Security Review - Sendit Lending Protocol

conducted by Neodyme AG

Lead Auditor:	Sebastian Fritsch
Second Auditor:	Nico Gründel
Third Auditor:	Benjamin Walny
Fourth Auditor:	Tobias Bucher
Administrative Lead:	Jasper Slussalek

December 08, 2025

Table of Contents

Appendices

1 | Executive Summary

Neodyme audited **Sendit's** on-chain lending and margin trading programs from October 2025 until November 2025.

The review was time-boxed to two weeks and not a full security audit. It consisted of a review of the changes to the SPL Lending program by Sendit as well as Sendit's `myield` and `cyield` programs. According to Neodymes Rating Classification, **8 security relevant** and **3 informational** were found. The number of findings identified throughout the audit, grouped by severity, can be seen in Figure 1.

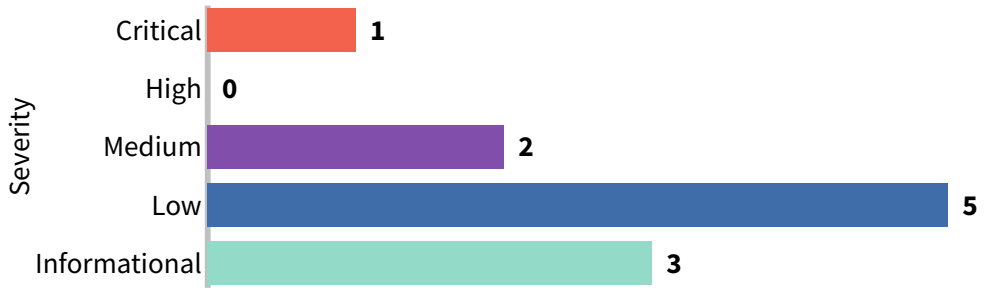

Figure 1: Overview of Findings

The auditors reported all findings to the Sendit developers, who addressed them promptly. The security fixes were verified for completeness by Neodyme. In addition to these findings, Neodyme delivered the Sendit team a list of nit-picks and additional notes that are not part of this report.

Disclaimer: Please note that time-boxed reviews do not give the same security guarantees as full audits. While we do our best to catch all critical bugs in the time provided, the chance that we miss such a bug is higher than in a full audit.

2 | Introduction

From October 2025 until November 2025, Sendit engaged Neodyme to do a time-boxed security review of their lending and margin trading programs. Two senior security researchers from Neodyme conducted independent reviews of the contracts between the 28th of October 2025 and the 7th of November 2025. Additionally two junior auditors joined the audit. Both senior auditors have a long track record of finding critical and other vulnerabilities in Solana programs, as well as in Solana's core code itself, and have extensive knowledge about the SPL lending contract.

The review focused on the technical security of the contracts. Additionally we considered some economical attacks on the lending and margin trading mechanisms. In the following sections, we present our findings.

Summary of Findings

All found issues were **quickly remediated**. In total, the audit revealed:

1 critical • **0** high-severity • **2** medium-severity • **5** low-severity • **3** informational

issues.

3 | Scope

The contract audit's scope comprised of three major components:

- Primarily the changes to the SPL lending protocol
- Additionally, the `cyield` vault contract and the `myield` vault contract.

Special focus was placed on the following two topics:

- **Vesting fees**, due to the fact that fees are taken directly from the deposit account
- **Yield Looping**, due to the fact that it is designed to (temporarily) violate invariants

Neodyme considers the source code, located at https://github.com/onogroup/partner-octane/, in scope for this audit. Third-party dependencies are not in scope.

During the audit, minor changes and fixes were made by Sendit, which the auditors also reviewed in-depth.

Relevant source code revisions are:

- `5369c92fe7675aedd2034bcb899755f9a277eec7` · Start of the audit
- `ec100f59732aa0df64fc492413e176e83465b1f4` · Last reviewed revision

4 | Project Overview

This section provides an overview of Sendit - covering core functionality, high-level design and its instructions.

