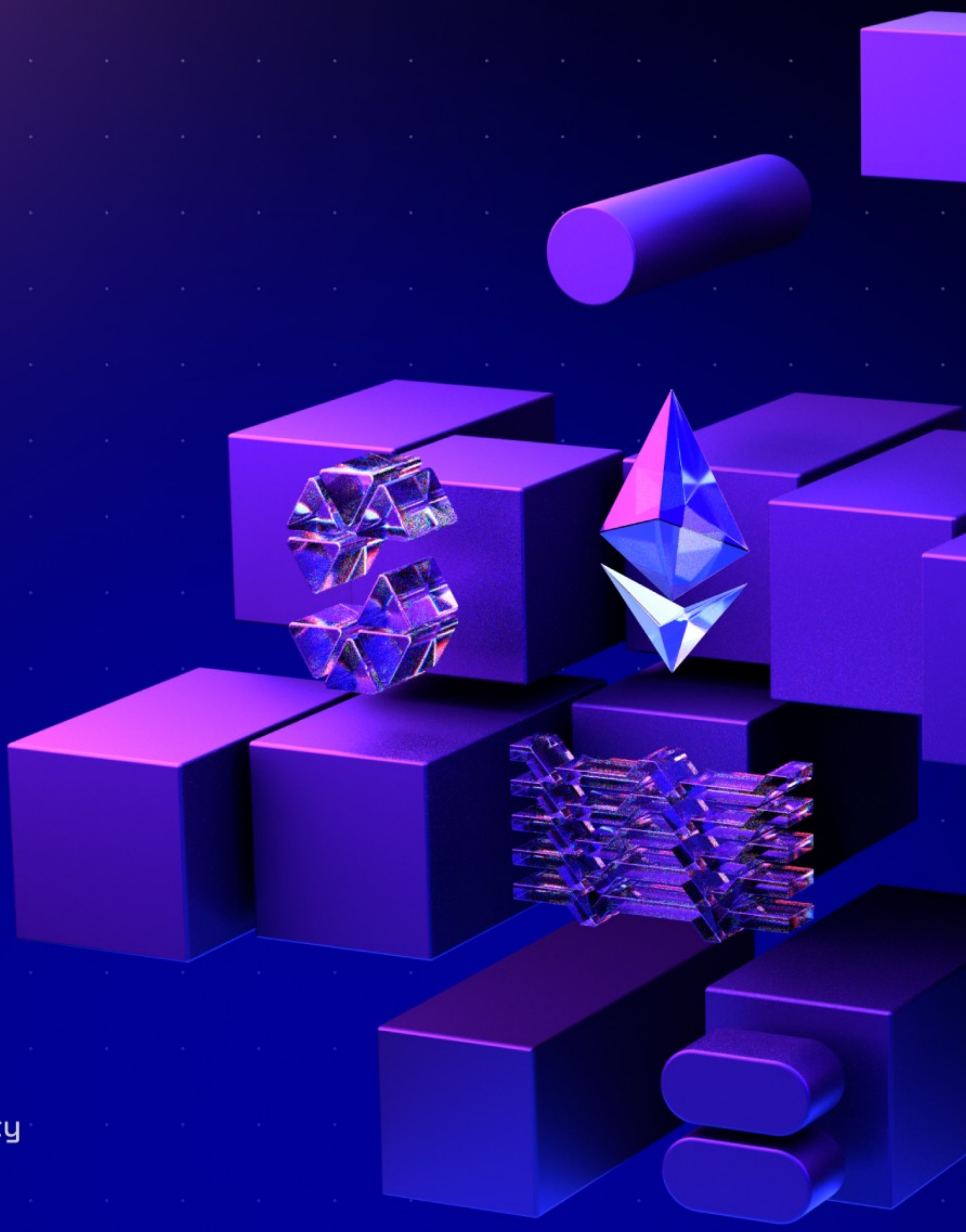
ackee
blockchain security

PWN

Protocol

20.12.2024



Ackee Blockchain Security

Contents

1. Document Revisions

1.0-draft	Draft Report	02.12.2024
1.0	Final Report	04.12.2024
2.0	Final Report	17.12.2024
2.1	Final Report	20.12.2024

2. Overview

This document presents our findings in reviewed contracts.

2.1. Ackee Blockchain Security

Ackee Blockchain Security is an in-house team of security researchers performing security audits focusing on manual code reviews with extensive fuzz testing for Ethereum and Solana. Ackee is trusted by top-tier organizations in web3, securing protocols including Lido, Safe, and Axelar.

We develop open-source security and developer tooling Wake for Ethereum and Trident for Solana, supported by grants from Coinbase and the Solana Foundation. Wake and Trident help auditors in the manual review process to discover hardly recognizable edge-case vulnerabilities.

Our team teaches about blockchain security at the Czech Technical University in Prague, led by our co-founder and CEO, Josef Gattermayer, Ph.D. As the official educational partners of the Solana Foundation, we run the School of Solana and the Solana Auditors Bootcamp.

Ackee's mission is to build a stronger blockchain community by sharing our knowledge.

Ackee Blockchain a.s.

Rohanske nabrezi 717/4

186 00 Prague, Czech Republic

https://ackee.xyz

hello@ackee.xyz

2.2. Audit Methodology

1. **Verification of technical specification**

   The audit scope is confirmed with the client, and auditors are onboarded to the project. Provided documentation is reviewed and compared to the audited system.

2. **Tool-based analysis**

   A deep check with Solidity static analysis tool Wake in companion with Solidity (Wake) extension is performed, flagging potential vulnerabilities for further analysis early in the process.

3. **Manual code review**

   Auditors manually check the code line by line, identifying vulnerabilities and code quality issues. The main focus is on recognizing potential edge cases and project-specific risks.

4. **Local deployment and hacking**

   Contracts are deployed in a local Wake environment, where targeted attempts to exploit vulnerabilities are made. The contracts' resilience against various attack vectors is evaluated.

5. **Unit and fuzz testing**

   Unit tests are run to verify expected system behavior. Additional unit or fuzz tests may be written using Wake framework if any coverage gaps are identified. The goal is to verify the system's stability under real-world conditions and ensure robustness against both expected and unexpected inputs.

