint remaining rewards (after fee) to vault
    let rewards_after_fee = (new_rewards as u128) - (fee_amount as u128);
    mint_to_vault(rewards_after_fee as u64);
}
```

However, this calculation does not account for minting fees. When stake is added to a delegation pool, an `add_stake` fee is deducted if the validator being delegated to is producing rewards for that epoch. This fee is temporarily subtracted from the delegator's active stake and is refunded in the next epoch. The protocol tracks this fee separately and allows the admin to collect it asynchronously. Despite this, the staked APTs are still marked as rewards by the `update_rewards` function, causing it to mint `kAPT` on their behalf. Later, when the admin collects these fees, `kAPT` is re-minted for the same amount, resulting in double-minting and an immediate depegging of `kAPT`.

Remediation

Update `update_rewards` to include the minting fees when calculating the total `kAPT` supply.

Patch

Fixed in 74d73ff.

© 2025 Otter Audits LLC. All Rights Reserved.

Rounding Error in Delegation Pool MEDIUM OS-KOF-ADV-02

Description

The majority of `delegation_pool` operations contain small rounding errors that affect delegators. When unlocking stake (undelegating) from a delegation pool, the amount unlocked may be slightly less than the requested amount. Similarly, during staking, users deposit a specific amount of `APT` in exchange for a calculated number of shares, but due to rounding during the conversion, the actual stake increase may be slightly less than the input amount. For example, a user may delegate `x APT`, but only `x-1 APT` is effectively staked.

It is essential that this rounding error is covered by the user and not the protocol, otherwise, the pool gradually loses value. Currently, neither delegation nor undelegation accounts for this error. When adding stake, the protocol mints to the caller the exact amount it attempts to delegate as `kAPT`, which does not reflect the real staked increase, resulting in the minting of more `kAPT` than the staked `APT`. Similarly, during undelegation, users withdraw the full requested amount, even if the actual decrease in the pending inactive stake is less, resulting in the pool losing value. This imbalance may even create a scenario where the total requested withdrawals exceed the available pending inactive and inactive stake, rendering full withdrawals impossible for some users.

Remediation

Compute the real stake changes by calling `get_stake` before and after each `delegation_pool` operation, and act only based on the actual changes.

Patch

Fixed in eded117 and 3d2bd6d.

© 2025 Otter Audits LLC. All Rights Reserved.

Abort on Temporary Imbalance During Epoch Transition LOW OS-KOF-ADV-03

Description

In `rewards_manager::update_rewards`, an abort occurs when `total_kapt` exceeds `total_staked`. However, this strict check fails to account for a valid edge case involving epoch timing. When the `add_stake_fee` is collected, it is immediately added to `minting_fees` and made available for the admin to withdraw. However, the corresponding stake does not become active until the next epoch. If the admin collects these fees within the same epoch, the `kAPT` supply may temporarily exceed the actual staked `APT`, triggering the abort and locking both staking and unstaking until the next epoch.

>_ *rewards_manager.move* RUST

```rust
public fun update_rewards() acquires RewardState {
    ...
    let total_staked = get_staked_apt();
    let total_kapt = (kAPT_coin::total_supply() as u128)
        + (state.collected_fees as u128);

    assert!((total_staked as u128) >= total_kapt, errors::total_staked_less_than_kapt());
    ...
}
```

Remediation

Replace the strict assertion with an early return when `total_kapt` exceeds `total_staked`, allowing the safe continuation of operations and preventing false failures during temporary imbalances.

Patch

Fixed in 246b120.

© 2025 Otter Audits LLC. All Rights Reserved.

Inconsistent Scaling in Conversion Rate Calculation LOW OS-KOF-ADV-04

Description

The `math::ratio` function is used to calculate the conversion rate between `kAPT` and `stkAPT` by considering the total locked `kAPT` and the total `stkAPT` supply, scaling the result to 8 decimals. To prevent overflows, the implementation uses two different branches depending on whether the locked `kAPT` amount exceeds the `stkAPT` supply.

>_ *math.move* RUST

```rust
public fun ratio(supply: u128, tvl: u128): u256 {
    ...
    if (supply >= tvl) {
        let whole = (supply / tvl) as u256;
        let remainder = (supply % tvl) as u256;
        let ratio = whole * PRECISION + (remainder * PRECISION / (tvl as u256));
        assert!(ratio <= RATIO_MAX, E_RATIO_OVERFLOW);
        ratio
    } else {
        // If supply < tvl, multiply first then divide to maintain precision
        let ratio = (supply as u256) * RATIO_MAX / (tvl as u256);
        assert!(ratio <= RATIO_MAX, E_RATIO_OVERFLOW);
        ratio
    }
}
```

The issue arises because these branches apply different scaling factors. The *if* branch scales the amount to `PRECISION`, while the *else* branch scales it to `RATIO_MAX`, leading to inconsistencies if that path is executed. Additionally, multiplying a `u128` by `PRECISION` will not result in an overflow of `u256`, rendering the branch distinction unnecessary.

Remediation

Merge the two branches into a single unified calculation and scale the result consistently to `PRECISION`.

Patch

Fixed in 5f606e6.

© 2025 Otter Audits LLC. All Rights Reserved.

Unnecessary Assertion Causes Protocol Lockup LOW OS-KOF-ADV-05

Description

In `math::ratio`, the assertion `ratio <= RATIO_MAX` is unnecessary. In an LSD system, the ratio between the underlying asset and the derivative token naturally increases over time and is not expected to decrease. As a result, if the ratio ever reaches `RATIO_MAX`, the assertion will trigger, causing the protocol to halt. This failure would prevent any further staking or unstaking operations, effectively locking all liquidity and breaking the protocol's functionality indefinitely.

>_ *math.move* RUST