Functionality

The time-boxed review focused on three main components, described below.

`lending`

The `lending` contract is derived from Solend (now Safe), which itself is built on the SPL lending architecture. It is implemented in raw Rust.

On top of basic lending mechanics, Sendit introduced additional features that were included in scope, such as:

- **Liquidity incentives**: Vault depositors receive incentives proportionally to their collateral share.
- **Yield looping**: Enables cycling borrowed SOL back into the vault to compound returns.
- **Simplified lending model**: Only SOL can be borrowed, while collateral must be non-SOL.
- **Loan origination fee flow**: Origination fees are redirected to SOL liquidity providers.

Various other extensions were also reviewed. As the central protocol component, the changes to the `lending` contract received the most examination.

`cyield`

The `cyield` program serves as a SOL-based vault system. Users deposit SOL, and a permissioned operator allocates these funds across lending pools.

Withdrawals are possible at any time; however, withdrawals that require market interactions do incur exit costs if the user deposited recently.

Incentives accrued from the lending contract are periodically converted to SOL by the operator at the current exchange rate when the corresponding instruction is executed.

`myield`

`myield` operates similarly to `cyield`, but for memecoins. Each vault is tied to a single lending market and aims to capture positive yield spreads, often driven by incentive emissions.

A target LTV is maintained per vault and adjusted by the operator as markets move.

Instructions

New instructions in `lending`

The `lending` contract exposes 6 new instructions, summarized below.

Instruction	Category	Summary
DepositMaxReserveLiquidityAndObligationCollateral	Permissionless	Deposits an account's full token balance as liquidity and collateral; convenience variant of DepositReserveLiquidityAndObligationCollateral
RepayMaxObligationLiquidity	Obligation-Owner only	Repays an obligation's borrowed assets using the maximum available liquidity
YieldLoop	Obligation-Owner only	Borrows SOL at the maximum allowed amount and redeploys it to compound yield
YieldUnloop	Obligation-Owner only	Reverses YieldLoop, reducing leverage and exposure
MarkLiquidatable	Permissionless	Marks an obligation as liquidatable for a liquidator
CloseLiquidationMark	Liquidator-only	Closes a liquidation mark created via MarkLiquidatable

`cyield`

The `cyield` contract exposes 8 instructions, summarized below.

Instruction	Category	Summary
CreateVault	Operator-only	Creates a new vault
Deposit	Permissionless	Deposits SOL into a vault and mints vault tokens in return
Withdraw	User-only	Redeems vault tokens for available SOL liquidity
WithdrawWithMarket	User-only	Redeems vault tokens for SOL, including funds pulled from a lending market if needed
DeployToMarket	Operator-only	Deploys vault liquidity into a lending market
WithdrawFromMarket	Operator-only	Withdraws liquidity from a lending market back to the vault
WithdrawFees	Operator-only	Withdraws accumulated performance fees
ClaimAndSellRewards	Operator-only	Claims market incentive tokens and swaps them to SOL

`myield`

The `myield` contract exposes 8 instructions, summarized below.

Instruction	Category	Summary
CreateVault	Operator-only	Creates a new vault
Deposit	Permissionless	Deposits memecoins into a lending market
Withdraw	User-only	Redeems vault tokens back to liquidity tokens including earned yield
IncreaseLTV	Vault-Operator-only	Increases vault LTV via leveraged looping
DecreaseLTV	Vault-Operator-only	Reduces LTV by unwinding a leverage loop
RealiseInterestLoss	Vault-Operator-only	Covers a SOL debt shortfall in exchange for collateral tokens
AccrueRewards	Permissionless	Accrues lending-market incentive rewards into the vault
AccrueInterestGain	Vault-Operator-only	Realizes interest gains accrued within the lending position

5 | Findings

This section outlines all of our findings. They are classified into one of five severity levels, detailed in Appendix C. In addition to these findings, Neodyme delivered the Sendit team a list of nit-picks and additional notes which are not part of this report.