2.3. Finding Classification

A *Severity* rating of each finding is determined as a synthesis of two sub-ratings: *Impact* and *Likelihood*. It ranges from *Informational* to *Critical*.

If we have found a scenario in which an issue is exploitable, it will be assigned an impact rating of *High*, *Medium*, or *Low*, based on the direness of the consequences it has on the system. If we haven't found a way, or the issue is only exploitable given a change in *configuration* (system settings or parameters, such as deployment scripts, compiler configurations, using multi-signature wallets for owners, etc.) or given a change in the codebase, then it will be assigned an impact rating of *Warning* or *Info*.

Low to *High* impact issues also have a *Likelihood*, which measures the probability of exploitability during runtime.

The full definitions are as follows:

Severity

		Likelihood			
		High	**Medium**	**Low**	**N/A**
Impact	**High**	Critical	High	Medium	-
	Medium	High	Medium	Low	-
	Low	Medium	Low	Low	-
	Warning	-	-	-	Warning
	Info	-	-	-	Info

Table 1. Severity of findings

Impact

- **High** - Code that activates the issue will lead to undefined or catastrophic consequences for the system.
- **Medium** - Code that activates the issue will result in consequences of serious substance.
- **Low** - Code that activates the issue will have outcomes on the system that are either recoverable or don't jeopardize its regular functioning.
- **Warning** - The issue cannot be exploited given the current code and/or *configuration*, but could be a security vulnerability if these were to change slightly. If we haven't found a way to exploit the issue given the time constraints, it might be marked as a "Warning" or higher, based on our best estimate of whether it is currently exploitable.
- **Info** - The issue is on the borderline between code quality and security. Examples include insufficient logging for critical operations. Another example is that the issue would be security-related if code or *configuration* was to change.

Likelihood

- **High** - The issue is exploitable by virtually anyone under virtually any circumstance.
- **Medium** - Exploiting the issue currently requires non-trivial preconditions.
- **Low** - Exploiting the issue requires strict preconditions.

2.4. Review Team

The following table lists all contributors to this report. For authors of the specific revision, see the “Revision team” section in the respective “Report revision” chapter.

Member’s Name	Position
Michal Převrátil	Lead Auditor
Naoki Yoshida	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

2.5. Disclaimer

We’ve put our best effort to find all vulnerabilities in the system, however our findings shouldn’t be considered as a complete list of all existing issues. The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them.

3. Executive Summary

PWN protocol offers a platform for peer-to-peer lending and borrowing of ERC-20 tokens with any token as collateral. Loans may be opened with different presets (loan types) defining the relationship between the borrowed credit amount and the collateral amount.

Revision 1.0

PWN engaged Ackee Blockchain Security to perform a security review of the PWN protocol with a total time donation of 19 engineering days in a period between November 4 and December 3, 2024, with Michal Převrátil as the lead auditor.

The audit was performed on the commit `7ea4de`[1] in the PWN Protocol repository and commit `17db9b`[2] in the PWN Periphery repository.

The scope of the audit included:

- the `src` directory in the PWN Protocol repository, excluding `src/Deployments.sol`; and
- the `src/pool-adapter` directory in the PWN Periphery repository.

We began our review by preparing manually guided differential forking fuzz tests in the Wake testing framework to verify the protocol implementation and integration with external dependencies, including Chainlink and Aave protocols. Our fuzz tests identified findings C2, M1, M2, W2, and W3.

The Wake static analysis detectors identified the C1 and M4 issues. During manual review, we focused on the following aspects:

- external calls to untrusted contracts cannot be abused for reentrancy attacks;

- contracts are resistant to signature replay attacks;
- token arithmetics inside the protocol match the documentation and our expectations; and
- integration with external dependencies is correctly implemented.

Our review resulted in 12 findings, ranging from Info to Critical severity. The most severe findings C1 and C2 posed risk of all ERC-20 tokens deposited to the protocol being stolen. Both critical vulnerabilities were discovered to be present in already deployed PWN contracts on several major chains, including the Ethereum mainnet, Polygon, Arbitrum, and Optimism. The code that contained both critical vulnerabilities was already audited by two independent companies (not Ackee Blockchain Security).

Ackee Blockchain Security initiated an immediate responsible disclosure to PWN as soon as the findings were discovered. Thanks to prompt engagement, all assets were protected and vulnerabilities mitigated.

Ackee Blockchain Security recommends PWN:

- implement static analysis tools like Wake to detect potential attack vectors;
- apply reentrancy guards on all public functions that perform external calls to untrusted contracts;
- ensure all Chainlink-like feed registry contracts maintained by PWN provide necessary price feeds and comply with expected behavior;
- exercise caution during contract upgrades regarding storage layout; and
- address all reported issues.

See Report Revision 1.0 for the system overview and trust model.

Revision 2.0

PWN engaged Ackee Blockchain Security to perform a security review of the fixes for the findings from the previous revision, with a total time donation of 4 engineering days in a period between December 16 and December 17, 2024, with Michal Převrátil as the lead auditor.

The review was performed on the commit `bbe7d9`[3] in the PWN Protocol repository, and the scope of the audit was the changes made to the codebase since the previous revision.

Our review started with updating the fuzz test created in the previous revision. The fuzz test discovered a new M5 issue. We then continued with running Wake static analysis detectors and performing a manual code review of the code changes.

During the manual review, we especially focused on the correct integration with Chainlink and the rest of the codebase.

Our review resulted in one Medium severity finding M5, which prevents the use of elastic Chainlink loan proposals due to an incorrect implementation of EIP-712 data encoding.

Ackee Blockchain Security recommends PWN:

- only deploy the updated `PWNConfig` contract with a new proxy to avoid issues caused by storage layout changes;
- reconsider applying reentrancy guards on all public functions that perform external calls to untrusted contracts;
- be cautious when implementing EIPs to ensure full compatibility with the standard.

See Report Revision 2.0 for the system overview and trust model.

Revision 2.1

Ackee Blockchain Security performed a fix review of the finding discovered in the previous revision and an incomplete fix of C1. The review was conducted on the commit `6f390c`[4].

No new findings were identified, all reported issues were resolved.

