
Security Review Report

NM-0074 PWN



NETHERMIND

(Mar 27, 2023)



Contents

1	Executive Summary	2
2	Audited Files	3
3	Assumptions	3
4	Summary of Issues	3
5	System Overview	4
5.1	lib/MultiToken/MultiToken.sol	4
5.2	Folder src/config	5
5.3	Folder src/deployer	5
5.4	Folder src/hub	6
5.4.1	PWNHubAccessControl.sol	6
5.4.2	PWNHubTags.sol	6
5.4.3	PWNHub.sol	6
5.5	Folder src/loan	7
5.5.1	PWNVault.sol	7
5.5.2	lib/PWNFeeCalculator.sol	8
5.5.3	lib/PWNSignatureChecker.sol	8
5.5.4	token/PWNLOAN.sol	8
5.5.5	terms/PWNLOANTerms.sol	9
5.5.6	terms/simple/loan/PWNSimpleLoan.sol	9
5.5.7	terms/simple/factory/request/PWNSimpleLoanSimpleRequest.sol	10
5.5.8	terms/simple/factory/request/base/PWNSimpleLoanRequest.sol	11
5.5.9	terms/simple/factory/offer/PWNSimpleLoanListOffer.sol	11
5.5.10	terms/simple/factory/offer/PWNSimpleLoanSimpleOffer.sol	12
5.5.11	terms/simple/factory/offer/base/PWNSimpleLoanOffer.sol	13
5.6	Folder src/nonce	13
6	Risk Rating Methodology	15
7	Issues	16
7.1	[High] "No Revert on Failure" tokens can be used to steal from lender	16
7.2	[High] Borrower can steal lender's assets by using WETH as collateral	16
7.3	[Medium] Borrower can prevent the lender from receiving collateral during a claim	17
7.4	[Medium] Fee-on-transfer tokens can be locked in the vault	18
7.5	[Medium] Late repayment feature could be abused by lenders to trick borrowers into losing their collateral	18
7.6	[Medium] Some common ERC20 tokens can't be used in the protocol	19
7.7	[Medium] CryptoKitties token is locked when using it as ERC721 type collateral	19
7.8	[Low] Function calculateFeeAmount(...) has unnecessary loss of precision	20
7.9	[Low] Malicious owner can arbitrarily change the fee to any value	20
7.10	[Low] Ownable contracts	21
7.11	[Low] offer.duration can be set to zero	21
7.12	[Info] Fee Collector can be set to address(0x0) in function setFeeCollector(...)	22
7.13	[Info] Token that reverts on zero-value transfers can cause createLoan(...) to fail when transferring fees	22
7.14	[Best Practices] UnlockedPragma	22
8	Documentation Evaluation	23
8.1	Documentation inconsistencies	23
8.1.1	Documentation about ERC721 does not match implementation	23
8.1.2	Documentation about onERC1155BatchReceived(...) does not match implementation	23
9	Test Suite Evaluation	24
9.1	Contracts Compilation Output	24
9.2	Tests Output	24
9.3	Code Coverage	30
9.4	Slither	30
10	About Nethermind	31

1 Executive Summary

This document presents the security review performed by [Nethermind](#) on the [PWN Contracts](#). PWN Finance is a DeFi solution for lending and borrowing tokens following ERC20, ERC721, ERC1155, and CryptoKitties standard. The protocol has an offer/request mechanism created off-chain to be fulfilled by a borrower/lender. PWN solves the possibility of under-collateralizing assets through a fixed amount of return of assets specified when the offer/request is created. PWN uses the MultiToken to interact with multiple types of assets in a single interface, providing functionalities such as transferring assets, approving allowance, and checking balances of a particular asset based on a certain category.

The audited code consists of 1,037 lines of Solidity with code coverage of 92.3%. The PWN team provided the [developer docs](#), which contains most of the info required for understanding the system. **The audit was performed using** (a) manual analysis of the codebase, (b) automated analysis tools, and (c) simulation of the smart contracts. **Along this document, we report** 14 points of attention, where two are classified as High, five are classified as Medium, four are classified as Low, and three are classified as Informational or Best Practices. The issues are summarized in Fig. 1.

This document is organized as follows. Section 2 presents the files in the scope of this audit. Section 3 presents the assumptions for this audit. Section 4 summarizes the issues. Section 5 presents the system overview. Section 6 discusses the risk rating methodology adopted for this audit. Section 7 details the issues. Section 8 discusses the documentation provided by the client for this audit. Section 9 presents the compilation, tests, coverage, and automated tests. Section 10 concludes the document.

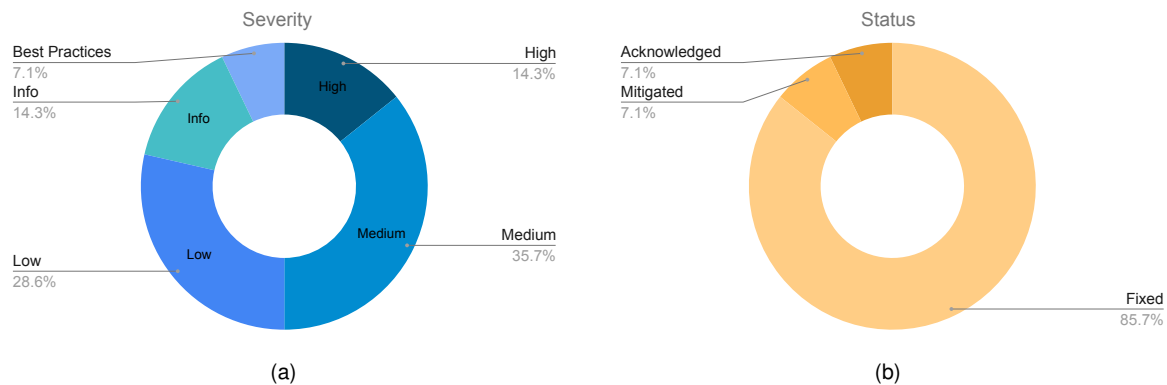


Fig. 1: Distribution of issues: Critical (0), High (2), Medium (5), Low (4), Undetermined (0), Informational (2), Best Practices (1).
Distribution of status: Fixed (12), Acknowledged (1), Mitigated (1), Unresolved (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	Mar. 13, 2023
Response from Client	Mar. 20, 2023
Final Report	Mar. 27, 2023
Methods	Manual Review, Automated Analysis
Repository	PWN
Commit Hash (Initial Audit)	455e8f7d5ab9d284eeb3f9207c6021c389a794e2
Commit Hash (Reaudit)	13c67a5c743e50c30f97c540111fd67692d7eb
Documentation	PWN Developer Docs
Documentation Assessment	Medium
Test Suite Assessment	High

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	lib/MultiToken/MultiToken.sol	154	134	87.0%	51	339
2	src/PWNErrors.sol	21	11	52.4%	11	43
3	src/hub/PWNHub.sol	25	46	184.0%	22	93
4	src/hub/PWNHubTags.sol	8	5	62.5%	6	19
5	src/hub/PWNHubAccessControl.sol	20	14	70.0%	13	47
6	src/loan/PWNVault.sol	63	94	149.2%	28	185
7	src/loan/lib/PWNFeeCalculator.sol	11	12	109.1%	6	29
8	src/loan/lib/PWNSignatureChecker.sol	46	19	41.3%	11	76
9	src/loan/token/IERC5646.sol	4	9	225.0%	3	16
10	src/loan/token/IPWNLoanMetadataProvider.sol	4	10	250.0%	3	17
11	src/loan/token/PWNLOAN.sol	41	63	153.7%	33	137
12	src/loan/terms/PWNLoanTerms.sol	13	12	92.3%	5	30
13	src/loan/terms/simple/loan/PWNSimpleLoan.sol	159	133	83.6%	74	366
14	src/loan/terms/simple/factory/IPWNSimpleLoanTermsFactory.sol	9	13	144.4%	5	27
15	src/loan/terms/simple/factory/request/PWNSimpleLoanSimpleRequest.sol	95	65	68.4%	33	193
16	src/loan/terms/simple/factory/request/base/PWNSimpleLoanRequest.sol	22	33	150.0%	18	73
17	src/loan/terms/simple/factory/offer/PWNSimpleLoanListOffer.sol	111	78	70.3%	37	226
18	src/loan/terms/simple/factory/offer/PWNSimpleLoanSimpleOffer.sol	97	67	69.1%	33	197
19	src/loan/terms/simple/factory/offer/base/PWNSimpleLoanOffer.sol	22	33	150.0%	18	73
20	src/nonce/PWNRevokedNonce.sol	35	66	188.6%	31	132
21	src/deployer/PWNDeployer.sol	18	35	194.4%	13	66
22	src/deployer/PWNContractDeployerSalt.sol	14	5	35.7%	8	27
23	src/config/PWNConfig.sol	45	65	144.4%	31	141
	Total	1037	1022	98.6%	493	2552

3 Assumptions

The prepared security review is based on the following assumptions:

- Only limited functionality of the library MultiToken.sol is used within the protocol;
- Offers and Request made off-chain by the Lenders and Borrowers have not been tampered with;
- The Merkle tree inclusion data is stored in a backend.

4 Summary of Issues

	Finding	Severity	Update
1	"No Revert on Failure" tokens can be used to steal from lender	High	Fixed
2	Borrower can steal lender's assets by using WETH as collateral	High	Fixed
3	Borrower can prevent the lender from receiving collateral during a claim	Medium	Fixed
4	Fee-on-transfer tokens can be locked in the vault	Medium	Fixed
5	Late repayment feature could be abused by lenders to trick borrowers into losing their collateral	Medium	Fixed
6	Some common ERC20 tokens can't be used in the protocol	Medium	Fixed
7	CryptoKitties token is locked when using it as ERC721 type collateral	Medium	Fixed
8	Function calculateFeeAmount(...) has unnecessary loss of precision	Low	Fixed
9	Malicious owner can arbitrarily change the fee to any value	Low	Mitigated
10	Ownable contracts	Low	Fixed
11	offer.duration can be set to zero	Low	Fixed
12	Fee Collector can be set to address(0x0) in function setFeeCollector(...)	Info	Fixed
13	Token that reverts on zero-value transfers can cause createLoan(...) to fail when transferring fees	Info	Fixed
14	Unlocked Pragma	Best Practices	Acknowledged

5 System Overview

PWN is a hub for peer-to-peer (P2P) loans backed by digital assets (DAI, LINK, APE, NFTs, and others) as collateral, and receive any ERC20 assets for the same. The borrower has the flexibility to provide any standard ERC20, ERC721, ERC1155 and CryptoKitties token. The lender can earn interest on the asset provided. The codebase is organized in the folders, as shown in Fig. 2.

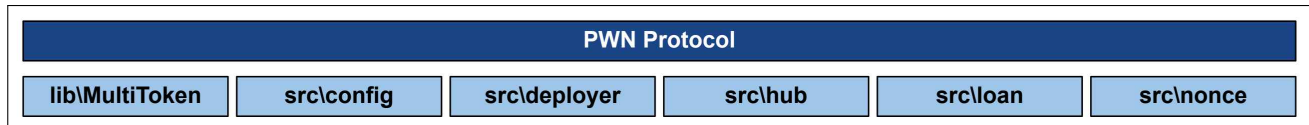


Fig. 2: Folders composing the protocol

5.1 lib/MultiToken/MultiToken.sol

This library provides functionalities for transferring assets of different categories. The library has the following imports.

```

1 import "openzeppelin-contracts/contracts/interfaces/IERC20.sol";
2 import "openzeppelin-contracts/contracts/interfaces/IERC721.sol";
3 import "openzeppelin-contracts/contracts/interfaces/IERC1155.sol";
4 import "openzeppelin-contracts/contracts/token/ERC20/extensions/draft-IERC20Permit.sol";
5 import "../interfaces/ICryptoKitties.sol";
  
```

Thus, we can notice that the protocol supports ERC20, ERC721, ERC1155 and CryptoKitties tokens, as illustrated in the enum Category reproduced below.

```

1 enum Category {
2     ERC20,
3     ERC721,
4     ERC1155,
5     CryptoKitties
6 }
  
```

Each asset is stored in the structure Asset reproduced below. The field category indicates the asset category according to the enum Category. The field assetAddress indicates the contract address defining the asset. The field id indicates the TokenID when the asset is an NFT. The field amount indicates the number of fungible tokens.

```

1 struct Asset {
2     Category category;
3     address assetAddress;
4     uint256 id;
5     uint256 amount;
6 }
  
```

The transfer functionalities are implemented using the six functions listed below.

```

1 function transferAssetFrom(Asset memory asset, address source, address dest) internal;
2
3 function safeTransferAssetFrom(Asset memory asset, address source, address dest) internal;
4
5 function _transferAssetFrom(Asset memory asset, address source, address dest, bool isSafe) internal;
6
7 function transferAssetFromCalldata(Asset memory asset, address source, address dest, bool fromSender) pure internal
8     → returns (bytes memory);
9
10 function safeTransferAssetFromCalldata(Asset memory asset, address source, address dest, bool fromSender) pure internal
11     → returns (bytes memory);
12
13 function _transferAssetFromCalldata(Asset memory asset, address source, address dest, bool fromSender, bool isSafe)
14     → pure private returns (bytes memory);
  