All findings are listed in Table 4 and further described in the following sections.

Identifier	Name	Severity	Status
ND-SDIT-CR-01	`RedeemReserveCollateral` does not check incentive accounts	CRITICAL	Resolved
ND-SDIT-MD-01	LiquidationMarks can be DoSed	MEDIUM	Resolved
ND-SDIT-MD-02	Vault incentive account can be DoSed	MEDIUM	Resolved
ND-SDIT-LO-01	`DepositReserveLiquidity` can be DoSed	LOW	Resolved
ND-SDIT-LO-02	`InitReserve` is permissionless	LOW	Resolved
ND-SDIT-LO-03	`InitReserve`: Fee owner constraint can be bypassed	LOW	Resolved
ND-SDIT-LO-04	Incorrect incentive calculation	LOW	Resolved
ND-SDIT-LO-05	WithdrawWithMarket: Flooring of transferred liquidity can fail	LOW	Resolved
ND-SDIT-IN-01	`DepositReserveLiquidity`: No address check on `user_incentive_info`	INFORMATIONAL	Resolved
ND-SDIT-IN-02	Closed LiquidationMarks can be revived	INFORMATIONAL	Resolved
ND-SDIT-IN-03	`DeployToMarket`: Wrong token program	INFORMATIONAL	Resolved

Table 4: Findings

[ND-SDIT-CR-01] `RedeemReserveCollateral` does not check incentive accounts

Severity	Impact	Affected Component	Status
CRITICAL	Loss of Funds	Incentive Withdrawal	Resolved

Sendit allows pool administrators to configure incentive rewards for a lending market, which depositors can earn in addition to regular interest. Incentives are paid out in the internal function `_redeem_reserve_collateral_incentive`, which is invoked by `RedeemReserveCollateral`. However, this call path performs **no** validation of the provided accounts, most critically, it does not verify that `incentive_token_account_info` is the legitimate incentive-vault token account.

```
if pending_incentives > 0 {
    // Update user's incentive debt (mark rewards as claimed)
    let mut updated_user_data = existing_user_data.clone();
    updated_user_data.incentive_debt = reserve
        .liquidity_incentives
        .calculate_incentive_debt(updated_user_data.ctoken_amount)?;

    // Transfer incentives
    let authority_signer_seeds = &[
        lending_market_info.key.as_ref(),
        &[lending_market.bump_seed],
    ];

    spl_token_transfer_liquidity(LiquidityTokenTransferParams {
        mint: incentive_token_mint_info.clone(),
        source: incentive_token_account_info.clone(),
        destination: user_incentive_token_account_info.clone(),
        amount: pending_incentives as u64,
        authority: lending_market_authority_info.clone(),
        authority_signer_seeds,
        token_program: incentive_token_program_id.clone(),
        decimals: reserve.liquidity_incentives.incentive_token_decimals,
    })?;
}
```

Because the program accepts any account as the incentive-vault source, an attacker can replace the correct `reserve.liquidity_incentives.incentive_token_account` with `reserve.liquidity.supply_pubkey`. This changes the transfer to take the tokens from the reserves liquidity vault, effectively draining it.

The attack becomes profitable whenever the reserve liquidity token is more valuable than the incentive token and the incentive decimals are 9 or when the reserve incentives are not set yet and can be set by an attacker. Three markets were affected by this flaw.

Resolution

Sendit quickly fixed the bug in commit `f990e0502527269fc44fc5c11159ba2407db9fa9` by enforcing that the supplied token account matches `reserve.liquidity_incentives.incentive_token_account`. Neodyme verified the fix.