[1] full commit hash: `7ea4de4b69642a171f5ba5febef3fb250713f9d5`

[2] full commit hash: `17db9bc8a93d710c308a5e642e7f5ab5e924e733`

[3] full commit hash: `bbe7d9044b458dd065b1f9435d8cf118df5300a4`

[4] full commit hash: `6f390c8a6627a81a90778f5aaf0443aaf39e7a9a`

4. Findings Summary

The following section summarizes findings we identified during our review. Unless overridden for purposes of readability, each finding contains:

- *Description*
- *Exploit scenario* (if severity is low or higher)
- *Recommendation*
- *Fix* (if applicable).

Summary of findings:

Critical	High	Medium	Low	Warning	Info	Total
2	0	5	1	3	2	13

Table 2. Findings Count by Severity

Findings in detail:

Finding title	Severity	Reported	Status
C1: Loan refinancing reentrancy	Critical	1.0	Fixed
C2: Incorrect optimization in loan refinancing	Critical	1.0	Fixed
M1: Chainlink common denominator bad logic	Medium	1.0	Fixed
M2: Outdated/reverting Chainlink feed causes DoS	Medium	1.0	Fixed
M3: Non-upgradable base contracts	Medium	1.0	Fixed

Finding title	Severity	Reported	Status
M4: Incorrect EIP-712 typehash	Medium	1.0	Fixed
L1: Decimal detection may lead to unexpected reverts	Low	1.0	Fixed
W1: Older versions of Aave and Compound not supported	Warning	1.0	Acknowledged
W2: `creditPerCollateralUnit` division by zero	Warning	1.0	Fixed
W3: `_checkTransfer` sender and receiver collision	Warning	1.0	Fixed
I1: `revokeNonces` nonce space can be cached	Info	1.0	Fixed
I2: `LoanDefaulted(uint40)` error parameter not named	Info	1.0	Fixed
M5: Incorrect EIP-712 data encoding	Medium	2.0	Fixed

Table 3. Table of Findings

Report Revision 1.0

Revision Team

Member's Name	Position
Michal Převrátil	Lead Auditor
Naoki Yoshida	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

System Overview

The PWN protocol allows users to establish peer-to-peer loans under well-defined terms without risk of losing funds, including liquidations. The protocol defines multiple predefined loan terms, including fixed loans and loans based on the latest Chainlink prices. Loans may be extended after a mutual agreement and refinanced for a new term.

Trust Model

Both a loan lender and a borrower must trust credit and collateral tokens not to perform any malicious behavior. The lender trusts the borrower to repay the loan. In the opposite case, the borrower loses the deposited collateral.

Fuzzing

Manually guided differential stateful fuzz tests were developed during the review to test the correctness and robustness of the system. The fuzz tests employ fork testing technique to test the system with external contracts exactly as they are deployed in the deployment environment. This is crucial to detect any potential integration issues.

A differential fuzz test keeps its own Python state according to the system's specification. Assertions are used to verify the Python state against the on-chain state in contracts.

The list of all implemented execution flows and invariants is available in Appendix B.

The fuzz tests simulate the whole system and make strict assertions about the behavior of the contracts. Findings C2, M1, M2, W2 and W3 were discovered using fuzz testing in the Wake testing framework.

The full source code of all fuzz tests is available at https://github.com/Ackee-Blockchain/tests-pwn-protocol.

Findings

The following section presents the list of findings discovered in this revision.

C1: Loan refinancing reentrancy

Critical severity issue

Impact:	High	Likelihood:	High
Target:	PWNSimpleLoan.sol	Type:	Reentrancy

Description

The `createLOAN` function enables new loan creation and existing loan refinancing operations. During refinancing, the function closes the previous loan and creates a new one while maintaining the same collateral. The function optionally supports ERC-2612 permit functionality for loan credit collection from the lender.

Listing 1. Excerpt from PWNSimpleLoan.createLOAN

```
if (callerSpec.permitData.length > 0) {
    Permit memory permit = abi.decode(callerSpec.permitData, (Permit));
    _checkPermit(msg.sender, loanTerms.credit.assetAddress, permit);
    _tryPermit(permit);
}

// Settle the loan
if (callerSpec.refinancingLoanId == 0) {
    // Transfer collateral to Vault and credit to borrower
    _settleNewLoan(loanTerms, lenderSpec);
} else {
    // Update loan to repaid state
    _updateRepaidLoan(callerSpec.refinancingLoanId);

    // Repay the original loan and transfer the surplus to the borrower if any
    _settleLoanRefinance({
        refinancingLoanId: callerSpec.refinancingLoanId,
        loanTerms: loanTerms,
        lenderSpec: lenderSpec
    });
}
```

During loan refinancing operations, the permit call executes before the previous loan is marked as repaid. This sequence creates a reentrancy vulnerability window, enabling an attacker to execute multiple `createLOAN` function calls, resulting in the division of the previous loan into multiple loans.

This finding was discovered by a [Wake](#) static analysis detector. See [Appendix B](#) for more details, including the attack transaction call trace.

Exploit scenario

Alice creates a loan against herself using a valuable token from the PWN vault as collateral and a malicious ERC-20 token as loan credit.

The malicious ERC-20 token implementation contains code to reenter the `createLOAN` function during the `permit` call:

```
address public target;
bytes public payload;

function permit(
    address owner,
    address spender,
    uint256 value,
    uint256 deadline,
    uint8 v,
    bytes32 r,
    bytes32 s
) public override {
    super.permit(owner, spender, value, deadline, v, r, s);
    (bool success, ) = target.call(payload);
    require(success, "MaliciousERC20: target call failed");
}
```

Alice configures the `target` variable to the PWN vault address and crafts the `payload` to call the `createLOAN` function for refinancing the previously created loan.

Alice executes `createLOAN` to refinance her initial loan using the permit feature. The reentrant call results in double refinancing of the original loan, creating two separate loans while closing the original loan.

Alice then repays both new loans and receives double the collateral of the original loan, effectively extracting half of the collateral value from the PWN vault.

Recommendation

Change the state of the loan before any external call, i.e., move the `_updateRepaidLoan(callerSpec.refinancingLoanId)` call before any external call.