```

The library also implements one function for granting approval via permit signature, which can be standard (65 bytes) or compact (64 bytes) defined in the EIP-2098.

```

1 function permit(Asset memory asset, address owner, address spender, bytes memory permitData) internal;
  
```

The balance on various token interfaces can be queried using the function `balanceOf(...)`.

```
1 function balanceOf(Asset memory asset, address target) internal view returns (uint256);
```

The tokens' approval can be performed using the function `approveAsset(...)`.

```
1 function approveAsset(Asset memory asset, address target) internal;
```

Finally, the library also has two helper functions. The function `isValid(...)` returns if a given structure `Asset` is valid or not, while the function `isSameAs(...)` returns if two assets are equal.

```
1 function isValid(Asset memory asset) internal pure returns (bool);
2
3 function isSameAs(Asset memory asset, Asset memory otherAsset) internal pure returns (bool);
```

5.2 Folder `src/config`

This folder holds the contract `PWNConfig.sol`, which provides configurable values for the PWN protocol. The contract has the following imports.

```
1 import "openzeppelin-contracts/contracts/access/Ownable.sol";
2 import "openzeppelin-contracts/contracts/proxy/utils/Initializable.sol";
```

The contract has three state variables. The fee is stored as an `uint16` having the precision of two decimal digits, i.e., the value 100 indicates 1% of fee. All the fees collected by the protocol are sent to the `feeCollector` address. The contract also stores one mapping that relates the loan contract address to the loan token metadata uri. These state variables are reproduced below.

```
1 uint16 public fee;
2 address public feeCollector;
3 mapping (address => string) public loanMetadataUri;
```

The contract provides initialization and re-initialization functionalities. Fees management is performed using the function `setFee(...)`, while the fees collector is set using the function `setFeeCollector(...)`. Both functions can only be called by the contract owner.

```
1 function setFee(uint16 _fee) external onlyOwner;
2
3 function _setFee(uint16 _fee) private;
4
5 function setFeeCollector(address _feeCollector) external onlyOwner;
6
7 function _setFeeCollector(address _feeCollector) private;
```

The contract also has the function `setLoanMetadataUri(...)`, which assigns a `metadataUri` to a `loanContract`, only callable by the contract owner.

```
1 function setLoanMetadataUri(address loanContract, string memory metadataUri) external onlyOwner;
```

5.3 Folder `src/deployer`

This folder has two contracts: a) `PWNContractDeployerSalt.sol`; and, b) `PWNDeployer.sol`. The contract `PWNContractDeployerSalt.sol` is only a placeholder for constants used by the PWN protocol, while the contract `PWNDeployer.sol` presents the logic for deploying a release of the protocol. The contract `PWNDeployer.sol` has the following imports.

```
1 import "openzeppelin-contracts/contracts/access/Ownable.sol";
2 import "openzeppelin-contracts/contracts/utils/Create2.sol";
```

The contract offers three functions. The function `deploy(...)` receives the bytecode and the salt for deploying a new contract at a deterministic address. The address where the contract will be deployed can be queried using the function `computeAddress(...)`. In addition, the contract also offers the function `deployAndTransferOwnership(...)` that performs the deployment and transfers the ownership to the owner.

```
1 function deploy(bytes32 salt, bytes memory bytecode) external onlyOwner returns (address);
2
3 function deployAndTransferOwnership(bytes32 salt, address owner, bytes memory bytecode) external onlyOwner returns
  ↳ (address deployedContract);
4
5 function computeAddress(bytes32 salt, bytes32 bytecodeHash) external view returns (address);
```

5.4 Folder src/hub

This folder has three contracts. The contract PWNHubTags.sol is just a placeholder for constants. The contract PWNHub.sol connects the PWN contracts into the protocol. The contract PWNHubAccessControl.sol is an abstract contract that implements access control modifiers for the PWN Hub.

5.4.1 PWNHubAccessControl.sol

The abstract contract PWNHubAccessControl.sol has the following imports.

```

1 import "@pwn/hub/PWNHub.sol";
2 import "@pwn/hub/PWNHubTags.sol";
3 import "@pwn/PWNErrors.sol";

```

The contract has only one storage variable, which is immutable. This variable is set in the constructor(...) to the address of the contract PWNHub.sol.

```

1 PWNHub immutable internal hub;
2
3 constructor(address pwnHub) {
4     hub = PWNHub(pwnHub);
5 }

```

The contract also has two modifiers. The modifier onlyActiveLoan() checks if msg.sender has the tag PWNHubTags.ACTIVE_LOAN, while the modifier onlyWithTag(bytes32 tag) checks if msg.sender has the tag specified as an argument. Both modifiers are presented below.

```

1 modifier onlyActiveLoan() {
2     if (hub.hasTag(msg.sender, PWNHubTags.ACTIVE_LOAN) == false)
3         revert CallerMissingHubTag(PWNHubTags.ACTIVE_LOAN);
4     _;
5 }

```

```

1 modifier onlyWithTag(bytes32 tag) {
2     if (hub.hasTag(msg.sender, tag) == false)
3         revert CallerMissingHubTag(tag);
4     _;
5 }

```

5.4.2 PWNHubTags.sol

The list of tags can be found in the library PWNHubTags.sol. The tags are: ACTIVE_LOAN, SIMPLE_LOAN_TERMS_FACTORY, LOAN_REQUEST, LOAN_OFFER, as reproduced below.

```

1 library PWNHubTags {
2     string internal constant VERSION = "1.0";
3
4     /// @dev Address can mint LOAN tokens and create LOANs via loan factory contracts.
5     bytes32 internal constant ACTIVE_LOAN = keccak256("PWN_ACTIVE_LOAN");
6
7     /// @dev Address can be used as a loan terms factory for creating simple loans.
8     bytes32 internal constant SIMPLE_LOAN_TERMS_FACTORY = keccak256("PWN_SIMPLE_LOAN_TERMS_FACTORY");
9
10    /// @dev Address can revoke loan request nonces.
11    bytes32 internal constant LOAN_REQUEST = keccak256("PWN_LOAN_REQUEST");
12    /// @dev Address can revoke loan offer nonces.
13    bytes32 internal constant LOAN_OFFER = keccak256("PWN_LOAN_OFFER");
14 }

```

5.4.3 PWNHub.sol

The contract PWNHub.sol has the following imports:

```

1 import "openzeppelin-contracts/contracts/access/Ownable.sol";
2 import "@pwn/PWNErrors.sol";

```

The contract has only one storage variable that maps the contract address and a tag to a boolean value indicating if the tag is true or false.

```
1 mapping (address => mapping (bytes32 => bool)) private tags;
```

The contract has only three functions: a) setTag(...); b) setTags(...); and, c) hasTag(...). The function setTag(address _address, bytes32 tag, bool _hasTag) receives as arguments the _address, the tag, and the Boolean _hasTag value indicating if that given address has the tag or not. The function can only be called by the contract owner.

```
1 function setTag(address _address, bytes32 tag, bool _hasTag) public onlyOwner {}
```

The function setTags(...) iterates over an array of addresses and tags, calling the function setTag(...).

```
1 function setTags(address[] memory _addresses, bytes32[] memory _tags, bool _hasTag) external onlyOwner {}
```

Finally, the contract also has the function hasTag(...), which is a standard getter function.

```
1 function hasTag(address _address, bytes32 tag) external view returns (bool) {}
```

5.5 Folder src/loan

5.5.1 PWNVault.sol

This is implemented as an abstract contract that is the basis for transferring and managing collateral and loan assets in the PWN protocol. It contains transfer functions, hooks, and interface checks. The transfer functions rely on the MultiToken.sol implementation.

```
1 using MultiToken for MultiToken.Asset;
```

The internal function _pull(...) transfers the assets into the vault.

```
1 function _pull(MultiToken.Asset memory asset, address origin) internal {}
```

The function _push(...) pushes the asset from the vault to a given recipient.

```
1 function _push(MultiToken.Asset memory asset, address beneficiary) internal {}
```

The function _pushFrom(...) transfers an asset from a lender to a borrower.

```
1 function _pushFrom(MultiToken.Asset memory asset, address origin, address beneficiary) internal {}
```

The function _permit(...) sets the vault allowance for an asset using a signed permit.

```
1 function _permit(MultiToken.Asset memory asset, address origin, bytes memory permit) internal {}
```

The contract also implements receive hooks for the ERC721 and ERC1155.

```
1 function onERC721Received(
2     address operator,
3     address /*from*/,
4     uint256 /*tokenId*/,
5     bytes calldata /*data*/
6 ) override external view returns (bytes4) {}
```

```
1 function onERC1155Received(
2     address operator,
3     address /*from*/,
4     uint256 /*id*/,
5     uint256 /*value*/,
6     bytes calldata /*data*/
7 ) override external view returns (bytes4) {}
```

The function onERC1155BatchReceived(...) is not supported in this version of Vault and is reverted, as shown below.


```

1 function onERC1155BatchReceived(
2     address /*operator*/,
3     address /*from*/,
4     uint256[] calldata /*ids*/,
5     uint256[] calldata /*values*/,
6     bytes calldata /*data*/
7 ) override external pure returns (bytes4) {
8     revert UnsupportedTransferFunction();
9 }

```

Finally, the contract implements the function `supportsInterface(...)` to return the supported interfaces

```

1 function supportsInterface(bytes4 interfaceId) external pure virtual override returns (bool) {}

```

5.5.2 lib/PWNFeeCalculator.sol

The library has only one function responsible for computing the fee amount. The function is shown below.

```

1 function calculateFeeAmount(uint16 fee, uint256 loanAmount) internal pure returns (uint256 feeAmount, uint256
   ↳ newLoanAmount) {}

```

5.5.3 lib/PWNSignatureChecker.sol

The library has only one function responsible for checking the signature using the standard (65 bytes) or compact (64 bytes) defined by EIP-2098.

```

1 function isValidSignatureNow( address signer, bytes32 hash, bytes memory signature ) internal view returns (bool) {}

```

5.5.4 token/PWNLOAN.sol

The contract `PWNLOAN.sol` inherits from `PWNHubAccessControl`, `IERC5646` and `ERC721`, and has the following imports:

```

1 import "openzeppelin-contracts/contracts/token/ERC721/ERC721.sol";
2 import "@pwn/hub/PWNHubAccessControl.sol";
3 import "@pwn/loan/token/IERC5646.sol";
4 import "@pwn/loan/token/IPWNLoanMetadataProvider.sol";
5 import "@pwn/PWNErrors.sol";

```

The contract has two storage variables that track the last loan id, and the mapping from the loan id to the loan contract.

```

1 uint256 public lastLoanId;
2 mapping (uint256 => address) public loanContract;

```

The contract has five functions: a) `mint(...)`; b) `burn(...)`; c) `tokenURI(...)`; d) `getStateFingerprint(...)`; and e) `supportsInterface(...)`. The function `mint(...)` and `burn(...)` are used to create and delete LOAN Tokens for a particular owner based on a certain loanId.

```

1 function mint(address owner) external onlyActiveLoan returns (uint256 loanId) {}
2
3 function burn(uint256 loanId) external {}

```

The function `tokenURI(...)` returns the LOAN Token Metadata URI.

```

1 function tokenURI(uint256 tokenId) public view virtual override returns (string memory) {}

```

The function `getStateFingerprint(...)` is used to return the current token state fingerprint defined by EIP-5646.

```

1 function getStateFingerprint(uint256 tokenId) external view virtual override returns (bytes32) {}

```

And the final function `supportsInterface(...)` which checks if the provided `interfaceId` is valid, along with if the `interfaceId` is the same interface id of `ERC5646`.

```

1 function supportsInterface(bytes4 interfaceId) public view virtual override returns (bool) {}

```

5.5.5 terms/PWNLOANTerms.sol

The library has only one structure Simple which defines the terms of a simple loan.

```

1 struct Simple {
2     address lender;
3     address borrower;
4     uint40 expiration;
5     bool lateRepaymentEnabled;
6     MultiToken.Asset collateral;
7     MultiToken.Asset asset;
8     uint256 loanRepayAmount;
9 }

```

5.5.6 terms/simple/loan/PWNSimpleLoan.sol

The contract PWNSimpleLoan.sol inherits from PWNVault, IERC5646 IPWNLoanMetadataProvider, and has the following imports:

```

1 import "MultiToken/MultiToken.sol";
2 import "@pwn/config/PWNConfig.sol";
3 import "@pwn/hub/PWNHub.sol";
4 import "@pwn/hub/PWNHubTags.sol";
5 import "@pwn/loan/lib/PWNFeeCalculator.sol";
6 import "@pwn/loan/terms/PWNLOANTerms.sol";
7 import "@pwn/loan/terms/simple/factory/IPWNSimpleLoanTermsFactory.sol";
8 import "@pwn/loan/token/IERC5646.sol";
9 import "@pwn/loan/token/PWNLOAN.sol";
10 import "@pwn/loan/PWNVault.sol";
11 import "@pwn/PWNErrors.sol";

```

The contract has three immutable storage and a mapping. The mapping stores all the data of a Loan based on a loan id.

```

1 PWNHub immutable internal hub;
2 PWNLOAN immutable internal loanToken;
3 PWNConfig immutable internal config;
4 mapping (uint256 => LOAN) private LOANs;

```

It also contains a structure LOAN, which defines a simple loan.

```

1 struct LOAN {
2     uint8 status;
3     address borrower;
4     uint40 expiration;
5     bool lateRepaymentEnabled;
6     address loanAssetAddress;
7     uint256 loanRepayAmount;
8     MultiToken.Asset collateral;
9 }

```

The contract has mainly four state-changing external functions: a) createLOAN(...); b) repayLOAN(...); c) claimLOAN(...); and d) enableLOANLateRepayment(...).