[ND-SDIT-MD-01] LiquidationMarks can be DoSed

Severity	Impact	Affected Component	Status
MEDIUM	Denial of Service of permissionless liquidations	Liquidation	Resolved

In the lending contract, a user can create a `LiquidationMark` to mark a liquidation as liquidatable by them in the future. This `LiquidationMark` is a PDA derived as `PDA([obligation_info.key, liquidation_info.key])`.

Because this address is fully predictable and the program uses a `CreateAccount` instruction to initialize the `LiquidationMark` account, an attacker can pre-fund the PDA with lamports. Once the PDA already exists with a balance, the program cannot create the expected account anymore, causing all attempts to set a `LiquidationMark` at that address to fail. This enables a denial of service against permissionless liquidations.

```
let authority_signer_seeds: &[&[&[u8]]] = &[&[
    obligation_info.key.as_ref(),
    liquidator_info.key.as_ref(),
    &[bump],
]];
let create_ix = create_account(
    liquidator_info.key,
    liquidation_mark_info.key,
    Rent::get()?.minimum_balance(LiquidationMark::LEN),
    LiquidationMark::LEN as u64,
    program_id,
);
let account_infos = vec![
    liquidator_info.clone(),
    liquidation_mark_info.clone(),
    system_account_info.clone(),
];
invoke_signed(&create_ix, &account_infos, authority_signer_seeds)?;
```

Resolution

Sendit quickly fixed the issue by using the idempotent account creation mechanism found at https://github.com/solana-program/associated-token-account/blob/main/program/src/tools/account.rs#L16 in commit `8b8691bf33169465b81f28ee065210041b43db99`. Neodyme verified the fix.

[ND-SDIT-MD-02] Vault incentive account can be DoSed

Severity	Impact	Affected Component	Status
MEDIUM	Blocks withdraw functionality	cyield: Withdraw	Resolved

The cyield contract creates the vault's incentive token account in `WithdrawWithMarket`, `WithdrawFromMarket`, and `ClaimAndSellRewards` whenever it is missing. This account is a PDA derived as `PDA(["incentive_token_account", vault.key, incentive_token_mint])`.

Because the address is fully predictable and the account is initialized via a `CreateAccount` instruction, an attacker can preemptively fund that PDA with lamports. Once funded, the program can no longer create the expected account, causing all affected instructions to fail and thereby blocking vault withdrawals.

blab.rs, lines 10-15

```rust
// Setup incentive token account if needed
if vault_incentive_token_account.data_is_empty() {
    // Create token account for incentive token
    let seeds = &[
        b"incentive_token_account".as_ref(),
        &vault.key().to_bytes()[..],
        &reserve.liquidity_incentives.incentive_token_mint.to_bytes()[..],
        &[vault_incentive_token_account_bump],
    ];
    let signer_seeds = &[&seeds[..]];
    let cpi_ctx = CpiContext::new_with_signer(
        token_program.to_account_info(),
        anchor_lang::system_program::CreateAccount {
            from: payer,
            to: vault_incentive_token_account.to_account_info(),
        },
        signer_seeds,
    );
    anchor_lang::system_program::create_account(
        cpi_ctx,
        Rent::get()?.minimum_balance(account_len as usize),
        account_len as u64,
        &token_program.key(),
    )?;
}
```

Resolution

Sendit quickly fixed the issue by using the idempotent account creation mechanism found at https://github.com/solana-program/associated-token-account/blob/main/program/src/tools/account.rs#L16 in commit `70a1ab3d6dad781e35c7c9fe5bc8a453a92ac767`. Neodyme verified the fix.

[ND-SDIT-LO-01] `DepositReserveLiquidity` can be DoSed

Severity	Impact	Affected Component	Status
LOW	Denial of Service	Liquidity Deposits	Resolved

In `DepositReserveLiquidity`, the lending contract creates a `UserIncentiveData` account to track a user's incentive-related state. This `UserIncentiveData` account is a PDA derived as `PDA(["user-incentive", user_transfer_authority_info.key, reserve.key])`.