Apply reentrancy guard to all state-changing functions of the PWN vault.

Partial solution 2.0

The `_updateRepaidLoan` state-changing function call was moved before any external call, and the permit functionality was removed. However, reentrancy guards were not applied.

Fix 2.1

Reentrancy guards were added to all external state-changing functions in the `PWNVault` contract with an exception of the `makeExtensionProposal` function which was evaluated as safe.

Go back to Findings Summary

C2: Incorrect optimization in loan refinancing

Critical severity issue

Impact:	High	Likelihood:	High
Target:	PWNSimpleLoan.sol	Type:	Logic error

Description

During loan refinancing, the `_settleLoanRefinance` function implements an optimization that only pulls the surplus credit token amount from the lender when the original and new loan lenders are identical. This optimization omits pulling the common credit amount shared between the previous and new loans.

Listing 2. Excerpt from PWNSimpleLoan._settleLoanRefinance

```
bool shouldTransferCommon =
    loanTerms.lender != loanOwner ||
    (loan.originalLender == loanOwner && loan.originalSourceOfFunds !=
lenderSpec.sourceOfFunds);
```

Listing 3. Excerpt from PWNSimpleLoan._settleLoanRefinance

```
if (shouldTransferCommon) {
    creditHelper.amount = common;
    _pull(creditHelper, loanTerms.lender);
}
```

The credit amount of the previous loan is claimed through the `tryClaimRepaidLOAN` call, considering the optimization. In this scenario, only the surplus amount is transferred to the lender.

Listing 4. Excerpt from PWNSimpleLoan._settleLoanRefinance

```
try this.tryClaimRepaidLOAN({
    loanId: refinancingLoanId,
```

```
    creditAmount: (shouldTransferCommon ? common : 0) + shortage,
    loanOwner: loanOwner
}) {} catch {
    // Note: Safe transfer or supply to a pool can fail. In that case the LOAN token stays in repaid state and
    // waits for the LOAN token owner to claim the repaid credit. Otherwise lender would be able to prevent
    // anybody from repaying the loan.
}
```

However, the `tryClaimRepaidLOAN` function may revert or return early under specific conditions, resulting in the previous loan remaining in a ”paid back” state without indicating that only the surplus amount (excluding the common part) is claimable.

This finding was discovered through a Wake manually-guided fuzzing campaign conducted as part of the audit. See Appendix B for more details.

Exploit scenario

The incorrect optimization allows the withdrawal of the common credit amount twice.

An attacker opens a loan against himself with credit as a valuable token present in the PWN vault. The attacker then transfers the loan token to a new address, effectively changing the loan owner.

The attacker then refinances the loan from the second address. Because of the loan token transfer, the old loan owner (lender) and the new lender are the same, triggering the optimization.

The `tryClaimRepaidLOAN` function is called but returns early without claiming any tokens, leaving the original loan in a ”paid back” state. The early return is due to logic preventing auto-claiming lender’s tokens when the original token source differs from the current lender.

Listing 5. Excerpt from [PWNSimpleLoan.tryClaimRepaidLOAN]

```
if (loan.originalLender != loanOwner)
    return;
```

The attacker's transaction successfully completes, without pulling the common credit amount from the attacker but with marking the original loan as "paid back".

The attacker then calls `claimLOAN` with the original loan, receiving not only the surplus amount but also the common credit amount from the PWN vault.

Alice is the borrower. Bob is a `previous loan` lender. Charlie is a `new loan` lender. Alice, Bob, and Charlie are attacker entities.

1. Alice calls `createLoan` with one ERC-721 token as collateral and 10,000 USDT ERC-20 tokens as credit amount with Bob as a lender with Bob's proposal. This loan is the `previous loan` in our term;
2. Alice got 10,000 of Bob's USDT ERC-20 tokens. The vault stored the ERC-721 token as collateral and minted the LOAN token for Bob;
3. Bob sends the LOAN token to Charlie;
4. Alice calls `createLoan` to refinance the `previous loan` with the same collateral and 11,000 of the USDT ERC-20 token as the credit amount, with Charlie as a lender and Charlie's proposal. This loan is the `new loan` in our term;
5. In the contract of the above operation, `shouldTransferCommon` is false, so it pulls only 1,000 of the USDT ERC-20 token from Charlie. But it does not complete claiming in the `tryClaimRepaidLOAN`, so the `previous loan` is the state "paid back," which means claimLoan is callable;
6. Alice calls `repayLoan` with the `new loan` and returns 11,000 USDT;
7. Charlie (the owner of the LOAN token of the `previous loan`) calls `claimLOAN`

with the `previous loan` and gets 10,000 of the USDT ERC-20 token from the `previous loan`;

8. Charlie (the owner of the `new loan` LOAN token) calls `claimLOAN` with the `new loan` and receives 11000 of the USDT ERC-20 token from the `new loan`.

In conclusion, the attacker used 1 of the ERC-721 tokens and 11,000 USDT ERC-20 tokens and got the same ERC-721 token and 21,000 USDT ERC-20 tokens. Thus, 10,000 of the USDT ERC-20 tokens are stolen from the PWN vault.

Recommendation

Perform an additional pull of the common credit amount from the lender when the optimization is enabled and the `tryClaimRepaidLOAN` function call reverts. Convert all early returns in the `tryClaimRepaidLOAN` function to reverts.

Fix 2.0

The issue was fixed by following the recommendation.