The function createLOAN(...) is used to create a Loan with particular collateral and asset. loanTerms are taken from a loanTermsFactoryContract specified in the calldata. Collateral is checked for validity, and then LOAN Token is minted to the lender. Collateral is transferred to the Loan Contract, the fee is read from config, and the corresponding fee is transferred to the feeCollector, and the rest of the amount is transferred to the borrower.

```

1 function createLOAN(
2     address loanTermsFactoryContract,
3     bytes calldata loanTermsFactoryData,
4     bytes calldata signature,
5     bytes calldata loanAssetPermit,
6     bytes calldata collateralPermit
7 ) external returns (uint256 loanId) {}

```

The function repayLOAN is used by the borrower to repay the LOAN with a particular loanId. loanId is checked for existence and invalid loan status. It is ensured that the loan has not expired, and if it has expired, it is checked if later repayment is enabled. If all conditions are satisfied, the loan is repaid to the Loan Contract, and the collateral from Loan Contract is returned to the borrower.

```

1 function repayLOAN(
2     uint256 loanId,
3     bytes calldata loanAssetPermit
4 ) external {}

```

The function `claimLOAN` is used by the lender to claim the loan proceeds with a particular `loanId`. Only the owner of the `LOAN Token` with the same `loanId` passed as `calldata` can call this function. The loan status is checked for mainly three cases: (a) Existence; (b) Repaid Loan; and (c) Expired Loan. In case (a), the call reverts. In case (b), the `LOAN Token` is burned from the caller, and the asset, along with interest, is paid back to the caller. In case (c), the `LOAN Token` is burned from the caller, and the collateral is sent to the caller.

```

1 function claimLOAN(uint256 loanId) external {}

```

The function `enableLOANLateRepayment` is used by the lender to allow late payment for a particular `loanId`. Only the owner of the `LOAN Token` with the same `loanId` passed as `calldata` can call this function. The call will revert if late repayment is already enabled. The loan has to be ongoing for this call to succeed.

```

1 function enableLOANLateRepayment(uint256 loanId) external {}

```

5.5.7 terms/simple/factory/request/PWNSimpleLoanSimpleRequest.sol

The contract `PWNSimpleLoanSimpleRequest.sol` inherits from `PWNSimpleLoanRequest`, and has the following imports:

```

1 import "MultiToken/MultiToken.sol";
2 import "@pwn/loan/lib/PWNSignatureChecker.sol";
3 import "@pwn/loan/terms/simple/factory/request/base/PWNSimpleLoanRequest.sol";
4 import "@pwn/loan/terms/PWNLOANTerms.sol";
5 import "@pwn/PWNErrors.sol";

```

The contract has two constant and one immutable storage, which stores the version, request typehash, and domain separator.

```

1 string internal constant VERSION = "1.0";
2 bytes32 constant internal REQUEST_TYPEHASH = keccak256(
3     "Request(uint8 collateralCategory,address collateralAddress,uint256 collateralId,uint256 collateralAmount,address
4     ↳ loanAssetAddress,uint256 loanAmount,uint256 loanYield,uint32 duration,uint40 expiration,address borrower,address
5     ↳ lender,bool lateRepaymentEnabled,uint256 nonce)"
6 );
7 bytes32 immutable internal DOMAIN_SEPARATOR;

```

It also contains a structure `Request`, which defines a simple request.

```

1 struct Request {
2     MultiToken.Category collateralCategory;
3     address collateralAddress;
4     uint256 collateralId;
5     uint256 collateralAmount;
6     address loanAssetAddress;
7     uint256 loanAmount;
8     uint256 loanYield;
9     uint32 duration;
10    uint40 expiration;
11    address borrower;
12    address lender;
13    bool lateRepaymentEnabled;
14    uint256 nonce;
15 }

```

The contract has mainly two state-changing external functions: a) `makeRequest(...)`; b) `createLOANTerms(...)`. The function `makeRequest()` creates an on-chain request. It has a `calldata` request which contains all the details of the loan request as a structure parameter. The function `createLOANTerms(...)` creates a simple loan term from the provided info as `calldata`. The request data is decoded from `factoryData`, and the signature is checked for validity, ensuring that the request is not expired already and no revoke is initiated for the particular request. As this is a request, the lender cannot be a zero address. If every check is passed, a `loanTerms` is created and returned with this function call.

```

1 function makeRequest(Request calldata request) external {}
2
3 function createLOANTerms(
4     address caller,
5     bytes calldata factoryData,
6     bytes calldata signature
7 ) external override onlyActiveLoan returns (PWNLOANTerms.Simple memory loanTerms) {}

```

5.5.8 terms/simple/factory/request/base/PWNSimpleLoanRequest.sol

The abstract contract PWNSimpleLoanRequest.sol inherits from IPWNSimpleLoanTermsFactory and PWNHubAccessControl, and has the following imports:

```

1 import "@pwn/hub/PWNHubAccessControl.sol";
2 import "@pwn/loan/terms/simple/factory/IPWNSimpleLoanTermsFactory.sol";
3 import "@pwn/nonce/PWNRevokedNonce.sol";
4 import "@pwn/PWNErrors.sol";

```

The contract has one immutable storage and mapping, which stores the revoked request nonces and bool status of requests made.

```

1 PWNRevokedNonce immutable internal revokedRequestNonce;
2 mapping (bytes32 => bool) public requestsMade;

```

The contract has mainly one state-changing external function revokeRequestNonce(...). The function revokeRequestNonce(...) revokes the nonce of the caller.

```

1 function revokeRequestNonce(uint256 requestNonce) external {}

```

5.5.9 terms/simple/factory/offer/PWNSimpleLoanListOffer.sol

The contract PWNSimpleLoanListOffer.sol inherits from PWNSimpleLoanOffer, and has the following imports:

```

1 import "MultiToken/MultiToken.sol";
2 import "openzeppelin-contracts/contracts/utils/cryptography/MerkleProof.sol";
3 import "@pwn/loan/lib/PWNSignatureChecker.sol";
4 import "@pwn/loan/terms/simple/factory/offer/base/PWNSimpleLoanOffer.sol";
5 import "@pwn/loan/terms/PWNLOANTerms.sol";
6 import "@pwn/PWNErrors.sol";

```

The contract has two constant and one immutable storage, which stores the version, and offers typehash and domain separator.

```

1 string internal constant VERSION = "1.0";
2 bytes32 constant internal OFFER_TYPEHASH = keccak256(
3     "Offer(uint8 collateralCategory,address collateralAddress,bytes32 collateralIdsWhitelistMerkleRoot,uint256
4     → collateralAmount,address loanAssetAddress,uint256 loanAmount,uint256 loanYield,uint32 duration,uint40
5     → expiration,address borrower,address lender,bool isPersistent,bool lateRepaymentEnabled,uint256 nonce)"
6 );
7 bytes32 immutable internal DOMAIN_SEPARATOR;

```

It also contains the structure Request, which defines a simple list offer.

```

1 struct Offer {
2     MultiToken.Category collateralCategory;
3     address collateralAddress;
4     bytes32 collateralIdsWhitelistMerkleRoot;
5     uint256 collateralAmount;
6     address loanAssetAddress;
7     uint256 loanAmount;
8     uint256 loanYield;
9     uint32 duration;
10    uint40 expiration;
11    address borrower;
12    address lender;
13    bool isPersistent;
14    bool lateRepaymentEnabled;
15    uint256 nonce;
16 }

```

And another structure OfferValues, which defines Offer values.

```

1 struct OfferValues {
2     uint256 collateralId;
3     bytes32[] merkleInclusionProof;
4 }

```

The contract has mainly two state-changing external functions: a) `makeOffer(...)`; b) `createLOANTerms(...)`. The function `makeOffer(...)` creates an on-chain offer. It has a calldata offer, which contains all the details of the loan offer as a struct parameter. The function `createLOANTerms(...)` creates a simple loan term from the provided info as calldata. The offer and offerValues data is decoded from factoryData, and the signature is checked for validity, ensuring that the offer is not expired already. No revoke is initiated for the particular offer, and in this case, it is also checked that the collateral is verified. As this is a offer, the borrower cannot be a zero address. If every check is passed, a loanTerms is created and returned with this function call.

```

1 function makeOffer(Offer calldata offer) external {}
2
3 function createLOANTerms(
4     address caller,
5     bytes calldata factoryData,
6     bytes calldata signature
7 ) external override onlyActiveLoan returns (PWNLOANTerms.Simple memory loanTerms) {}

```

5.5.10 terms/simple/factory/offer/PWNSimpleLoanSimpleOffer.sol

The contract `PWNSimpleLoanSimpleOffer.sol` inherits from `PWNSimpleLoanOffer`, and has the following imports:

```

1 import "MultiToken/MultiToken.sol";
2 import "@pwn/loan/lib/PWNSignatureChecker.sol";
3 import "@pwn/loan/terms/simple/factory/offer/base/PWNSimpleLoanOffer.sol";
4 import "@pwn/loan/terms/PWNLOANTerms.sol";
5 import "@pwn/PWNErrors.sol";

```

The contract has two constant and one immutable storage, which stores the version, offer typehash and domain separator.

```

1 string internal constant VERSION = "1.0";
2 bytes32 constant internal OFFER_TYPEHASH = keccak256(
3     "Offer(uint8 collateralCategory,address collateralAddress,uint256 collateralId,uint256 collateralAmount,address
4     → loanAssetAddress,uint256 loanAmount,uint256 loanYield,uint32 duration,uint40 expiration,address borrower,address
5     → lender,bool isPersistent,bool lateRepaymentEnabled,uint256 nonce)"
6 );
7 bytes32 immutable internal DOMAIN_SEPARATOR;

```

It also contains a structure Request, which defines a simple list offer.

```

1 struct Offer {
2     MultiToken.Category collateralCategory;
3     address collateralAddress;
4     uint256 collateralId;
5     uint256 collateralAmount;
6     address loanAssetAddress;
7     uint256 loanAmount;
8     uint256 loanYield;
9     uint32 duration;
10    uint40 expiration;
11    address borrower;
12    address lender;
13    bool isPersistent;
14    bool lateRepaymentEnabled;
15    uint256 nonce;
16 }

```

The contract has mainly two state-changing external functions: a) `makeOffer(...)`; b) `createLOANTerms(...)`. The function `makeOffer(...)` creates an on-chain offer. It has a calldata offer, which contains all the details of the loan offer as a struct parameter. The function `createLOANTerms(...)` creates a simple loan term from the provided info as calldata. The offer and offerValues data is decoded from factoryData, the signature is checked for validity, ensured that the offer is not expired already and no revoke is initiated for the particular offer. As this is a offer, the borrower cannot be a zero address. If every check is passed, a loanTerms is created and returned with this function call.

```

1 function makeOffer(Offer calldata offer) external {}
2
3 function createLOANterms(
4     address caller,
5     bytes calldata factoryData,
6     bytes calldata signature
7 ) external override onlyActiveLoan returns (PWNLOANterms.Simple memory loanTerms) {}

```

5.5.11 terms/simple/factory/offer/base/PWNSimpleLoanOffer.sol

The abstract contract `PWNSimpleLoanOffer.sol` inherits from `IPWNSimpleLoanTermsFactory` and `PWNHubAccessControl`, and has the following imports:

```

1 import "@pwn/hub/PWNHubAccessControl.sol";
2 import "@pwn/loan/terms/simple/factory/IPWNSimpleLoanTermsFactory.sol";
3 import "@pwn/nonce/PWNRevokedNonce.sol";
4 import "@pwn/PWNErrors.sol";

```

The contract has one immutable storage and mapping, which stores the revoked offer nonces and bool status of offers made.

```

1 PWNRevokedNonce immutable internal revokedOfferNonce;
2 mapping (bytes32 => bool) public offersMade;

```

The contract has mainly one state-changing external function `revokeOfferNonce(...)`. The function `revokeOfferNonce(...)` revokes the nonce of the caller.

```

1 function revokeOfferNonce(uint256 offerNonce) external {}

```

5.6 Folder src/nonce

This folder has only the contract `PWNRevokedNonce.sol` which inherits `PWNHubAccessControl`. This contract has the following imports.

```

1 import "@pwn/hub/PWNHubAccessControl.sol";
2 import "@pwn/PWNErrors.sol";

```

The contract has three storage variables: a) `accessTag`; b) `revokedNonces`; c) `minNonces`, which stores the tag, revoked nonces and minimum nonce value for a particular address.

```

1 bytes32 immutable internal accessTag;
2 mapping (address => mapping (uint256 => bool)) private revokedNonces;
3 mapping (address => uint256) private minNonces;

```

The contract has mainly three state-changing external functions: a) `revokeNonce(...)`; b) `revokeNonce(...)` (Different parameters); c) `setMinNonce(...)`. Both the function `revokeNonce(...)` is used to revoke a particular nonce of a certain owner. In the first function, it can be used by any caller to revoke the nonce of itself, and in the second function, it can be used by any caller, which has the required `accessTag`. The function `setMinNonce(...)` is used to set a minimum nonce value for that particular caller. Any nonce smaller than the `minNonce` is considered revoked.

```

1 function revokeNonce(uint256 nonce) external {}
2 function revokeNonce(address owner, uint256 nonce) external onlyWithTag(accessTag) {}
3 function setMinNonce(uint256 minNonce) external {}

```

Fig. 3 shows the structural diagram of the protocol, highlighting the borrow and lend use cases.