Since the PDA address is fully predictable and the program initializes the account using a `CreateAccount` instruction, an attacker can prefund the PDA with lamports before the legitimate depositor calls the instruction. Once the PDA already exists with a balance, the program cannot create the expected account, causing all liquidity deposit attempts that rely on this account to fail, resulting in a denial of service.

Resolution

Sendit quickly fixed the issue by using the idempotent account creation mechanism found at https://github.com/solana-program/associated-token-account/blob/main/program/src/tools/account.rs#L16 in commit `0a277b491e38f56daf8adfca0a7b99252de3fa5e`. Neodyme verified the fix.

[ND-SDIT-LO-02] `InitReserve` is permissionless

Severity	Impact	Affected Component	Status
LOW	Griefing of market initialization	Market initialization	Resolved

`InitReserve` does not verify that `lending_market.owner == lending_market_owner_info`. As a result, an attacker could inject their own reserve into the lending market before the legitimate initializer does, effectively sabotaging the intended setup. This issue does not arise when both the lending market and the reserve are initialized atomically within the same transaction.

Resolution

Sendit resolved the problem by introducing the missing ownership check in commit `5a84fc68c119839ea4965e781fbc829aea8d6d6f`. Neodyme has verified the fix.

[ND-SDIT-LO-03] `InitReserve`: Fee owner constraint can be bypassed

Severity	Impact	Affected Component	Status
LOW	Market creation with an invalid fee destination	Market initialization	Resolved

Within `InitReserve`, the lending contract does not verify that the `protocol_fee_receiver_info` account is actually owned by the token program. An attacker can create a non–token-program-owned account that mimics the structure of a token account and sets its `owner` field to the global `FEE_RECEIVER`.

After the reserve is initialized, the attacker can delete this fake account and recreate it with a legitimate token account they fully control, effectively redirecting all protocol fees for that market to themselves.

Resolution

Sendit addressed the issue by adding an explicit owner check in commit `8abc9d221362a9679759953a8e8d21a1bb28f985`. Neodyme has verified the fix.

[ND-SDIT-LO-04] Incorrect incentive calculation

Severity	Impact	Affected Component	Status
LOW	Wrong incentive payout	Liquidity incentives	Resolved

In `add_incentives`, the program computes `incentives_per_slot` using the incentive mint's decimals. However, if the mint has more than `9` decimals, the scaling logic becomes incorrect: `10u128.pow(9.saturating_sub(decimals))` underflows and effectively collapses to `10^0`, producing an invalid conversion factor and therefore an incorrect incentive distribution.

```
let amount_in_lamports = (incentive_amount as u128)
    .checked_mul(10u128.pow(9_u32.saturating_sub(decimals as u32)))
    .ok_or(LendingError::MathOverflow)?;

// Calculate incentives per slot (amount / total_slots)
let new_incentives_per_slot = amount_in_lamports
    .checked_mul(INCENTIVE_PRECISION)
    .ok_or(LendingError::MathOverflow)?
    .checked_div(raw_total_slots_u128)
    .ok_or(LendingError::MathUnderflow)?;
```

Resolution

Sendit fixed the issue by enforcing a mint has `decimals <= 9` in commit `ec100f59732aa0df64fc492413e176e83465b1f4`. Neodyme verified the fix.

[ND-SDIT-LO-05] WithdrawWithMarket: Flooring of transferred liquidity can fail

Severity	Impact	Affected Component	Status
LOW	Withdrawal failure	cyield: WithdrawWithMarket	Resolved

When withdrawing liquidity with `WithdrawWithMarket`, the instruction attempts to compensate for the flooring behavior of `liquidity_to_collateral` by applying `saturating_add(1)` to the required liquidity amount. For tokens with a high value in lamports, this adjustment does not reliably offset the floor operation, causing the computed collateral amount to fall short. This results in sporadic failures of `WithdrawWithMarket`, depending on the exchange relationship between the cToken and SOL.

```
// Add 1 to ensure we withdraw enough liquidity because
liquidity_to_collateral floors
let c_token_amount = reserve
    .collateral_exchange_rate()?
    .liquidity_to_collateral(needed_liquidity.saturating_add(1))?
    .min(ctx.accounts.vault_collateral_token_account.amount);
```

Resolution

Sendit fixed the issue in commit `b9d63b9509f5345e1a6d5c244c5cd1a701bbcfe8` by moving the `saturating_add` after the call to `liquidity_to_collateral`. Neodyme verified the fix.