Listing 6. Excerpt from [PWNSimpleLoan._settleLoanRefinance]()

```
    try this.tryClaimRepaidLOAN({
        loanId: refinancingLoanId,
        creditAmount: (shouldTransferCommon ? common : 0) + shortage,
        loanOwner: loanOwner
    }) {} catch {
        // Note: Safe transfer or supply to a pool can fail. In that case the LOAN token stays in repaid state and
        // waits for the LOAN token owner to claim the repaid credit. Otherwise lender would be able to prevent
        // anybody from repaying the loan.

        // Transfer loan common to the Vault if necessary
        // Shortage part is already in the Vault
        if (!shouldTransferCommon) {
            creditHelper.amount = common;
            if (lenderSpec.sourceOfFunds != loanTerms.lender) {
                // Lender is not the source of funds
                // Withdraw credit asset to the lender first
```

```
                _withdrawCreditFromPool(creditHelper, loanTerms,
lenderSpec);
            }
            _pull(creditHelper, loanTerms.lender);
        }
    }
}
```

[Go back to Findings Summary]

M1: Chainlink common denominator bad logic

Medium severity issue

Impact:	Medium	Likelihood:	Medium
Target:	Chainlink.sol	Type:	Logic error

Description

The function `fetchPricesWithCommonDenominator` is responsible for fetching the credit and collateral prices using the same denominator. Each price may be fetched with a different denominator. In such cases, an additional query is executed to fetch the price of one denominator against the other. Subsequent calculations are performed to convert one of the credit or collateral prices to the other denominator.

Listing 7. Excerpt from [Chainlink.fetchPricesWithCommonDenominator]()

```
if (creditDenominator == ChainlinkDenominations.USD) {
    (success, creditPrice, creditPriceDecimals) = convertPriceDenominator({
        feedRegistry: feedRegistry,
        nominatorPrice: creditPrice,
        nominatorDecimals: creditPriceDecimals,
        originalDenominator: creditDenominator,
        newDenominator: collateralDenominator
    });
} else {
    (success, collateralPrice, collateralPriceDecimals) =
convertPriceDenominator({
        feedRegistry: feedRegistry,
        nominatorPrice: collateralPrice,
        nominatorDecimals: collateralPriceDecimals,
        originalDenominator: collateralDenominator,
        newDenominator: collateralDenominator == ChainlinkDenominations.USD
            ? creditDenominator
            : ChainlinkDenominations.ETH
    });
```

When neither of the denominators is USD, the function incorrectly fetches the price of the collateral denominator against ETH. This issue arises when the collateral denominator is also ETH.

This finding was discovered through a Wake manually-guided fuzzing campaign performed as a part of the audit. See Appendix B for more details.

Exploit scenario

A new Chainlink elastic loan is about to be created with CRD as the credit token and COL as the collateral token. CRD only has a feed with BTC as a denominator, while COL only has a feed with ETH as a denominator.

CRD/BTC and COL/ETH prices are fetched. Due to the different denominators, additional logic is executed to convert one of the prices to the common denominator. Due to the flawed logic, a query is made to fetch the price of ETH/ETH, resulting in a revert.

Recommendation

Add an additional if-else branch to correctly select the appropriate common denominator. Consider that ETH/BTC and BTC/ETH feeds may not be available simultaneously.

Fix 2.0

The issue was fixed by re-designing elastic Chainlink loan proposals. The user is now responsible for specifying the correct denominators used to calculate the price.

Go back to Findings Summary

M2: Outdated/reverting Chainlink feed causes DoS

Medium severity issue

Impact:	Medium	Likelihood:	Medium
Target:	Chainlink.sol	Type:	Denial of service

Description

The `fetchPrice` function retrieves the latest price from a Chainlink feed using the feed registry.

Listing 8. Excerpt from [Chainlink]()

```
function fetchPrice(IChainlinkFeedRegistryLike feedRegistry, address asset, address denominator)
    internal
    view
    returns (bool, uint256, uint8)
{
    try feedRegistry.getFeed(asset, denominator) returns (IChainlinkAggregatorLike aggregator) {
        (, int256 price,, uint256 updatedAt,) = aggregator.latestRoundData();
        if (price < 0) {
            revert ChainlinkFeedReturnedNegativePrice({ asset: asset, denominator: denominator, price: price });
        }
        if (block.timestamp - updatedAt > MAX_CHAINLINK_FEED_PRICE_AGE) {
            revert ChainlinkFeedPriceTooOld({ asset: asset, updatedAt: updatedAt });
        }

        uint8 decimals = aggregator.decimals();
        return (true, uint256(price), decimals);
    } catch {
        return (false, 0, 0);
    }
}
```

The function is expected to return `false` if the price feed could not be found, allowing the logic to try a different denominator (one of USD, BTC, ETH). However, if either of `aggregator.latestRoundData()` or `aggregator.decimals()` calls reverts, the returned price is negative or the feed is too old, the whole execution reverts.

This vulnerability was identified through a Wake manually-guided fuzzing campaign during the audit. For detailed information, refer to Appendix B.

Exploit scenario

One of token feeds with USD as a denominator becomes stale. The feed against BTC is being updated, but due to the reverting logic, the BTC feed is not being used. This causes denial of service for the token.

Recommendation

Return `false` if either of the external calls reverts, or the returned price is negative or the feed is too old.

Fix 2.0

The issue was fixed by re-designing elastic Chainlink loan proposals. The user is now responsible for specifying the correct denominators used to calculate the price. The contract implementation no longer tries to find the correct denominator.

Go back to Findings Summary

M3: Non-upgradable base contracts

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	PWNConfig.sol	Type:	Storage clashes

Description

The `PWNConfig` contract inherits from non-upgradeable implementations of the `Initializable` and `Ownable2Step` contracts. These non-upgradeable implementations lack storage gap protection, requiring strict adherence to the current storage layout in future `PWNConfig` contract revisions.

wake print storage-layout

PWNConfig storage layout

Slot	Offset	Name	Type	Contract
0	0	_owner	address	Ownable
1	0	_pendingOwner	address	Ownable2Step
	20	_initialized	uint8	Initializable
	21	_initializing	bool	Initializable
	22	fee	uint16	PWNConfig
2	0	feeCollector	address	PWNConfig
3	0	_loanMetadataUri	mapping(address => string)	PWNConfig
4	0	_sfComputerRegistry	mapping(address => address)	PWNConfig
5	0	_poolAdapterRegistry	mapping(address => address)	PWNConfig

Figure 1. PWNConfig Storage Layout

Exploit scenario

Alice deploys a new version of the `PWNConfig` contract where the `Ownable2Step` base contract contains an additional storage variable. Due to this modification, the `_initialized` variable shifts to a different storage slot. Bob exploits this storage collision to reinitialize the contract and claim ownership.