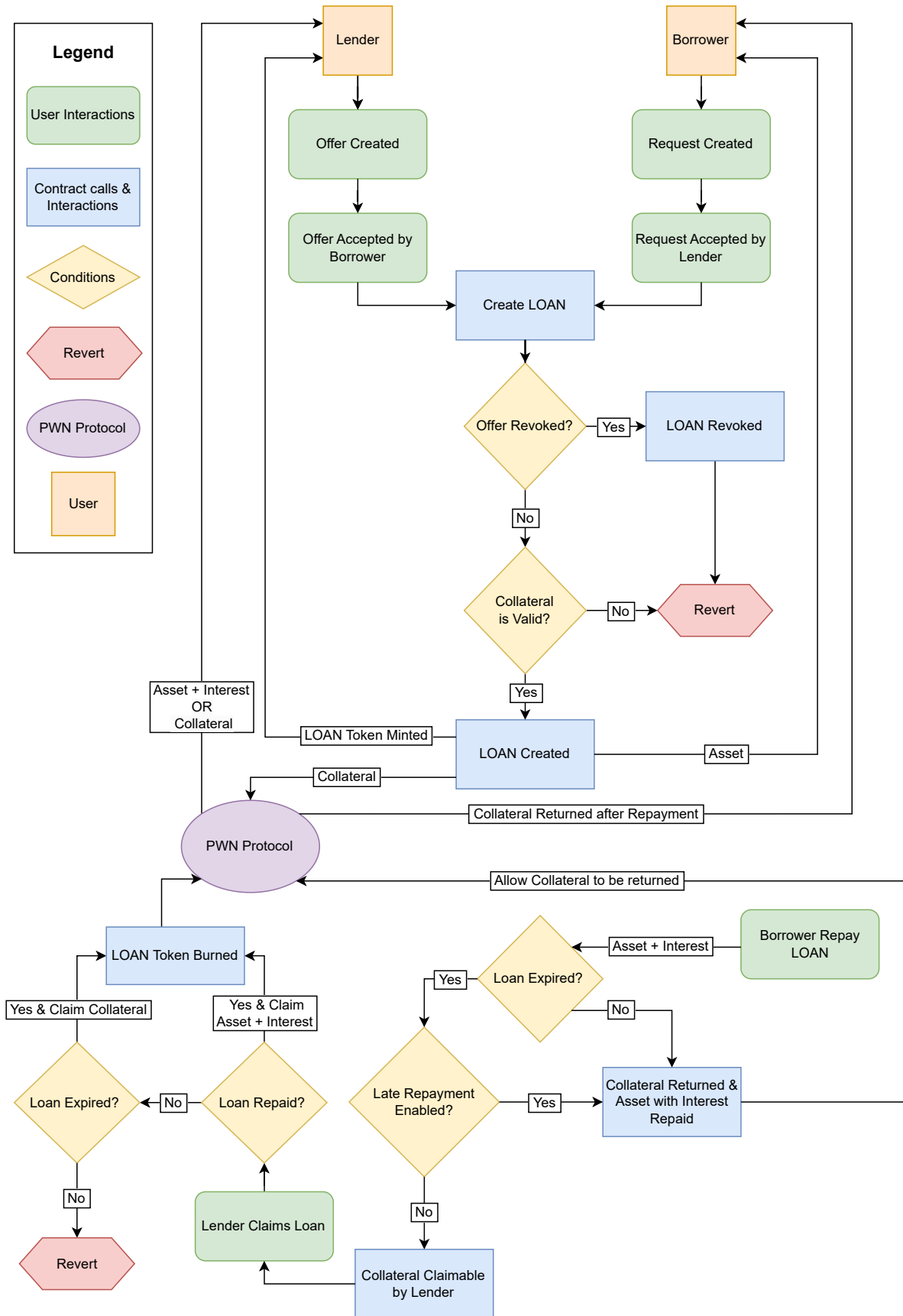


Fig. 3: Borrow and Lend protocol flow

6 Risk Rating Methodology

The risk rating methodology used by [Nethermind](#) follows the principles established by the [OWASP Foundation](#). The severity of each finding is determined by two factors: **Likelihood** and **Impact**.

Likelihood measures how likely the finding is to be uncovered and exploited by an attacker. This factor will be one of the following values:

- a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;
- b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;
- c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding other factors are also considered. These can include but are not limited to: Motive, opportunity, exploit accessibility, ease of discovery, and ease of exploit.

Impact is a measure of the damage that may be caused if the finding were to be exploited by an attacker. This factor will be one of the following values:

- a) **High**: The issue can cause significant damage such as loss of funds or the protocol entering an unrecoverable state;
- b) **Medium**: The issue can cause moderate damage such as impacts that only affect a small group of users or only a particular part of the protocol;
- c) **Low**: The issue can cause little to no damage such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding other factors are also considered. These can include but are not limited to Data/state integrity, loss of availability, financial loss, and reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

		Severity Risk		
Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Info/Best Practices	Low	Medium
	Undetermined	Undetermined	Undetermined	Undetermined
		Low	Medium	High
		Likelihood		

To address issues that do not fit a High/Medium/Low severity, [Nethermind](#) also uses three more finding severities: **Informational**, **Best Practices**, and **Undetermined**.

- a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to pass to the client formally;
- b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;
- c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

7 Issues

7.1 [High] "No Revert on Failure" tokens can be used to steal from lender

File(s): [pwn_contracts/src/loan/terms/simple/loan/PWNSimpleLoan.sol](#)

Description: Some ERC20 tokens don't revert on failure but return false instead. Posting a request with such a token as collateral but with an incorrect asset type (ERC721 or CryptoKitties) may be used to create a loan for which no collateral is transferred. A good example of such a token is ZRX: [Etherscan code](#).

Exploit Scenario:

- Alice posts a loan request with ZRX as collateral, but with ERC721 type. Alice does not have any ZRX tokens;
- Bob creates a loan from Alice's request;
- On loan creation, Bob's loan assets are transferred to Alice. However, Alice's collateral is not transferred since Alice has no ZRX token. The `ZRX.transferFrom(...)` does not revert but returns false;
- The function `_transferAssetFrom(...)` does not check the return value for the CryptoKitties type, so the failed transfer `ICryptoKitties(ZRX).transferFrom(...)` will silently return;
- Bob lent assets to Alice, but Alice did not provide any collateral.

The function is shown below.

```

1 function transferFrom(address _from, address _to, uint _value) returns (bool) {
2     if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
3         balances[_to] += _value;
4         balances[_from] -= _value;
5         allowed[_from][msg.sender] -= _value;
6         Transfer(_from, _to, _value);
7         return true;
8     } else { return false; }
9 }

```

Recommendation(s): Check asset balance differences before and after each transfer.

Status: Fixed

Update from the client: Fixed by checking asset category in the `isValid` function of the `MultiToken` library.

Relevant commits:

- [3139c5745edd73313e7b7381378209f6f134b21d](#) (category check);
- [e5078eff465c2b53d0da0428ed46ebf1d102ff52](#) (gas optimisation).

Update from Nethermind: We agree with the checks added to the function `isValid(...)` in commit [e5078ef](#). However, it does not consider tokens that support ERC165 but are not ERC721, ERC1155, or CryptoKitties. In the future, such tokens may become popular and be used with `assert.category = ERC20`. To address this issue, we recommend verifying that the asset interface id is ERC20 if it supports the ERC165 standard. This solution would resolve the aforementioned problem while also allowing potential ERC20 assets that implement ERC165.

Update from the client after Reaudit: Fixed by checking for support of `IERC20` interface id (`0x36372b07`) in case where ERC20 asset implements ERC165.

Relevant commits:

- [a2971be1ee44a4cd49c2d9f5a88830cce9560a6b](#).

7.2 [High] Borrower can steal lender's assets by using WETH as collateral

File(s): [pwn_contracts/src/loan/terms/simple/loan/PWNSimpleLoan.sol](#)

Description: The borrower can create a request with the correct amount of the WETH token as collateral but with `collateralCategory = ERC1155` to steal the lender's funds. The types of collateral are checked with `isValid(...)`, but the ERC1155 is not checked in that function, which effectively allows the creation of an object with an asset of category ERC1155 but with a WETH token. When the collateral is sent from the borrower to the contract `PWNSimpleLoan`, the ERC1155 interface is used. However, the WETH token doesn't contain the function `safeTransferFrom(...)` implemented, and its fallback function will ignore any unknown function. We present the WETH fallback function below.

```

1 function() public payable {
2     deposit();
3 }
4
5 function deposit() public payable {
6     balanceOf[msg.sender] += msg.value;
7     Deposit(msg.sender, msg.value);
8 }

```

Pulling collateral from the borrower will silently return with no WETH tokens transferred. Next, the loan funds from the lender will be sent to the borrower. At this point, the borrower has no incentive to return the loan since no collateral is kept in the contract.

Recommendation(s): Check the balances after transfers. Checking balances for each token type mitigates using incorrect types and ensures the transfer of correct amounts.

Status: Fixed

Update from the client: Fixed by checking asset category in the function `isValid(...)` of the MultiToken library.

Relevant commits:

- [3139c5745edd73313e7b7381378209f6f134b21d](#) (category check);
- [e5078eff465c2b53d0da0428ed46ebf1d102ff52](#) (gas optimisation).

Update from Nethermind: We agree with the checks added to the function `isValid(...)` in commit [e5078ef](#). However, it does not consider tokens that support ERC165 but are not ERC721, ERC1155, or CryptoKitties. In the future, such tokens may become popular and be used with `assert.category = ERC20`. To address this issue, we recommend verifying that the asset interface id is ERC20 if it supports the ERC165 standard. This solution would resolve the problem above while allowing potential ERC20 assets that implement ERC165.

Update from the client after Reaudit: Fixed by checking for support of IERC20 interface id (`0x36372b07`) in case where ERC20 asset implements ERC165.

Relevant commits:

- [a2971be1ee44a4cd49c2d9f5a88830cce9560a6b](#).

7.3 [Medium] Borrower can prevent the lender from receiving collateral during a claim

File(s): `pwn/src/loan/terms/simple/loan/PWNSimpleLoan.sol`

Description: During the creation, repayment, or claiming of a loan, the MultiToken library handles token transfers through the `_pull`, `_push`, and `_pushFrom` functions. For ERC721 token transfers, the MultiToken library will use `transferFrom(...)` during a token "pull" since moving tokens to itself implicitly implies that it has ERC721 support and therefore does not need to check for `onERC721Received`. During a token "push" the `safeTransferFrom` functionality is used since the recipient may not support ERC721 tokens.

```

1 function() public payable {
2     deposit();
3 }
4
5 function deposit() public payable {
6     balanceOf[msg.sender] += msg.value;
7     Deposit(msg.sender, msg.value);
8 }

```

A borrower could create an order with WETH as collateral and specify it as an ERC721 asset type. Upon a lender accepting the loan, the WETH would successfully transfer to the vault, since it uses a "pull" which uses `transferFrom`. However, if the borrower decides not to repay the loan, when the lender attempts to claim the collateral, a `safeTransferFrom` is used, which will not transfer tokens. In this case, both parties will lose out on their collateral. However, the borrower will have kept the collateral, so the borrower's losses are less than the lender's. For example, with collateral of \$1000 and a borrowed amount of \$800, the attack would cost the borrower \$200, i.e., \$1000 - \$800, to inflict an \$800 loss (the borrowed amount) onto the lender.

Recommendation(s): Check balances after transfers. Checking balances for each token type will mitigate using incorrect types and will ensure the transfer of correct amounts.

Status: Fixed

Update from the client: Fixed by checking asset category in the `isValid` function of the MultiToken library.

Relevant commits:

- [3139c5745edd73313e7b7381378209f6f134b21d](#) (category check);
- [e5078eff465c2b53d0da0428ed46ebf1d102ff52](#) (gas optimisation).

Update from Nethermind: We agree with the checks added to the function `isValid(...)` in commit [e5078ef](#). However, it does not consider tokens that support ERC165 but are not ERC721, ERC1155, or CryptoKitties. In the future, such tokens may become popular and be used with `assert.category = ERC20`. To address this issue, we recommend verifying that the asset interface id is ERC20 if it supports the ERC165 standard. This solution would resolve the aforementioned problem while also allowing potential ERC20 assets that implement ERC165.

Update from the client after Reaudit: Fixed by checking for support of IERC20 interface id (0x36372b07) in case where ERC20 asset implements ERC165.

Relevant commits:

- [a2971be1ee44a4cd49c2d9f5a88830cce9560a6b](#).