```rust
public fun ratio(supply: u128, tvl: u128): u256 {
    ...
    if (supply >= tvl) {
        let whole = (supply / tvl) as u256;
        let remainder = (supply % tvl) as u256;
        let ratio = whole * PRECISION + (remainder * PRECISION / (tvl as u256));
        assert!(ratio <= RATIO_MAX, E_RATIO_OVERFLOW);
        ratio
    } else {
        // If supply < tvl, multiply first then divide to maintain precision
        let ratio = (supply as u256) * RATIO_MAX / (tvl as u256);
        assert!(ratio <= RATIO_MAX, E_RATIO_OVERFLOW);
        ratio
    }
}
```

Remediation

Remove the assertion.

Patch

Fixed in 3cc1271.

© 2025 Otter Audits LLC. All Rights Reserved.

Risk of Overpayment LOW OS-KOF-ADV-06

Description

`math::from_shares` calculates the amount of `kAPT` a user should receive based on their `stkAPT` unstaking amount and the current conversion rate. If the computed share amount is 0, while the unstaking amount is greater than 0, the protocol sets the share amount to 1 to prevent the user from burning tokens without receiving anything in return. However, this logic introduces an overpayment vulnerability, as the user receives more value than they originally unstaked.

>_ *math.move* RUST

```rust
public fun from_shares(ratio: u256, amount: u64): u64 {
    let shares = (amount as u256) * ratio / PRECISION;
    assert!(shares <= (U64_MAX as u256), E_U64_OVERFLOW);
    if (amount > 0 && shares == 0) {
        shares = 1;
    };
    (shares as u64)
}
```

Remediation

Replace the condition with an assertion to prevent users from losing their funds while also avoiding overpayment during the unstaking process.

Patch

Fixed in 63d3a44.

© 2025 Otter Audits LLC. All Rights Reserved.

Protocol Insolvency via Validator Removal LOW OS-KOF-ADV-07

Description

`config::update_validator_config_admin` allows an administrator to update the validator set and their stake allocations. However, it is possible to completely remove certain validators, yet there is no mechanism to manage their existing stakes or safely unstake the full amount when this occurs. As a result, part of the staked `APT` may become irretrievable, risking temporary protocol insolvency as the total `kAPT` supply may exceed the total staked `APT`. This condition is reversible and may be resolved by re-adding the affected validators.

>_ *config.move* RUST

```rust
public(friend) fun update_validator_config_admin(
    admin: &signer, addresses: vector<address>, allocations: vector<u64>
) acquires GlobalConfig {
    ...
    let validator_config = &mut config.validator_config;
    ...
    validator_config.addresses = addresses;
    validator_config.allocations = allocations;
    event::emit(ValidatorConfigUpdated { validator_config: *validator_config });
}
```

Remediation

Restrict the removal of validators and enforce validation checks within `update_validator_config_admin` to allow only additions or updates, preventing unhandled stakes from becoming irretrievable.

Patch

Fixed in 1ff0089.

© 2025 Otter Audits LLC. All Rights Reserved.

Buffer Vault Drainage Due to Unaccounted Staking Fees LOW OS-KOF-ADV-08

Description

The protocol introduces a buffer vault that holds initialization funds to ensure validators maintain the required levels of active and pending inactive stakes, as mandated by protocol parameters. To achieve this, `delegation_manager::ensure_minimum_amounts_from_buffer` enforces that the delegation pool's stakes always match the required amounts. For pending inactive stakes, if the balance falls below the minimum, the function stakes additional `APT` and immediately unstakes it to restore the expected levels.

>_ *delegation_manager.move* RUST

```rust
public(friend) fun ensure_minimum_amounts_from_buffer(
    pool_address: address
) {
    ...
    if (pending_inactive < min_pending_inactive) {
        ...
        delegation_pool::add_stake(&vault_signer, pool_address, amount_needed);
        let add_stake_fee =
            delegation_pool::get_add_stake_fee(pool_address, amount_needed);

        kAPT_coin::mint(
            signer::address_of(&buffer_signer), amount_needed - add_stake_fee
        );