Recommendation

Consider re-deploying a new instance of the `PWNConfig` contract with a new proxy and upgradable base contracts. If not possible, ensure the storage layout is strictly followed in the next `PWNConfig` revisions.

Fix 2.0

The `PWNConfig` contract now inherits from the `Ownable2StepUpgradeable` contract.

[Go back to Findings Summary]

M4: Incorrect EIP-712 typehash

Medium severity issue

Impact:	Medium	Likelihood:	Medium
Target:	core/src/loan/terms/simple/proposal/*.sol	Type:	Standards violation

Description

The protocol implements multiple loan proposal types, each containing a `Proposal` structure that defines proposal parameters. These proposals can be signed by lenders or borrowers using [EIP-712](EIP-712).

Listing 9. Excerpt from PWNSimpleLoanSimpleProposal

```
struct Proposal {
    MultiToken.Category collateralCategory;
    address collateralAddress;
    uint256 collateralId;
    uint256 collateralAmount;
    bool checkCollateralStateFingerprint;
    bytes32 collateralStateFingerprint;
    address creditAddress;
    uint256 creditAmount;
    uint256 availableCreditLimit;
    bytes32 utilizedCreditId;
    uint256 fixedInterestAmount;
    uint24 accruingInterestAPR;
    uint32 durationOrDate;
    uint40 expiration;
    address allowedAcceptor;
    address proposer;
    bytes32 proposerSpecHash;
    bool isOffer;
    uint256 refinancingLoanId;
    uint256 nonceSpace;
    uint256 nonce;
    address loanContract;
}
```

A type mismatch exists between the `accruingInterestAPR` parameter definitions:

- `Proposal` structure: uint24
- EIP-712 typehash: uint40

Listing 10. Excerpt from [PWNSimpleLoanSimpleProposal]()

```
bytes32 public constant PROPOSAL_TYPEHASH = keccak256(
    "Proposal(uint8 collateralCategory,address collateralAddress,uint256 collateralId,uint256 collateralAmount,bool checkCollateralStateFingerprint,bytes32 collateralStateFingerprint,address creditAddress,uint256 creditAmount,uint256 availableCreditLimit,bytes32 utilizedCreditId,uint256 fixedInterestAmount,uint40 accruingInterestAPR,uint32 durationOrDate,uint40 expiration,address allowedAcceptor,address proposer,bytes32 proposerSpecHash,bool isOffer,uint256 refinancingLoanId,uint256 nonceSpace,uint256 nonce,address loanContract)"
);
```

This inconsistency affects the following contract files:

- PWNSimpleLoanDutchAuctionProposal.sol
- PWNSimpleLoanElasticChainlinkProposal.sol
- PWNSimpleLoanElasticProposal.sol
- PWNSimpleLoanListProposal.sol
- PWNSimpleLoanSimpleProposal.sol

This finding was identified by a Wake static analysis detector. For additional details, refer to Appendix B.

Exploit scenario

Due to the incorrect typehash, the EIP-712 signature verification will fail for all signatures generated with the correct `accruingInterestAPR` type. This will result in a disability of accepting off-chain lending proposals.

Recommendation

Unify the `accruingInterestAPR` type in the `Proposal` structure and the [EIP-712](https://eips.ethereum.org/EIPS/eip-712) type hash for all the aforementioned contracts.

Fix 2.0

The issue was fixed by using `uint24` for the `accruingInterestAPR` parameter in the EIP-712 type hashes.

Go back to Findings Summary

L1: Decimal detection may lead to unexpected reverts

Low severity issue

Impact:	Medium	Likelihood:	Low
Target:	safeFetchDecimals.sol	Type:	Logic error

Description

The `safeFetchDecimals` function executes a static call to the target contract using the `decimals()` function signature. The function reverts if the target contract does not implement the `decimals()` function interface.

Listing 11. Excerpt from [safeFetchDecimals]()

```
function safeFetchDecimals(address asset) view returns (uint256) {
    bytes memory rawDecimals = Address.functionStaticCall(asset,
abi.encodeWithSignature("decimals()"));
    if (rawDecimals.length == 0) {
        return 0;
    }
    return abi.decode(rawDecimals, (uint256));
}
```

Exploit scenario

If the `safeFetchDecimals` function is ever used with a non-ERC-20 token (or non-standard ERC-20 token), it will revert the execution instead of returning `0`.

Recommendation

Consider returning `0` even in the case of an unsuccessful low-level static call.

Fix 2.0

The `safeFetchDecimals` function now returns `0` even if the low-level static call fails.

Go back to Findings Summary

W1: Older versions of Aave and Compound not supported

Impact:	Warning	Likelihood:	N/A
Target:	N/A	Type:	Code quality

Description

The project allows to withdraw credit from Aave or Compound when a new loan is being created. However, only the latest versions of Aave (v3) and Compound (v3) are supported.

Recommendation

Consider adding support for older versions of Aave and Compound protocols.

Acknowledgment 2.0

The client acknowledged the finding as being intended by design.

Go back to Findings Summary

W2: `creditPerCollateralUnit` division by zero

Impact:	Warning	Likelihood:	N/A
Target:	N/A	Type:	Data validation

Description

The `creditPerCollateralUnit` variable specifies the exchange rate between collateral and credit units. When this variable is set to zero, the operation results in a division by zero error without providing an appropriate error message to the user.

Recommendation

Consider raising a custom user-defined error when `creditPerCollateralUnit` is zero.

Fix 2.0

The contract logic now reverts with a new `ZeroCreditPerCollateralUnit` error if the `creditPerCollateralUnit` is zero.

Go back to Findings Summary

W3: `_checkTransfer` sender and receiver collision

Impact:	Warning	Likelihood:	N/A
Target:	PWNVault.sol	Type:	Data validation

Description

The `_checkTransfer` function validates token transfers by verifying either the sender’s negative balance difference or the receiver’s positive balance difference.

Listing 12. Excerpt from [PWNVault]()

```
function _checkTransfer(
    MultiToken.Asset memory asset,
    uint256 originalBalance,
    address checkedAddress,
    bool checkIncreasingBalance
) private view {
    uint256 expectedBalance = checkIncreasingBalance
        ? originalBalance + asset.getTransferAmount()
        : originalBalance - asset.getTransferAmount();

    if (expectedBalance != asset.balanceOf(checkedAddress)) {
        revert IncompleteTransfer();
    }
}
```

The validation logic fails to handle scenarios where the sending address matches the receiving address.