7.4 [Medium] Fee-on-transfer tokens can be locked in the vault

File(s): [pwn_contracts/src/loan/terms/simple/loan/PWNSimpleLoan.sol](#)

Description: The loan can be created for any pair of tokens. Some tokens (e.g., STA, PAX) deduce fees during a transfer. If such tokens are used as collateral, the collateral amount registered in the object `loan` would be greater than the actual amount of tokens in the vault, which would fail while trying to withdraw the collateral. There is a second scenario where two users use the same fee-on-transfer Token as collateral, and the first one who repays the loan has a better chance of getting the entire collateral back. In contrast, the second user will not be able to recover the collateral.

Recommendation(s): Consider checking balances before and after transfers to ensure the amount of the transferred tokens is correct. This solution would, however, make fee-on-transfer tokens unusable in this protocol.

Status: Fixed

Update from the client: Fixed by checking balances before and after a transfer.

Relevant commits:

- [5d601c37b77b3eae57903340c5e20f1d29e1b8f](#) (new auxiliary transfer amount function);
- [0f0b2968dd5e581cdd0d18296bc01b170422767d](#) (check invalid transfer).

7.5 [Medium] Late repayment feature could be abused by lenders to trick borrowers into losing their collateral

File(s): [pwn_contracts/src/loan/terms/simple/loan/PWNSimpleLoan.sol](#)

Description: In `PWNSimpleLoan`, lenders can enable late repayment to allow borrowers to repay after the loan has expired. However, this feature can also be used by lenders to trick borrowers into giving up their collateral.

Exploit Scenario:

- Alice (the lender) and Bob (the borrower) agreed on a loan Alice offered. Usually, the value of the collateral Bob deposited must be higher than the loan amount;
- Alice told Bob that she would allow him to repay the loan late by calling the function `enableLOANLateRepayment(...)`. Bob saw this and thought that Alice was being friendly and that he could repay the loan a little bit late;
- However, as soon as the loan expired, Alice called the function `claimLOAN(...)` to get all the collateral. Since the collateral value was higher than the loan, Alice made a profit;

Recommendation(s): Extend the duration of the loan to a fixed value in case the lender enables late repayment and does not allow the lender to claim the collateral before the new expiration.

Status: Fixed

Update from the client: Fixed by removing `enableLOANLateRepayment` functionality and implementing `extendLOANExpirationDate` conditions:

- extended date cannot be before the current expiration date;
- extended date cannot be before `current block.timestamp`;
- extended date cannot be further than `block.timestamp + 30 days` to protect the lender;

Relevant commits:

- [598f84ad5abfa56377bfc0a3b6d2791b16f3031b](#).

Update from Nethermind: In commit [598f84a](#), the comment regarding `loanRepayAmount` was mistakenly deleted instead of the comment about `lateRepaymentEnabled`, in the contract `PWNSimpleLoan`.

7.6 [Medium] Some common ERC20 tokens can't be used in the protocol

File(s): `MultiToken/src/MultiToken.sol`

Description: The function `_transferAssetFrom(...)` checks the return value from the ERC20 token transfer. The code snippet is reproduced below.

```

1  if (asset.category == Category.ERC20) {
2      if (source == address(this))
3          require(IERC20(asset.assetAddress).transfer(dest, asset.amount), "MultiToken: ERC20 transfer failed");
4      else
5          require(IERC20(asset.assetAddress).transferFrom(source, dest, asset.amount), "MultiToken: ERC20 transferFrom
→ failed");
6
7  }
```

That makes a problem for transferring tokens that do not return a success value. Some tokens as [USDT](#) may not be usable in the protocol, since the `require` statement would fail when no `boolean` value is returned, even if the transfer would be successful. Others as [BNB](#), may make lenders lose their funds, below we present an exploit scenario.

Exploit Scenario:

- A loan is created with BNB as collateral;
- The lender sends loan assets to the borrower, and the borrower successfully provides BNB as collateral (`BNB.transferFrom(...)` returns `bool`);
- When the loan is repaid or expired, the collateral can't be withdrawn from the vault since the function `BNB.transfer(...)` does not return a `bool` value, and the `require` statement in `_transferAssetFrom(...)` will always fail, effectively locking BNB in the vault;

Recommendation(s): Consider using the `SafeERC20` library to allow transfers of ERC20 tokens that do not return on a successful transfer.

Status: Fixed

Update from the client: Fixed by using `OZ SafeERC20` library and checking loan asset validity by the `isValid` function.

Relevant commits:

- [53205a29c572a66c60b93dc13ea4463d8d5dc52b](#) (`SafeERC20`);
- [bcfa846cad13682f9d4581a1bb0e2defc303c5ec](#) (loan asset check).

7.7 [Medium] CryptoKitties token is locked when using it as ERC721 type collateral

File(s): `pwn_contracts/src/loan/terms/simple/loan/PWNSimpleLoan.sol`

Description: When creating a loan, the collateral token is not checked to match its category. Since `CryptoKitties` and `ERC721` share the same function `transferFrom(...)`, if the collateral token is `CryptoKitties` but the user signs it as `ERC721` category, it will successfully transfer in. However, when transferring out, `ERC721` category uses the function `safeTransferFrom(...)`, which does not exist in `CryptoKitties`. As a result, this `CryptoKitties` collateral is locked in `PWNVault`. Below we present part of the function `_transferAssetFrom(...)`, which is responsible for transferring `ERC721` assets:

```

1  function _transferAssetFrom(Asset memory asset, address source, address dest, bool isSafe) private {
2      ...
3      } else if (asset.category == Category.ERC721) {
4          if (!isSafe)
5              IERC721(asset.assetAddress).transferFrom(source, dest, asset.id);
6          else
7              IERC721(asset.assetAddress).safeTransferFrom(source, dest, asset.id, "");
8      ...
9  }
```

Recommendation(s): We recommend to use the `ERC165` interface to verify the type of asset, as both `CryptoKitties` and `ERC721` support different interfaces.

Status: Fixed

Update from the client: Fixed by checking asset category in the `isValid` function of the `MultiToken` library.

Relevant commits:

- [3139c5745edd73313e7b7381378209f6f134b21d](#) (category check);
- [e5078eff465c2b53d0da0428ed46ebf1d102ff52](#) (gas optimisation).

Update from Nethermind: We agree with the checks added to the function `isValid(...)` in commit [e5078ef](#). However, it does not consider tokens that support ERC165 but are not ERC721, ERC1155, or CryptoKitties. In the future, such tokens may become popular and be used with `assert.category = ERC20`. To address this issue, we recommend verifying that the asset interface id is ERC20 if it supports the ERC165 standard. This solution would resolve the problem above, while also allowing potential ERC20 assets that implement ERC165.

Update from the client after Reaudit: Fixed by checking for support of IERC20 interface id (`0x36372b07`) in case where ERC20 asset implements ERC165.

Relevant commits:

- [a2971be1ee44a4cd49c2d9f5a88830cce9560a6b](#).

7.8 [Low] Function `calculateFeeAmount(...)` has unnecessary loss of precision

File(s): `pwn_contracts/src/loan/lib/PWNFeeCalculator.sol`

Description: The function `calculateFeeAmount(...)` is used to compute the fee amount given the loan amount and fees value in basis points. Currently in the codebase, if `loanAmount >= 1e4`, to avoid overflow in an edge-case, it does the division before the multiplication and accepts the precision loss. However, the overflow only happens when `loanAmount*fee` overflows, i.e., `loanAmount >= 1e73` in case `fee = 1e4`, which is much less likely to happen compared to `loanAmount >= 1e4` in the current codebase.

```

1  function calculateFeeAmount(uint16 fee, uint256 loanAmount) internal pure returns (uint256 feeAmount, uint256
   ↳ newLoanAmount) {
2      if (loanAmount < 1e4)
3          feeAmount = loanAmount * uint256(fee) / 1e4;
4      else
5          feeAmount = loanAmount / 1e4 * uint256(fee);
6
7      newLoanAmount = loanAmount - feeAmount;
8  }
```

Recommendation(s): Consider changing the check to reduce loss of precision. For example, `(loanAmount * fee) / fee == loanAmount`. With this check, in case `loanAmount * fee` overflows, it cannot get the same `loanAmount` if divided by `fee`. Or consider only using the first formula since no token is not malicious but has `1e73` tokens in the circulating supply.

```

1  function calculateFeeAmount(uint16 fee, uint256 loanAmount) internal pure returns (uint256 feeAmount, uint256
   ↳ newLoanAmount) {
2      unchecked {
3          if ((loanAmount * fee) / fee == loanAmount)
4              feeAmount = loanAmount * uint256(fee) / 1e4;
5          else
6              feeAmount = loanAmount / 1e4 * uint256(fee);
7      }
8      newLoanAmount = loanAmount - feeAmount;
9  }
```

Status: Fixed

Update from the client: Fixed by changing the check according to recommendations.

Relevant commits:

- [506e5e04076f28502eb7d2a7d9b63d448e9ff44f](#).

7.9 [Low] Malicious owner can arbitrarily change the fee to any value

File(s): `pwn_contracts/src/config/PWNConfig.sol`

Description: PWN protocol charges fees when users create loans on the platform. However, a malicious/lost private key owner can arbitrarily change the fee to any value without a timelock for users to react. Consider the scenario when a user creates a large loan with 0.1% fees. The owner can front-run to change fees to 100% right before the user's transaction is executed and steals all the assets.

```

1 function setFee(uint16 _fee) external onlyOwner {
2     _setFee(_fee);
3 }
4
5 function _setFee(uint16 _fee) private {
6     uint16 oldFee = fee;
7     fee = _fee;
8     emit FeeUpdated(oldFee, _fee);
9 }

```

Recommendation(s): Add an upper threshold for fees that the owner can update. You can also consider a time lock so users can have enough time to react to owner actions.

Status: Mitigated

Update from the client: PWNConfig will be deployed as a proxy, so any hardcoded value can be updated anytime. Nevertheless, we set the max value to 10% in combination with TimelockController set as a PWNConfig owner to make it harder for a malicious owner to execute such an attack.

Relevant commits:

- [fb856111ef7116cfa5834e67839e37a95d050d2](#).

7.10 [Low] Ownable contracts

File(s): [pwn_contracts/src/hub/PWNHub.sol](#), [pwn_contracts/src/config/PWNConfig.sol](#), [pwn_contracts/src/deployer/PWNDeployer.sol](#)

Description: The contracts PWNHub, PWNConfig, and PWNDeployer inherit from OpenZeppelin's Ownable contract, which provides a one-step ownership transfer. Transferring ownership in a single step is error-prone and can severely harm the protocol if a mistake happens. Additionally, those contracts inherit the function `renounceOwnership(...)`. If this function were called, those essential contracts would be unusable.

Recommendation(s): We recommend implementing a two-step process for transferring ownership, such as the propose-accept scheme. Check the [Ownable2Step.sol](#) contract from OpenZeppelin. Consider also disabling the function `renounceOwnership(...)` to ensure it's never called.

Status: Fixed

Update from the client: Fixed by inheriting from `Ownable2Step` in PWNHub and PWNConfig. Decided to keep simple `Ownable` for PWNDeployer and keep the `renounceOwnership` function active.

Relevant commits:

- [f7977c1183731b3995fd1b42c84b00079ff90646](#) (PWNConfig update);
- [4fe0ced7e979ea57649437e16eeb75910d1e555e](#) (PWNHub update).

7.11 [Low] offer.duration can be set to zero

File(s): [pwn_contracts/src/loan/terms/simple/factory/offer/PWNSimpleLoanListOffer.sol](#),
[pwn_contracts/src/loan/terms/simple/factory/offer/PWNSimpleLoanSimpleOffer.sol](#),
[pwn/src/loan/terms/simple/factory/request/PWNSimpleLoanSimpleRequest.sol](#)

Description: The `loanTerms.expiration` can be set to the current timestamp if the `offer.duration` is zero. This will allow the lender to force claim the collateral after fulfilling the request. The code snippet is reproduced below.

```

1 ...
2 expiration: uint40(block.timestamp) + offer.duration,
3 ...

```

Recommendation(s): Ensure that `offer.duration` exceeds zero. The lowest value could be around one block time.

Status: Fixed

Update from the client: Set min loan duration to 10 min.

Relevant commits:

- [5c712e486411615ec3ac133a8a5591cd302beaad](#) (revert on 0);
- [53786f804ed4641eae96dc5ef1e376491a9a4f16](#) (set min 10 min duration).

7.12 [Info] Fee Collector can be set to address(0x0) in function setFeeCollector(...)

File(s): [pwn_contracts/src/config/PWNConfig.sol](#)

Description: The function `setFeeCollector(...)` is designed to set a new address for the fee collector. However, the function does not check the new address for `address(0x0)`, which can lead to asset loss. The function is reproduced below.

```

1  function setFeeCollector(address _feeCollector) external onlyOwner {
2      _setFeeCollector(_feeCollector);
3  }
4
5  function _setFeeCollector(address _feeCollector) private {
6      //////////////////////////////////////
7      // @audit Not checking "_feeCollector" for "address(0x0)".
8      //////////////////////////////////////
9      address oldFeeCollector = feeCollector;
10     feeCollector = _feeCollector;
11     emit FeeCollectorUpdated(oldFeeCollector, _feeCollector);
12 }

```

Recommendation(s): Although the function is `onlyOwner`, checking the `_feeCollector` for `address(0x0)` can prevent typos and fund losses.

Status: Fixed

Update from the client: Check for the zero address before setting the new fee collector address.