[ND-SDIT-IN-01] `DepositReserveLiquidity`: No address check on `user_incentive_info`

Severity	Impact	Affected Component	Status
INFORMATIONAL	None	Liquidity Deposits	Resolved

In `DepositReserveLiquidity`, the lending contract does not explicitly validate that the correct PDA was provided for `user_incentive_info` in cases where the account already contains data. This is not a security issue because `_update_user_incentive_ctoken` always writes to `user_incentive_info`, causing the Solana runtime to implicitly enforce that the account is program-owned. However, for correctness and consistency, the PDA should be validated on every call, not only when the account is newly created.

```
if user_incentive_info.data_is_empty() && reserve.config.loan_to_value_ratio
== 0 {
    // Find and validate user incentive data PDA
    let bump_seed = find_and_validate_user_incentive_data_address(
        program_id,
        user_transfer_authority_info.key,
        reserve_info.key,
        user_incentive_info,
    )?;

    // […]
}
```

Resolution

Sendit moved the address check in commit `9bd8e815bf4759d545a0a75282d303ab4d04c27a`. Neodyme verified the fix.

[ND-SDIT-IN-02] Closed LiquidationMarks can be revived

Severity	Impact	Affected Component	Status
INFORMATIONAL	None	Liquidation marks	Resolved

In `CloseLiquidationMark`, the program closes the `LiquidationMark` account by setting its lamports to zero. This marks the account for garbage collection by the Solana runtime. However, since ownership is not reassigned to the system program, an attacker could theoretically top up the account's rent after closure, preventing its cleanup. This does not introduce a security risk as `MarkLiquidatable` is idempotent, but it is generally considered poor practice.

Resolution

Sendit addressed the issue by assigning the account back to the system program while closing the account in commit `58420c0fa5d36409177a3109af6e38dff92526d2`. Neodyme has verified the fix.

[ND-SDIT-IN-03] `DeployToMarket`: Wrong token program

Severity	Impact	Affected Component	Status
INFORMATIONAL	None	cyield	Resolved

The call to `DepositReserveLiquidity` in cyield's `DeployToMarket` reuses the `token_program` account for both the `token_program` and `collateral_token_program` function arguments, instead of using the `collateral_token_program` account supplied.

Resolution

Sendit addressed the issue in commit `60d067a39bd0c5943c37cb7ff7f6f4abe3028516`. Neodyme has verified the fix.

A | About Neodyme

Security is difficult.

To understand and break complex things, you need a certain type of people. People who thrive in complexity, who love to play around with code, and who don't stop exploring until they fully understand every aspect of it. That's us.

Our team never outsources audits. Having found over 80 High or Critical bugs in Solana's core code itself, we believe that Neodyme hosts the most qualified auditors for Solana programs. We've also found and disclosed critical vulnerabilities in many of Solana's top projects and have responsibly disclosed issues that could have resulted in the theft of over $10B in TVL on the Solana blockchain.

All of our team members have a background in competitive hacking. During such hacking competitions, called CTFs, we competed and collaborated while finding vulnerabilities, breaking encryption, reverse engineering complicated algorithms, and much more. Through the years, many of our team members have won national and international hacking competitions, and keep ranking highly among some of the hardest CTF events worldwide. In 2020, some of our members started experimenting with validators and became active members in the early Solana community. With the prospect of an interesting technical challenge and bug bounties, they quickly encouraged others from our CTF team to look for security issues in Solana. The result was so successful that after reporting several bugs, in 2021, the Solana Foundation contracted us for source code auditing. As a result, Neodyme was born.