Recommendation

Either revert with a different error message if the sending source is the same as the receiver address, or allow such a case and adjust the logic to handle it.

Fix 2.0

The `_checkTransfer` function now reverts with a custom user-defined error if the sender and receiver are the same.

[Go back to Findings Summary]

I1: revokeNonces nonce space can be cached

Impact:	Info	Likelihood:	N/A
Target:	PWNRevokedNonce.sol	Type:	Gas optimization

Description

The `revokeNonces` function performs redundant storage reads when revoking multiple nonces in the current nonce space:

Listing 13. Excerpt from [PWNRevokedNonce]()

```
function revokeNonces(uint256[] calldata nonces) external {
    for (uint256 i; i < nonces.length; ++i) {
        _revokeNonce(msg.sender, _nonceSpace[msg.sender], nonces[i]);
    }
}
```

While the nonce space remains constant throughout the function execution, the Solidity optimizer may not consolidate multiple storage reads into a single operation, resulting in unnecessary gas costs.

Recommendation

Load the `_nonceSpace[msg.sender]` value before the loop and use it instead of reading from the storage in each iteration.

Fix 2.0

The inefficiency was fixed by caching the nonce space before the loop.

[Go back to Findings Summary]()

I2: `LoanDefaulted(uint40)` error parameter not named

Impact:	Info	Likelihood:	N/A
Target:	PWNSimpleLoan.sol	Type:	Code quality

Description

`LoanDefaulted(uint40)` is the only error without a parameter name.

Recommendation

Consider adding a variable name to the error for better readability.

Fix 2.0

The `LoanDefaulted` parameter was named `timestamp`.

Go back to Findings Summary

Report Revision 2.0

Revision Team

Member's Name	Position
Michal Převrátil	Lead Auditor
Naoki Yoshida	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

System Overview

Elastic Chainlink loan proposals were reworked, requiring the user to specify the price denominators used to compute the price of collateral to the credit. The code no longer attempts to find the denominators automatically.

Except for the elastic Chainlink proposals, only direct fixes of the previously discovered issues were made to the codebase without any other significant changes.

Fuzzing

The manually guided differential stateful fuzz test created in the previous revision was updated in accordance with the changes made to the codebase.

The fuzz test discovered a new M5 issue.

The full source code of the updated fuzz test is available at https://github.com/Ackee-Blockchain/tests-pwn-protocol/tree/revision-2.0.

Findings

The following section presents the list of findings discovered in this revision.

M5: Incorrect EIP-712 data encoding

Medium severity issue

Impact:	Medium	Likelihood:	Medium
Target:	PWNSimpleLoanElasticProposal.sol	Type:	Standards violation

Description

The protocol implements multiple loan proposal types, each containing a `Proposal` structure that defines proposal parameters. These proposals can be signed by lenders or borrowers using [EIP-712](EIP-712).

Listing 14. Excerpt from PWNSimpleLoanElasticChainlinkProposal

```
struct Proposal {
    MultiToken.Category collateralCategory;
    address collateralAddress;
    uint256 collateralId;
    bool checkCollateralStateFingerprint;
    bytes32 collateralStateFingerprint;
    address creditAddress;
    address[] feedIntermediaryDenominations;
    bool[] feedInvertFlags;
    uint256 loanToValue;
    uint256 minCreditAmount;
    uint256 availableCreditLimit;
    bytes32 utilizedCreditId;
    uint256 fixedInterestAmount;
    uint24 accruingInterestAPR;
    uint32 durationOrDate;
    uint40 expiration;
    address allowedAcceptor;
    address proposer;
    bytes32 proposerSpecHash;
    bool isOffer;
    uint256 refinancingLoanId;
    uint256 nonceSpace;
    uint256 nonce;
    address loanContract;
```

```
}
```

The `Proposal` structure in the `PWNSimpleLoanElasticChainlinkProposal` contract contains `feedIntermediaryDenominations` and `feedInvertFlags` array members. The implementation uses `abi.encode()` for encoding the entire structure, which results in incorrect encoding of array members as specified in [EIP-712](EIP-712).

Listing 15. Excerpt from PWNSimpleLoanElasticChainlinkProposal

```
function getProposalHash(Proposal calldata proposal) public view returns
(bytes32) {
    return _getProposalHash(PROPOSAL_TYPEHASH, abi.encode(proposal));
}
```

This issue was identified through a Wake manually-guided fuzzing campaign during the audit. For detailed information, refer to Appendix B.

Exploit scenario

Due to the incorrect data encoding, all correct signatures will be rejected and it will be impossible to accept elastic Chainlink loan proposals.

Recommendation

Replace `abi.encode()` with concatenation of encoding of each member of the `Proposal` structure. Use `keccak256(abi.encodePacked(array))` for array members.

Fix 2.1

The issue was fixed by following the recommendation.

Go back to Findings Summary

Report Revision 2.1

Revision Team

Member’s Name	Position
Michal Převrátil	Lead Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

Overview

Since there were no comprehensive changes in this revision, the complete overview is listed in the Executive Summary section Revision 2.1.

Appendix A: How to cite

Please cite this document as:

Ackee Blockchain Security, PWN: Protocol, 20.12.2024.

Appendix B: Wake Findings

This section lists the outputs from the Wake framework used for testing and static analysis during the audit.

B.1. Fuzzing

The following table lists all implemented execution flows in the Wake fuzzing framework.

ID	Flow	Added
F1	Setting of protocol fee	1.0
F2	Setting of protocol fee collector	1.0
F3	Making new on-chain simple proposal	1.0
F4	Accepting simple proposal, including refinancing	1.0
F5	Making new on-chain dutch auction proposal	1.0
F6	Accepting dutch auction proposal, including refinancing	1.0
F7	Making new on-chain elastic proposal	1.0
F8	Accepting elastic proposal, including refinancing	1.0
F9	Making new on-chain elastic Chainlink proposal	1.0
F10	Accepting elastic Chainlink proposal, including refinancing	1.0
F11	Making new on-chain list proposal	1.0
F12	Accepting list proposal, including refinancing	1.0
F13	Repayment of loan	1.0
F14	Claiming repaid loan	1.0
F15	Claiming expired loan	1.0
F16	Transferring LOAN token	1.0

ID	Flow	Added
F17	Making on-chain extension proposal	1.0
F18	Extending loan	1.0
F19	Revoking nonce	1.0
F20	Revoking nonce space	1.0

Table 4. Wake fuzzing flows

The following table lists the invariants checked after each flow.