Relevant commits:

- [1a55bd8e36ed9e2b9b6ceb81143d4eaa3bf29c99](#).

7.13 [Info] Token that reverts on zero-value transfers can cause createLoan(...) to fail when transferring fees

File(s): [pwn_contracts/src/loan/terms/simple/loan/PWNSimpleLoan.sol](#)

Description: Some tokens (e.g., LEND) revert when transferring a zero-value amount. If `feeAmount` is rounded to zero for a small loan, it can cause `createLoan(...)` to fail, meaning users cannot create loans with some tokens.

```

1  // Transfer fee amount to fee collector
2  //////////////////////////////////////
3  // @audit Transferring a zero amount can fail for some tokens
4  //////////////////////////////////////
5  loanTerms.asset.amount = feeAmount;
6  _pushFrom(loanTerms.asset, loanTerms.lender, config.feeCollector());

```

Recommendation(s): Only perform transfers when the amount is larger than zero.

Status: Fixed

Update from the client: Fixed by only transferring the fee when the amount is `>0`.

Relevant commits:

- [adcdd70167db89a95c9f73655bec8b1fc36a17fb](#);

7.14 [Best Practices] Unlocked Pragma

File(s): [MultiToken/src/MultiToken.sol](#)

Description: The contract uses unlocked pragma `^0.8.0`. A better approach is to stick to a single Solidity version, ensuring the protocol works in that context. The solidity compiler frequently has new versions, and using an old version prevents access to new Solidity security checks. Also, notice that some versions are more stable than others.

Recommendation(s): Lock the pragma to version `0.8.16` and use this version for deploying the contract. For more information, please check this [reference](#).

Status: Acknowledged

Update from the client: Acknowledged. Nevertheless, the MultiToken lib is used in more projects which do not share the same pragma version. We've decided to keep the unlocked pragma.

8 Documentation Evaluation

Technical documentation is created to explain what the software product does. This way, developers and stakeholders can easily follow the purpose and the underlying functionality of each file/function/line. Documentation can come not only in the form of a README.md, but also using code as documentation (to write clear code), diagrams, websites, research papers, videos, and external documentation. Besides being a good programming practice, proper technical documentation improves the efficiency of audits. Less time can be spent understanding the protocol and more time can be put towards auditing which improves the efficiency and overall output of the audit.

The PWN team provided developer documents to assist the audit process, describing MultiToken library, the PWN Vault and PWN Loan contracts and a diagram describing the protocol flow. These documents covered the most common terms used in the source code, explanations for the core business logic, and functions flow. By reading the whole developer documentation, we could get a proper understanding of how *PWN Vault* and *PWN Loan* contracts should operate. **The provided documentation is a complete source of resources for developers and auditors, and we are satisfied with it. The codebase has sufficient inline comments, helping the audit team to understand the function flow and to detect some issues.**

8.1 Documentation inconsistencies

Along this investigation, we noticed some areas in the code where the implementation differs from the specification and/or inline comments. Consider updating either the implementation or the documentation to match each other based on the correct business logic.

8.1.1 Documentation about ERC721 does not match implementation

The function `isValid(...)` in the contract library MultiToken assures that the amount of the ERC721 token must be 0. However, in the [developer documentation](#) it is stated that the `collateralAmount` for ERC721 asset must be 1.

8.1.2 Documentation about `onERC1155BatchReceived(...)` does not match implementation

The documentation in contract PWNVault mentions that in order for the batch transfer to be accepted, the `onERC1155BatchReceived(...)` should return its own function selector. But it reverts directly with a custom error message.

```
/**
 * @dev Handles the receipt of a multiple ERC1155 token types. This function
 * is called at the end of a `safeBatchTransferFrom` after the balances have
 * been updated. To accept the transfer(s), this must return
 * `bytes4(keccak256("onERC1155BatchReceived(address,address,uint256[],uint256[],bytes)"))`
 * (i.e. 0xbc197c81, or its own function selector).
 * @return `bytes4(keccak256("onERC1155BatchReceived(address,address,uint256[],uint256[],bytes)"))` if transfer is
 * → allowed
 */
function onERC1155BatchReceived(
    address /*operator*/,
    address /*from*/,
    uint256[] calldata /*ids*/,
    uint256[] calldata /*values*/,
    bytes calldata /*data*/
) override external pure returns (bytes4) {
    revert UnsupportedTransferFunction();
}
```


9 Test Suite Evaluation

9.1 Contracts Compilation Output

```
pwn_contracts % forge build
[] Compiling...
[] Compiling 96 files with 0.8.16
[] Solc 0.8.16 finished in 27.70s
Compiler run successful
```

9.2 Tests Output

```
pwn_contracts % forge test
[] Compiling...
No files changed, compilation skipped

Running 1 test for test/unit/PWNSimpleLoanSimpleOffer.t.sol:PWNSimpleLoanSimpleOffer_EncodeLoanTermsFactoryData_Test
[PASS] test_shouldReturnEncodedLoanTermsFactoryData() (gas: 153472)
Test result: ok. 1 passed; 0 failed; finished in 7.50ms

Running 6 tests for test/unit/PWNHub.t.sol:PWNHub_SetTags_Test
[PASS] test_shouldAddTagsToAddress() (gas: 80019)
[PASS] test_shouldEmitEvent_TagSet_forEverySet() (gas: 82528)
[PASS] test_shouldFail_whenCallerIsNotOwner() (gas: 25714)
[PASS] test_shouldFail_whenDiffInputLengths() (gas: 22214)
[PASS] test_shouldNotFail_whenEmptyList() (gas: 13944)
[PASS] test_shouldRemoveTagsFromAddress() (gas: 25684)
Test result: ok. 6 passed; 0 failed; finished in 7.18ms

Running 6 tests for test/unit/PWNSimpleLoan.t.sol:PWNSimpleLoan_EnableLOANLateRepayment_Test
[PASS] test_shouldEmitEvent_LOANLateRepaymentEnabled() (gas: 35224)
[PASS] test_shouldFail_whenCallerIsNotLOANTokenHolder() (gas: 32608)
[PASS] test_shouldFail_whenLateRepaymentIsAlreadyEnabled() (gas: 35808)
[PASS] test_shouldFail_whenLoanIsNotRunning() (gas: 36236)
[PASS] test_shouldPass_whenLoanIsExpired() (gas: 33987)
[PASS] test_shouldStoreThatLateRepaymentIsEnabled() (gas: 34464)
Test result: ok. 6 passed; 0 failed; finished in 9.15ms

Running 3 tests for test/unit/PWNConfig.t.sol:PWNConfig_Initialize_Test
[PASS] test_shouldFail_whenCalledSecondTime() (gas: 20795)
[PASS] test_shouldFail_whenOwnerIsZeroAddress() (gas: 14165)
[PASS] test_shouldSetOwner() (gas: 19573)
Test result: ok. 3 passed; 0 failed; finished in 9.54ms

Running 6 tests for test/unit/PWNConfig.t.sol:PWNConfig_Reinitialize_Test
[PASS] test_shouldFail_whenCalledSecondTime() (gas: 48758)
[PASS] test_shouldFail_whenCallerIsNotOwner() (gas: 20611)
[PASS] test_shouldFail_whenFeeCollectorIsZeroAddress() (gas: 20459)
[PASS] test_shouldFail_whenInitializeCalledAfterReinitialize() (gas: 48517)
[PASS] test_shouldFail_whenOwnerIsZeroAddress() (gas: 16519)
[PASS] test_shouldSetParameters() (gas: 48180)
Test result: ok. 6 passed; 0 failed; finished in 10.04ms

Running 5 tests for test/integration/PWNProtocolIntegrity.t.sol:PWNProtocolIntegrityTest
[PASS] test_shouldClaimRepaidLOANWithNotActiveLoanContract() (gas: 572100)
[PASS] test_shouldFailCreatingLOANOnNotActiveLoanContract() (gas: 210666)
[PASS] test_shouldFail_whenCallerIsNotActiveLoan() (gas: 29414)
[PASS] test_shouldFail_whenPassingInvalidTermsFactoryContract() (gas: 205572)
[PASS] test_shouldRepayLOANWithNotActiveLoanContract() (gas: 556549)
Test result: ok. 5 passed; 0 failed; finished in 22.59ms

Running 3 tests for test/unit/PWNConfig.t.sol:PWNConfig_SetFeeCollector_Test
[PASS] test_shouldEmitEvent_FeeCollectorUpdated() (gas: 25149)
[PASS] test_shouldFail_whenCallerIsNotOwner() (gas: 12686)
[PASS] test_shouldSetFeeCollectorAddress() (gas: 21826)
Test result: ok. 3 passed; 0 failed; finished in 476.42µs
```

Running 1 test for test/unit/PWNSimpleLoanListOffer.t.sol:PWNSimpleLoanListOffer_EncodeLoanTermsFactoryData_Test
 [PASS] test_shouldReturnEncodedLoanTermsFactoryData() (gas: 194733)
 Test result: ok. 1 passed; 0 failed; finished in 1.46ms

Running 1 test for test/unit/PWNSimpleLoanSimpleOffer.t.sol:PWNSimpleLoanSimpleOffer_GetOfferHash_Test
 [PASS] test_shouldReturnOfferHash() (gas: 32572)
 Test result: ok. 1 passed; 0 failed; finished in 611.63µs

Running 4 tests for test/unit/PWNLOAN.t.sol:PWNLOAN_Burn_Test
 [PASS] test_shouldBurnLOANToken() (gas: 24323)
 [PASS] test_shouldDeleteStoredLoanContract() (gas: 23738)
 [PASS] test_shouldEmitEvent_LOANBurned() (gas: 24354)
 [PASS] test_shouldFail_whenCallerIsNotStoredLoanContractForGivenLoanId() (gas: 15595)
 Test result: ok. 4 passed; 0 failed; finished in 3.21ms

Running 1 test for test/unit/PWNDeployer.t.sol:PWNDeployer_ComputeAddress_Test
 [PASS] test_shouldComputeAddress(bytes32,bytes32) (runs: 256, : 6181, ~: 6181)
 Test result: ok. 1 passed; 0 failed; finished in 28.01ms

Running 5 tests for test/unit/PWNFeeCalculator.t.sol:PWNFeeCalculator_CalculateFeeAmount_Test
 [PASS] testFuzz_feeAndNewLoanAmountAreEqToOriginalLoanAmount(uint16,uint256) (runs: 256, : 3662, ~: 3658)
 [PASS] test_shouldHandleSmallAmount() (gas: 632)
 [PASS] test_shouldHandleZeroAmount() (gas: 633)
 [PASS] test_shouldReturnCorrectValue_forNonZeroFee() (gas: 611)
 [PASS] test_shouldReturnCorrectValue_forZeroFee() (gas: 655)
 Test result: ok. 5 passed; 0 failed; finished in 27.27ms

Running 2 tests for test/unit/PWNSimpleLoan.t.sol:PWNSimpleLoan_GetLOAN_Test
 [PASS] test_shouldReturnEmptyLOANDataForNonExistingLoan() (gas: 36715)
 [PASS] test_shouldReturnLOANData() (gas: 31240)
 Test result: ok. 2 passed; 0 failed; finished in 2.76ms

Running 3 tests for test/unit/PWNConfig.t.sol:PWNConfig_SetFee_Test
 [PASS] test_shouldEmitEvent_FeeUpdated() (gas: 18875)
 [PASS] test_shouldFail_whenCallerIsNotOwner() (gas: 10657)
 [PASS] test_shouldSetFeeValue() (gas: 17762)
 Test result: ok. 3 passed; 0 failed; finished in 1.53ms

Running 1 test for test/unit/PWNSimpleLoanSimpleOffer.t.sol:PWNSimpleLoanSimpleOffer_MakeOffer_Test
 [PASS] test_shouldMakeOffer() (gas: 62261)
 Test result: ok. 1 passed; 0 failed; finished in 663.29µs

Running 1 test for test/unit/PWNSimpleLoanListOffer.t.sol:PWNSimpleLoanListOffer_GetOfferHash_Test
 [PASS] test_shouldReturnOfferHash() (gas: 32641)
 Test result: ok. 1 passed; 0 failed; finished in 631.42µs

Running 3 tests for test/unit/PWNRevokedNonce.t.sol:PWNRevokedNonce_IsNonceRevoked_Test
 [PASS] test_shouldReturnFalse_whenNonceIsNotRevoked() (gas: 14378)
 [PASS] test_shouldReturnTrue_whenNonceIsRevoked() (gas: 13524)
 [PASS] test_shouldReturnTrue_whenNonceIsSmallerThanMinNonce() (gas: 11063)
 Test result: ok. 3 passed; 0 failed; finished in 390.13µs

Running 1 test for test/unit/PWNLOAN.t.sol:PWNLOAN_Constructor_Test
 [PASS] test_shouldHaveCorrectNameAndSymbol() (gas: 13878)
 Test result: ok. 1 passed; 0 failed; finished in 1.45ms

Running 2 tests for test/unit/PWNSimpleLoan.t.sol:PWNSimpleLoan_GetStateFingerprint_Test
 [PASS] test_shouldReturnCorrectStateFingerprint() (gas: 50636)
 [PASS] test_shouldReturnZeroIfLoanDoesNotExist() (gas: 21042)
 Test result: ok. 2 passed; 0 failed; finished in 2.89ms