B | Methodology

We are not checklist auditors.

In fact, we pride ourselves on that. We adapt our approach to each audit, investing considerable time into understanding the program upfront and exploring its expected behavior, edge cases, invariants, and ways in which the latter could be violated.

We use our uniquely deep knowledge of Solana internals, and our years-long experience in auditing Solana programs to find bugs that others miss. We often extend our audit to cover off-chain components in order to see how users could be tricked or the contract affected by bugs in those components.

Nonetheless, we also have a list of common vulnerability classes, which we always exhaustively look for. We provide a sample of this list here.

Select Common Vulnerabilities

Our most common findings are still specific to Solana itself. Among these are vulnerabilities such as the ones listed below:

- Insufficient validation, such as:
  - Missing ownership checks
  - Missing signer checks
  - Signed invocation of unverified programs
  - Account confusions
  - Missing freeze authority checks
  - Insufficient SPL account verification
  - Dangerous user-controlled bumps
  - Insufficient Anchor account linkage
- Account reinitialization vulnerabilities
- Account creation DoS
- Redeployment with cross-instance confusion
- Missing rent exemption assertion
- Casting truncation
- Arithmetic over- or underflows
- Numerical precision and rounding errors
- Anchor pitfalls, such as accounts not being reloaded
- Non-unique seeds
- Issues arising from CPI recursion
- Log truncation vulnerabilities
- Vulnerabilities specific to integration of Token Extensions, for example unexpected external token hook calls

Apart from such Solana-specific findings, some of the most common vulnerabilities relate to the general logical structure of the contract. Specifically, such findings may be:

- Errors in business logic
- Mismatches between contract logic and project specifications
- General denial-of-service attacks
- Sybil attacks
- Incorrect usage of on-chain randomness
- Contract-specific low-level vulnerabilities, such as incorrect account memory management
- Vulnerability to economic attacks
- Allowing front-running or sandwiching attacks

Miscellaneous other findings are also routinely checked for, among them:

- Unsafe design decisions that might lead to vulnerabilities being introduced in the future
    - Additionally, any findings related to code consistency and cleanliness
- Rug pull mechanisms or hidden backdoors

Often, we also examine the authority structure of a contract, investigating their security as well as the impact on contract operations should they be compromised.

Over the years, we have found hundreds of high and critical severity findings, many of which are highly nontrivial and do not fall into the strict categories above. This is why our approach has always gone way beyond simply checking for common vulnerabilities. We believe that the only way to truly secure a program is a deep and tailored exploration that covers all aspects of a program, from small low-level bugs to complex logical vulnerabilities.

C | Vulnerability Severity Rating

We use the following guideline to classify the severity of vulnerabilities. Note that we assess each vulnerability on an individual basis and may deviate from these guidelines in cases where it is well-founded. In such cases, we always provide an explanation.

Severity	Description
CRITICAL	Vulnerabilities that will likely cause loss of funds. An attacker can trigger them with little or no preparation, or they are expected to happen accidentally. Effects are difficult to undo after they are detected.
HIGH	Bugs that can be used to set up loss of funds in a more limited capacity, or to render the contract unusable.
MEDIUM	Bugs that do not cause direct loss of funds but that may lead to other exploitable mechanisms, or that could be exploited to render the contract partially unusable.
LOW	Bugs that do not have a significant immediate impact and could be fixed easily after detection.
INFORMATIONAL	Bugs or inconsistencies that have little to no security impact, but are still noteworthy.

Additionally, we often provide the client with a list of nit-picks, i.e. findings whose severity lies below Informational. In general, these findings are not part of the report.

Neodyme AG

Dirnismaning 55
Halle 13
85748 Garching
Germany

E-Mail: contact@neodyme.io

https://neodyme.io