ID	Invariant	Added	Status
IV1	Transactions do not revert except where explicitly expected	1.0	**Fail** (M1, M2, M5, W2, W3)
IV2	Balances of all ERC-20 tokens match expected value for all important accounts	1.0	**Fail** (C2)
IV3	Owners of all ERC-721 tokens match expected values	1.0	Success
IV4	Balances of all ERC-1155 tokens match expected values for all important accounts	1.0	Success

Table 5. Wake fuzzing invariants

B.2. Detectors

This section contains vulnerability and code quality detections from the Wake tool.

```
wake detect reentrancy

[HIGH][LOW] Possible reentrancy in `PWNVault._tryPermit(struct Permit)` [reentrancy]
  180      */
  181     function _tryPermit(Permit memory permit) internal {
  182         if (permit.asset != address(0)) {
❱ 183             try IERC20Permit(permit.asset).permit({
  184                 owner: permit.owner,
  185                 spender: address(this),
  186                 value: permit.amount,
src/loan/vault/PWNVault.sol
├── Exploitable from `PWNSimpleLoan.createLOAN(struct PWNSimpleLoan.ProposalSpec,struct PWNSimpleLoan.Le
│     437             if (callerSpec.permitData.length > 0) {
│     438                 Permit memory permit = abi.decode(callerSpec.permitData, (Permit));
│     439                 _checkPermit(msg.sender, loanTerms.credit.assetAddress, permit);
│   ❱ 440                 _tryPermit(permit);
│     441             }
│     442
│     443             // Settle the loan
│   src/loan/terms/simple/loan/PWNSimpleLoan.sol
├── Exploitable from `PWNSimpleLoan.repayLOAN(uint256,bytes)`
│     703                 Permit memory permit = abi.decode(permitData, (Permit));
│     704                 _checkPermit(msg.sender, loan.creditAddress, permit);
│     705                 // wake-disable-next-line reentrancy
│   ❱ 706                 _tryPermit(permit);
│     707             }
│     708
│     709             // Transfer the repaid credit to the Vault
│   src/loan/terms/simple/loan/PWNSimpleLoan.sol
└── Exploitable from `PWNSimpleLoan.extendLOAN(struct PWNSimpleLoan.ExtensionProposal,bytes,bytes)`
      1040             if (permitData.length > 0) {
      1041                 Permit memory permit = abi.decode(permitData, (Permit));
      1042                 _checkPermit(msg.sender, extension.compensationAddress, permit);
    ❱ 1043                 _tryPermit(permit);
      1044             }
      1045             _pushFrom(compensation, loan.borrower, loanOwner);
      1046         }
    src/loan/terms/simple/loan/PWNSimpleLoan.sol
```

Figure 2. Reentrancy detection

wake detect eip-712-typehash-mismatch

```
[MEDIUM][LOW] Structure used in typehash does not match any structure definition in codebase [eip-712-typehash-mismatch]
    /**
     * @dev EIP-712 simple proposal struct type hash.
     */
❯   bytes32 public constant PROPOSAL_TYPEHASH = keccak256(
        "Proposal(uint8 collateralCategory,address collateralAddress,uint256 collateralId,uint256 collateralAmount,b
    );
src/loan/terms/simple/proposal/PWNSimpleLoanDutchAuctionProposal.sol

[MEDIUM][LOW] Structure used in typehash does not match any structure definition in codebase [eip-712-typehash-mismatch]
    /**
     * @dev EIP-712 simple proposal struct type hash.
     */
❯   bytes32 public constant PROPOSAL_TYPEHASH = keccak256(
        "Proposal(uint8 collateralCategory,address collateralAddress,uint256 collateralId,bool checkCollateralStateF
    );
src/loan/terms/simple/proposal/PWNSimpleLoanElasticChainlinkProposal.sol

[MEDIUM][LOW] Structure used in typehash does not match any structure definition in codebase [eip-712-typehash-mismatch]
    /**
     * @dev EIP-712 simple proposal struct type hash.
     */
❯   bytes32 public constant PROPOSAL_TYPEHASH = keccak256(
        "Proposal(uint8 collateralCategory,address collateralAddress,uint256 collateralId,bool checkCollateralStateF
    );
src/loan/terms/simple/proposal/PWNSimpleLoanElasticProposal.sol

[MEDIUM][LOW] Structure used in typehash does not match any structure definition in codebase [eip-712-typehash-mismatch]
    /**
     * @dev EIP-712 simple proposal struct type hash.
     */
❯   bytes32 public constant PROPOSAL_TYPEHASH = keccak256(
        "Proposal(uint8 collateralCategory,address collateralAddress,bytes32 collateralIdsWhitelistMerkleRoot,uint25
    );
src/loan/terms/simple/proposal/PWNSimpleLoanListProposal.sol

[MEDIUM][LOW] Structure used in typehash does not match any structure definition in codebase [eip-712-typehash-mismatch]
    /**
     * @dev EIP-712 simple proposal struct type hash.
     */
❯   bytes32 public constant PROPOSAL_TYPEHASH = keccak256(
        "Proposal(uint8 collateralCategory,address collateralAddress,uint256 collateralId,uint256 collateralAmount,b
    );
src/loan/terms/simple/proposal/PWNSimpleLoanSimpleProposal.sol
```

Figure 3. EIP-712 typehash mismatch detections

B.3. Other assets

This section contains assets generated by the Wake tool used during the audit.

Figure 4. C1 vulnerability attack call trace

ackee
blockchain security

Thank You

Ackee Blockchain a.s.

Rohanske nabrezi 717/4
186 00 Prague
Czech Republic

hello@ackee.xyz