Running 1 test for test/unit/PWNHub.t.sol:PWNHub_Constructor_Test
 [PASS] test_shouldSetHubOwner() (gas: 366545)
 Test result: ok. 1 passed; 0 failed; finished in 300.21µs

Running 1 test for test/integration/PWNSimpleLoanSimpleOfferIntegration.t.sol:PWNSimpleLoanSimpleOfferIntegrationTest
 [PASS] test_shouldRevokeOffersInGroup_whenAcceptingOneFromGroup() (gas: 565110)
 Test result: ok. 1 passed; 0 failed; finished in 4.86ms

```

Running 3 tests for test/unit/PWNConfig.t.sol:PWNConfig_SetLoanMetadataUri_Test
[PASS] test_shouldEmitEvent_LoanMetadataUriUpdated() (gas: 45263)
[PASS] test_shouldFail_whenCallerIsNotOwner() (gas: 15552)
[PASS] test_shouldSetLoanMetadataUriToLoanContract() (gas: 44244)
Test result: ok. 3 passed; 0 failed; finished in 557.25µs

Running 2 tests for test/unit/PWNRevokedNonce.t.sol:PWNRevokedNonce_RevokeNonceByOwner_Test
[PASS] test_shouldEmitEvent_NonceRevoked() (gas: 38416)
[PASS] test_shouldStoreNonceAsRevoked() (gas: 37414)
Test result: ok. 2 passed; 0 failed; finished in 313.75µs

Running 1 test for test/unit/PWNSimpleLoanListOffer.t.sol:PWNSimpleLoanListOffer_MakeOffer_Test
[PASS] test_shouldMakeOffer() (gas: 62396)
Test result: ok. 1 passed; 0 failed; finished in 669.29µs

Running 2 tests for test/unit/PWNSimpleLoan.t.sol:PWNSimpleLoan_LoanMetadataUri_Test
[PASS] test_shouldCallConfig() (gas: 15060)
[PASS] test_shouldReturnCorrectValue() (gas: 12818)
Test result: ok. 2 passed; 0 failed; finished in 814.88µs

Running 3 tests for test/unit/PWNRevokedNonce.t.sol:PWNRevokedNonce_RevokeNonceWithOwner_Test
[PASS] test_shouldEmitEvent_NonceRevoked() (gas: 43586)
[PASS] test_shouldFail_whenCallerIsDoesNotHaveAccessTag() (gas: 18695)
[PASS] test_shouldStoreNonceAsRevoked() (gas: 42571)
Test result: ok. 3 passed; 0 failed; finished in 447.21µs

Running 2 tests for test/unit/PWNLOAN.t.sol:PWNLOAN_GetStateFingerprint_Test
[PASS] test_shouldCallLoanContract() (gas: 13647)
[PASS] test_shouldReturnZeroIfLoanDoesNotExist() (gas: 9919)
Test result: ok. 2 passed; 0 failed; finished in 559.21µs

Running 2 tests for test/unit/PWNHub.t.sol:PWNHub_HasTag_Test
[PASS] test_shouldReturnFalse_whenAddressDoesNotHaveTag() (gas: 11289)
[PASS] test_shouldReturnTrue_whenAddressDoesHaveTag() (gas: 11301)
Test result: ok. 2 passed; 0 failed; finished in 1.56ms

Running 3 tests for test/unit/PWNRevokedNonce.t.sol:PWNRevokedNonce_SetMinNonceByOwner_Test
[PASS] test_shouldEmitEvent_MinNonceSet() (gas: 38432)
[PASS] test_shouldFail_whenNewValueIsSmallerThanCurrent() (gas: 11916)
[PASS] test_shouldStoreNewMinNonce() (gas: 37313)
Test result: ok. 3 passed; 0 failed; finished in 414.54µs

Running 6 tests for test/unit/PWNLOAN.t.sol:PWNLOAN_Mint_Test
[PASS] test_shouldEmitEvent_LOANMinted() (gas: 87671)
[PASS] test_shouldFail_whenCallerIsNotActiveLoanContract() (gas: 14481)
[PASS] test_shouldIncreaseLastLoanId() (gas: 85707)
[PASS] test_shouldMintLOANToken() (gas: 110110)
[PASS] test_shouldReturnLoanId() (gas: 85169)
[PASS] test_shouldStoreLoanContractUnderLoanId() (gas: 109805)
Test result: ok. 6 passed; 0 failed; finished in 907.42µs

Running 3 tests for test/unit/PWNSimpleLoanOffer.t.sol:PWNSimpleLoanOffer_MakeOffer_Test
[PASS] test_shouldEmitEvent_OfferMade() (gas: 38675)
[PASS] test_shouldFail_whenCallerIsNotLender() (gas: 13382)
[PASS] test_shouldMarkOfferAsMade() (gas: 37356)
Test result: ok. 3 passed; 0 failed; finished in 442.75µs

Running 15 tests for test/unit/PWNSimpleLoanSimpleRequest.t.sol:PWNSimpleLoanSimpleRequest_CreateLOANTerms_Test
[PASS] test_shouldFail_whenCallerIsNotActiveLoan() (gas: 39164)
[PASS] test_shouldFail_whenCallerIsNotLender_whenSetLender() (gas: 149699)
[PASS] test_shouldFail_whenInvalidSignature_whenContractAccount() (gas: 51430)
[PASS] test_shouldFail_whenInvalidSignature_whenEOA() (gas: 146164)
[PASS] test_shouldFail_whenPassingInvalidRequestData() (gas: 21481)
[PASS] test_shouldFail_whenRequestIsExpired() (gas: 129540)
[PASS] test_shouldFail_whenRequestIsRevoked() (gas: 131262)
[PASS] test_shouldPass_whenRequestHasBeenMadeOnchain() (gas: 51552)
[PASS] test_shouldPass_whenRequestHasNoExpiration() (gas: 130904)
[PASS] test_shouldPass_whenRequestIsNotExpired() (gas: 133716)
[PASS] test_shouldPass_whenValidSignature_whenContractAccount() (gas: 57085)
[PASS] test_shouldPass_withValidSignature_whenEOA_whenCompactEIP2098Signature() (gas: 130289)

```

```

[PASS] test_shouldPass_withValidSignature_whenEOA_whenStandardSignature() (gas: 152458)
[PASS] test_shouldReturnCorrectValues() (gas: 133213)
[PASS] test_shouldRevokeRequest() (gas: 133654)
Test result: ok. 15 passed; 0 failed; finished in 17.95ms

Running 8 tests for test/unit/PWNSimpleLoan.t.sol:PWNSimpleLoan_RepayLOAN_Test
[PASS] test_shouldEmitEvent_LOANPaidBack() (gas: 42446)
[PASS] test_shouldFail_whenLoanDoesNotExist() (gas: 32912)
[PASS] test_shouldFail_whenLoanIsExpired_whenLateRepaymentDisable() (gas: 31134)
[PASS] test_shouldFail_whenLoanIsNotRunning() (gas: 33111)
[PASS] test_shouldMoveLoanToRepaidState() (gas: 41721)
[PASS] test_shouldPass_whenLoanIsExpired_whenLateRepaymentEnable() (gas: 44334)
[PASS] test_shouldTransferCollateralToBorrower() (gas: 42349)
[PASS] test_shouldTransferRepaidAmountToVault() (gas: 159614)
Test result: ok. 8 passed; 0 failed; finished in 15.26ms

Running 4 tests for test/unit/PWNHub.t.sol:PWNHub_SetTag_Test
[PASS] test_shouldAddTagToAddress() (gas: 42388)
[PASS] test_shouldEmitEvent_TagSet() (gas: 43622)
[PASS] test_shouldFail_whenCallerIsNotOwner() (gas: 15495)
[PASS] test_shouldRemoveTagFromAddress() (gas: 11108)
Test result: ok. 4 passed; 0 failed; finished in 591.88µs

Running 1 test for test/unit/PWNLOAN.t.sol:PWNLOAN_SupportsInterface_Test
[PASS] test_shouldSupportERC5646() (gas: 5838)
Test result: ok. 1 passed; 0 failed; finished in 982.88µs

Running 1 test for test/unit/PWNSimpleLoanOffer.t.sol:PWNSimpleLoanOffer_RevokeOfferNonce_Test
[PASS] test_shouldCallRevokeOfferNonce() (gas: 16469)
Test result: ok. 1 passed; 0 failed; finished in 902.83µs

Running 1 test for
↳ test/unit/PWNSimpleLoanSimpleRequest.t.sol:PWNSimpleLoanSimpleRequest_EncodeLoanTermsFactoryData_Test
[PASS] test_shouldReturnEncodedLoanTermsFactoryData() (gas: 144353)
Test result: ok. 1 passed; 0 failed; finished in 1.17ms

Running 9 tests for test/unit/PWNSignatureChecker.t.sol:PWNSignatureChecker_isValidSignatureNow_Test
[PASS] testFail_shouldFail_whenSignerIsEOA_whenInvalidSignature() (gas: 101168)
[PASS] test_shouldCallEIP1271Function_whenSignerIsContractAccount() (gas: 11704)
[PASS] test_shouldFail_whenSignerIsContractAccount_whenEIP1271FunctionNotReturnsCorrectValue() (gas: 10423)
[PASS] test_shouldFail_whenSignerIsContractAccount_whenEIP1271FunctionReturnsWrongDataLength() (gas: 10258)
[PASS] test_shouldFail_whenSignerIsEOA_whenSignatureHasWrongLength() (gas: 100949)
[PASS] test_shouldReturnFalse_whenSignerIsEOA_whenSignerIsNotRecoveredAddressOfSignature() (gas: 104484)
[PASS] test_shouldReturnTrue_whenSignerIsContractAccount_whenEIP1271FunctionReturnsCorrectValue() (gas: 10339)
[PASS] test_shouldReturnTrue_whenSignerIsEOA_whenSignerIsRecoveredAddressOfSignature() (gas: 106574)
[PASS] test_shouldSupportCompactEIP2098Signatures_whenSignerIsEOA() (gas: 84411)
Test result: ok. 9 passed; 0 failed; finished in 3.19ms

Running 1 test for test/unit/PWNSimpleLoanSimpleRequest.t.sol:PWNSimpleLoanSimpleRequest_MakeRequest_Test
[PASS] test_shouldMakeRequest() (gas: 61870)
Test result: ok. 1 passed; 0 failed; finished in 668.67µs

Running 1 test for test/unit/PWNSimpleLoanSimpleRequest.t.sol:PWNSimpleLoanSimpleRequest_GetRequestHash_Test
[PASS] test_shouldReturnRequestHash() (gas: 32157)
Test result: ok. 1 passed; 0 failed; finished in 616.71µs

Running 2 tests for test/unit/PWNLOAN.t.sol:PWNLOAN_TokenUri_Test
[PASS] test_shouldCallLoanContract() (gas: 19018)
[PASS] test_shouldReturnCorrectValue() (gas: 17015)
Test result: ok. 2 passed; 0 failed; finished in 632.13µs

Running 19 tests for test/unit/PWNSimpleLoanListOffer.t.sol:PWNSimpleLoanListOffer_CreateLOANTerms_Test
[PASS] testFail_shouldNotRevokeOffer_whenIsPersistent() (gas: 142201)
[PASS] test_shouldAcceptAnyCollateralId_whenMerkleRootIsZero() (gas: 139509)
[PASS] test_shouldFail_whenCallerIsNotActiveLoan() (gas: 44035)
[PASS] test_shouldFail_whenCallerIsNotBorrower_whenSetBorrower() (gas: 138552)
[PASS] test_shouldFail_whenGivenCollateralIdIsNotWhitelisted() (gas: 202601)
[PASS] test_shouldFail_whenInvalidSignature_whenContractAccount() (gas: 57197)
[PASS] test_shouldFail_whenInvalidSignature_whenEOA() (gas: 152068)