        delegation_pool::unlock(&vault_signer, pool_address, amount_needed);
        kAPT_coin::burn(&buffer_signer, amount_needed);
        ...
    };
}
```

However, during this delegation process, an `add_stake_fee` may be applied. This fee reduces the amount of APT actually minted to the buffer, while the burned kAPT remains equivalent to the originally required amount, `amount_needed`. As a result, the buffer vault burns more than it mints, gradually draining its balance. Eventually, this depletion leads to unexpected aborts, which subsequently block withdrawals from the protocol.

Remediation

Since the staked amount always equals the unstaked amount and the `add_stake_fee` is only temporary for one epoch, remove the kAPT minting and burning operations during the buffer adjustment process.

© 2025 Otter Audits LLC. All Rights Reserved.

Patch

Fixed in c677934.

© 2025 Otter Audits LLC. All Rights Reserved.

Faulty Withdrawal Logic LOW OS-KOF-ADV-09

Description

`delegation_manager::withdraw_stake` is responsible for withdrawing inactive stakes from validators. During this process, the amount unstaked from each validator is split into two parts: the `config::get_min_pending_inactive` portion is sent to the buffer, as it represents `APT` unstaked by the buffer during `ensure_minimum_amounts_from_buffer`, while the remaining amount is sent to the unlocked vault for user withdrawals.

>_ *delegation_manager.move* RUST

```rust
public(friend) fun withdraw_stake() {
    ...
    let len = vector::length(&config::get_validator_addresses());
    while (i < len) {
        ...
        let (_active, withdraw_amount, _pending_inactive) =
            delegation_pool::get_stake(delegation_pool_address, vault_apt_address);

        if (withdraw_amount > 0) {
            delegation_pool::withdraw(
                &vault_signer, delegation_pool_address, withdraw_amount
            );

            let unlocked_apt =
                vault::withdraw_apt(
                    withdraw_amount - config::get_min_pending_inactive()
                );
            let buffer_apt = vault::withdraw_apt(config::get_min_pending_inactive());
            vault::deposit_unlocked_apt(unlocked_apt);
            vault::deposit_buffer_apt(buffer_apt);
        };

        i = i + 1;
    };
}
```

However, the current logic assumes that `withdraw_amount` is always greater than or equal to the minimum threshold specified by `get_min_pending_inactive`. If this condition is not met, either due to protocol parameter changes or rounding errors during delegation, the subtraction fails, resulting in unexpected aborts and potentially blocking user withdrawals.

© 2025 Otter Audits LLC. All Rights Reserved.

Remediation

Include a condition to transfer the full withdrawn amount to the buffer if it is less than `min_pending_inactive`, preventing unexpected aborts and ensuring consistent processing.

Patch

Fixed in 212a282.

© 2025 Otter Audits LLC. All Rights Reserved.

05 — General Findings

Here, we present a discussion of general findings 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	Description
OS-KOF-SUG-00	Suggestions regarding the removal of redundant and unutilized code and improving the overall efficiency of the codebase, ensuring adherence to coding best practices.

© 2025 Otter Audits LLC. All Rights Reserved.

Code Maturity OS-KOF-SUG-00

Description

1. In the current implementation of `config`, `get_global_config` reconstructs the `GlobalConfig` structure field by field. A direct reference to the global resource is sufficient. Also, `get_validator_addresses` unnecessarily dereferences and re-references the field utilizing `*&`, which is redundant and may be removed.

   >_ *config.move* RUST

   ```rust
   fun get_global_config(): GlobalConfig acquires GlobalConfig {
       let global_config = borrow_global<GlobalConfig>(@kofi);
       GlobalConfig {
           gateway_config: global_config.gateway_config,
           ...
           package_metadata: global_config.package_metadata
       }
   }

   public fun get_validator_addresses(): vector<address> acquires GlobalConfig {
       *&borrow_global<GlobalConfig>(@kofi).validator_config.addresses
   }
   ```

2. In `init_module`, `access_control::initialize_owner` is called with a hardcoded address during contract initialization, limiting deployment flexibility across environments. Avoid hardcoding the address and instead update it with a reference defined in `Move.toml` to improve maintainability.

   >_ *config.move* RUST

   ```rust
   fun init_module(admin: &signer) {
       // initialize access control
       access_control::initialize_owner(
           admin,
           @0xc0de36135d4eda6ba6cada45fdab868cc5dde0bac1c6ed798af87a631e5a825f
       );
       ...
   }
   ```

3. The `SnapshotState` structure in `rewards_manager` and `handle_apt_deposit` in `minting_manager` serve no purpose and should be removed to improve code clarity and eliminate dead code. Similarly, unutilized variables highlighted during compilation should be removed.

© 2025 Otter Audits LLC. All Rights Reserved.

4. In `withdraw_manager::request_withdrawal`, replace the hardcoded value of `10000` to improve

>_ *withdraw_manager.move* RUST

```rust
let withdraw_fees = kapt_amount * config::get_withdrawal_fee() / 10000;
```

Remediation

Implement the above-mentioned suggestions.

Patch

1. Fixed in 35d383b.
2. Fixed in b3515c2.
3. Fixed in f88f6ef.
4. Fixed in 7060e40.

© 2025 Otter Audits LLC. All Rights Reserved.

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.

© 2025 Otter Audits LLC. All Rights Reserved.

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.

© 2025 Otter Audits LLC. All Rights Reserved.