```

```
[PASS] test_shouldFail_whenOfferIsExpired() (gas: 135554)
[PASS] test_shouldFail_whenOfferIsRevoked() (gas: 137258)
[PASS] test_shouldFail_whenPassingInvalidOfferData() (gas: 21443)
[PASS] test_shouldPass_whenGivenCollateralIdIsWhitelisted() (gas: 203788)
[PASS] test_shouldPass_whenOfferHasBeenMadeOnchain() (gas: 57628)
[PASS] test_shouldPass_whenOfferHasNoExpiration() (gas: 136847)
[PASS] test_shouldPass_whenOfferIsNotExpired() (gas: 139747)
[PASS] test_shouldPass_whenValidSignature_whenContractAccount() (gas: 62945)
[PASS] test_shouldPass_withValidSignature_whenEOA_whenCompactEIP2098Signature() (gas: 136276)
[PASS] test_shouldPass_withValidSignature_whenEOA_whenStandardSignature() (gas: 158467)
[PASS] test_shouldReturnCorrectValues() (gas: 139294)
[PASS] test_shouldRevokeOffer_whenIsNotPersistent() (gas: 139863)
Test result: ok. 19 passed; 0 failed; finished in 16.27ms

Running 3 tests for test/unit/PWNSimpleLoanRequest.t.sol:PWNSimpleLoanRequest_MakeRequest_Test
[PASS] test_shouldEmitEvent_RequestMade() (gas: 38697)
[PASS] test_shouldFail_whenCallerIsNotBorrower() (gas: 13404)
[PASS] test_shouldMarkRequestAsMade() (gas: 37356)
Test result: ok. 3 passed; 0 failed; finished in 497.83µs

Running 1 test for test/unit/PWNDeployer.t.sol:PWNDeployer_Constructor_Test
[PASS] test_shouldSetParameters(address) (runs: 256, : 364397, ~: 364397)
Test result: ok. 1 passed; 0 failed; finished in 71.47ms

Running 3 tests for test/unit/PWNDeployer.t.sol:PWNDeployer_DeployAndTransferOwnership_Test
[PASS] test_shouldDeployContract() (gas: 386056)
[PASS] test_shouldFail_whenCallerIsNotOwner() (gas: 17807)
[PASS] test_shouldTransferOwnership() (gas: 386215)
Test result: ok. 3 passed; 0 failed; finished in 899.17µs

Running 1 test for test/unit/PWNSimpleLoanRequest.t.sol:PWNSimpleLoanRequest_RevokeRequestNonce_Test
[PASS] test_shouldCallRevokeRequestNonce() (gas: 16469)
Test result: ok. 1 passed; 0 failed; finished in 344.83µs

Running 5 tests for test/unit/PWNVault.t.sol:PWNVault_ReceivedHooks_Test
[PASS] test_shouldFail_whenOnERC1155BatchReceived() (gas: 10019)
[PASS] test_shouldFail_whenOperatorIsNotVault_onERC1155Received() (gas: 9415)
[PASS] test_shouldFail_whenOperatorIsNotVault_onERC721Received() (gas: 9271)
[PASS] test_shouldReturnCorrectValue_whenOperatorIsVault_onERC1155Received() (gas: 6343)
[PASS] test_shouldReturnCorrectValue_whenOperatorIsVault_onERC721Received() (gas: 6224)
Test result: ok. 5 passed; 0 failed; finished in 622.38µs

Running 2 tests for test/unit/PWNVault.t.sol:PWNVault_PushFrom_Test
[PASS] test_shouldCallSafeTransferFrom_fromOrigin_toBeneficiary() (gas: 22785)
[PASS] test_shouldEmitEvent_VaultPushFrom() (gas: 24603)
Test result: ok. 2 passed; 0 failed; finished in 2.59ms

Running 12 tests for test/unit/PWNSimpleLoan.t.sol:PWNSimpleLoan_ClaimLOAN_Test
[PASS] test_shouldBurnLOANToken() (gas: 31248)
[PASS] test_shouldDeleteLoanData() (gas: 86714)
[PASS] test_shouldEmitEvent_LOANClaimed_whenDefaulted() (gas: 33584)
[PASS] test_shouldEmitEvent_LOANClaimed_whenRepaid() (gas: 30391)
[PASS] test_shouldFail_whenCallerIsNotLOANTokenHolder() (gas: 32729)
[PASS] test_shouldFail_whenLoanDoesNotExist() (gas: 18566)
[PASS] test_shouldFail_whenLoanIsNotRepaidNorExpired() (gas: 36524)
[PASS] test_shouldPass_whenLoanIsExpired_whenLateRepaymentDisable() (gas: 31899)
[PASS] test_shouldPass_whenLoanIsExpired_whenLateRepaymentEnable() (gas: 31924)
[PASS] test_shouldPass_whenLoanIsRepaid() (gas: 28759)
[PASS] test_shouldTransferCollateralToLender_whenLoanIsExpired() (gas: 53677)
[PASS] test_shouldTransferRepaidAmountToLender_whenLoanIsRepaid() (gas: 32173)
Test result: ok. 12 passed; 0 failed; finished in 2.56ms

Running 2 tests for test/unit/PWNDeployer.t.sol:PWNDeployer_Deploy_Test
[PASS] test_shouldDeployContract() (gas: 381024)
[PASS] test_shouldFail_whenCallerIsNotOwner() (gas: 15545)
Test result: ok. 2 passed; 0 failed; finished in 509.58µs

Running 3 tests for test/unit/PWNVault.t.sol:PWNVault_SupportsInterface_Test
[PASS] test_shouldReturnTrue_whenIERC1155Receiver() (gas: 5736)
[PASS] test_shouldReturnTrue_whenIERC165() (gas: 5632)
```

```
[PASS] test_shouldReturnTrue_whenIERC721Receiver() (gas: 5643)
Test result: ok. 3 passed; 0 failed; finished in 419.96µs

Running 16 tests for test/unit/PWNSimpleLoanSimpleOffer.t.sol:PWNSimpleLoanSimpleOffer_CreateLOANTerms_Test
[PASS] testFail_shouldNotRevokeOffer_whenIsPersistent() (gas: 136747)
[PASS] test_shouldFail_whenCallerIsNotActiveLoan() (gas: 39282)
[PASS] test_shouldFail_whenCallerIsNotBorrower_whenSetBorrower() (gas: 133128)
[PASS] test_shouldFail_whenInvalidSignature_whenContractAccount() (gas: 51774)
[PASS] test_shouldFail_whenInvalidSignature_whenEOA() (gas: 146633)
[PASS] test_shouldFail_whenOfferIsExpired() (gas: 130130)
[PASS] test_shouldFail_whenOfferIsRevoked() (gas: 131812)
[PASS] test_shouldFail_whenPassingInvalidOfferData() (gas: 21437)
[PASS] test_shouldPass_whenOfferHasBeenMadeOnchain() (gas: 52173)
[PASS] test_shouldPass_whenOfferHasNoExpiration() (gas: 131392)
[PASS] test_shouldPass_whenOfferIsNotExpired() (gas: 134292)
[PASS] test_shouldPass_whenValidSignature_whenContractAccount() (gas: 57482)
[PASS] test_shouldPass_withValidSignature_whenEOA_whenCompactEIP2098Signature() (gas: 130821)
[PASS] test_shouldPass_withValidSignature_whenEOA_whenStandardSignature() (gas: 152979)
[PASS] test_shouldReturnCorrectValues() (gas: 133839)
[PASS] test_shouldRevokeOffer_whenIsNotPersistent() (gas: 134387)
Test result: ok. 16 passed; 0 failed; finished in 7.55ms

Running 2 tests for test/unit/PWNVault.t.sol:PWNVault_Permit_Test
[PASS] testFail_shouldNotCallPermit_whenPermitIsZero() (gas: 14337)
[PASS] test_shouldCallPermit_whenPermitNonZero() (gas: 19081)
Test result: ok. 2 passed; 0 failed; finished in 560.79µs

Running 2 tests for test/unit/PWNVault.t.sol:PWNVault_Pull_Test
[PASS] test_shouldCallTransferFrom_fromOrigin_toVault() (gas: 19966)
[PASS] test_shouldEmitEvent_VaultPull() (gas: 21451)
Test result: ok. 2 passed; 0 failed; finished in 578.71µs

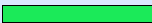


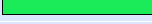

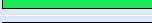

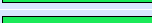
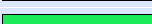
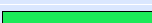
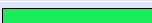

Running 10 tests for test/unit/PWNSimpleLoan.t.sol:PWNSimpleLoan_CreateLoan_Test
[PASS] test_shouldEmitEvent_LOANCreated() (gas: 189324)
[PASS] test_shouldFailWhenCollateralAssetIsInvalid() (gas: 73618)
[PASS] test_shouldFail_whenLoanFactoryContractIsNotTaggerInPWNHub() (gas: 22683)
[PASS] test_shouldGetLOANTermsStructFromGivenFactoryContract() (gas: 478016)
[PASS] test_shouldMintLOANToken() (gas: 167980)
[PASS] test_shouldReturnCreatedLoanId() (gas: 162441)
[PASS] test_shouldStoreLoanData() (gas: 262096)
[PASS] test_shouldTransferCollateral_fromBorrower_toVault() (gas: 320792)
[PASS] test_shouldTransferLoanAsset_fromLender_toBorrowerAndFeeCollector_whenNonZeroFee() (gas: 314992)
[PASS] test_shouldTransferLoanAsset_fromLender_toBorrower_whenZeroFees() (gas: 306430)
Test result: ok. 10 passed; 0 failed; finished in 15.95ms

Running 12 tests for test/integration/PWNSimpleLoanIntegration.t.sol:PWNSimpleLoanIntegrationTest
[PASS] test_shouldClaimDefaultedLOAN() (gas: 491571)
[PASS] test_shouldClaimRepaidLOAN() (gas: 561866)
[PASS] test_shouldCreateLOAN_fromListOffer() (gas: 508982)
[PASS] test_shouldCreateLOAN_fromSimpleOffer() (gas: 504189)
[PASS] test_shouldCreateLOAN_fromSimpleRequest() (gas: 503119)
[PASS] test_shouldCreateLOAN_withCryptoKittiesCollateral() (gas: 165)
[PASS] test_shouldCreateLOAN_withERC1155Collateral() (gas: 521980)
[PASS] test_shouldCreateLOAN_withERC20Collateral() (gas: 499191)
[PASS] test_shouldCreateLOAN_withERC721Collateral() (gas: 497582)
[PASS] test_shouldFailToRepayLoan_whenLOANExpired() (gas: 534879)
[PASS] test_shouldRepayLoan_whenExpired_withLateRepaymentEnabled() (gas: 553307)
[PASS] test_shouldRepayLoan_whenNotExpired() (gas: 548438)
Test result: ok. 12 passed; 0 failed; finished in 79.93ms

Running 2 tests for test/unit/PWNVault.t.sol:PWNVault_Push_Test
[PASS] test_shouldCallSafeTransferFrom_fromVault_toBeneficiary() (gas: 20117)
[PASS] test_shouldEmitEvent_VaultPush() (gas: 21454)
Test result: ok. 2 passed; 0 failed; finished in 483.58µs

Running 1 test for
↳ test/integration/PWNSimpleLoanSimpleRequestIntegration.t.sol:PWNSimpleLoanSimpleRequestIntegrationTest
[PASS] test_shouldRevokeRequestsInGroup_whenAcceptingOneFromGroup() (gas: 563807)
Test result: ok. 1 passed; 0 failed; finished in 5.24ms
```


9.3 Code Coverage

Directory	Line Coverage ↕	Functions ↕	Branches ↕
config	 100.0 % 17 / 17	100.0 % 7 / 7	- 0 / 0
deployer	 100.0 % 4 / 4	100.0 % 3 / 3	- 0 / 0
hub	 100.0 % 9 / 9	100.0 % 3 / 3	- 0 / 0
loan	 100.0 % 16 / 16	100.0 % 8 / 8	- 0 / 0
loan/lib	 13.6 % 3 / 22	50.0 % 1 / 2	- 0 / 0
loan/terms/simple/factory/offer	 100.0 % 48 / 48	100.0 % 8 / 8	- 0 / 0
loan/terms/simple/factory/offer/base	 100.0 % 5 / 5	100.0 % 2 / 2	- 0 / 0
loan/terms/simple/factory/request	 100.0 % 21 / 21	100.0 % 4 / 4	- 0 / 0
loan/terms/simple/factory/request/base	 100.0 % 5 / 5	100.0 % 2 / 2	- 0 / 0
loan/terms/simple/loan	 100.0 % 73 / 73	100.0 % 8 / 8	- 0 / 0
loan/token	 100.0 % 16 / 16	100.0 % 5 / 5	- 0 / 0
nonce	 100.0 % 12 / 12	100.0 % 4 / 4	- 0 / 0

9.4 Slither

All the relevant issues raised by Slither have been incorporated into the issues described in this report.

10 About Nethermind

Founded in 2017 by a small team of world-class technologists, Nethermind builds Ethereum solutions for developers and enterprises. Boosted by a grant from the Ethereum Foundation in August 2018, our team has worked tirelessly to deliver the fastest Ethereum client in the market. Our flagship Ethereum client is all about performance and flexibility. Built on .NET core, a widespread, enterprise-friendly platform, Nethermind makes integration with existing infrastructures simple, without losing sight of stability, reliability, data integrity, and security. **Nethermind is made up of several engineering teams across various disciplines, all collaborating to realize the Ethereum roadmap, by conducting research and building high-quality tools.** Teams focus on specific areas of the Ethereum problem space. Each consists of specialists and experienced developers working alongside interns, learning the ropes in the Nethermind Internship Program. **Our mission is to gather passionate talent from around the world, and to tackle some of the blockchain's most complex problems.** Nethermind provides software solutions and services for developers and enterprises building the Ethereum ecosystem. We offer security reviews to projects built on EVM compatible chains and StarkNet. We have expertise in multiple areas of the Ethereum ecosystem, including protocol design, smart contracts (written in Solidity and Cairo), MEV, etc. We develop some of the most used tools on Starknet and one of the most used Ethereum clients. Learn more about us at <https://nethermind.io>.

Disclaimer

This report is based on the scope of materials and documentation provided by you to Nethermind in order that Nethermind could conduct the security review outlined in **1. Executive Summary** and **2. Audited Files**. The results set out in this report may not be complete nor inclusive of all vulnerabilities. Nethermind has provided the review and this report on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. Blockchain technology remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. This report does not indicate the endorsement of any particular project or team, nor guarantee its security. No third party should rely on this report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. To the fullest extent permitted by law, Nethermind disclaims any liability in connection with this report, its content, and any related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Nethermind does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and Nethermind will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.