hexens × Slash Payment

Aug.23

SECURITY REVIEW REPORT FOR SLASH PAYMENT

CONTENTS

CONTENTS

ABOUT HEXENS

Hexens is a cybersecurity company that strives to elevate the standards of security in Web 3.0, create a safer environment for users, and ensure mass Web 3.0 adoption.

Hexens has multiple top-notch auditing teams specialized in different fields of information security, showing extreme performance in the most challenging and technically complex tasks, including but not limited to: Infrastructure Audits, Zero Knowledge Proofs / Novel Cryptography, DeFi and NFTs. Hexens not only uses widely known methodologies and flows, but focuses on discovering and introducing new ones on a day-to-day basis.

In 2022, our team announced the closure of a $4.2 million seed round led by IOSG Ventures, the leading Web 3.0 venture capital. Other investors include Delta Blockchain Fund, Chapter One, Hash Capital, ImToken Ventures, Tenzor Capital, and angels from Polygon and other blockchain projects.

Since Hexens was founded in 2021, it has had an impressive track record and recognition in the industry: Mudit Gupta - CISO of Polygon Technology - the biggest EVM Ecosystem, joined the company advisory board after completing just a single cooperation iteration. Polygon Technology, 1inch, Lido, Hats Finance, Quickswap, Layerswap, 4K, RociFi, as well as dozens of DeFi protocols and bridges, have already become our customers and taken proactive measures towards protecting their assets.

AUDIT LED BY

KASPER ZWIJSEN

Head of Smart Contract Audits | Hexens

Audit Starting Date
21.08.2023

Audit Completion Date
31.08.2023

hexens × Slash Payment

METHODOLOGY

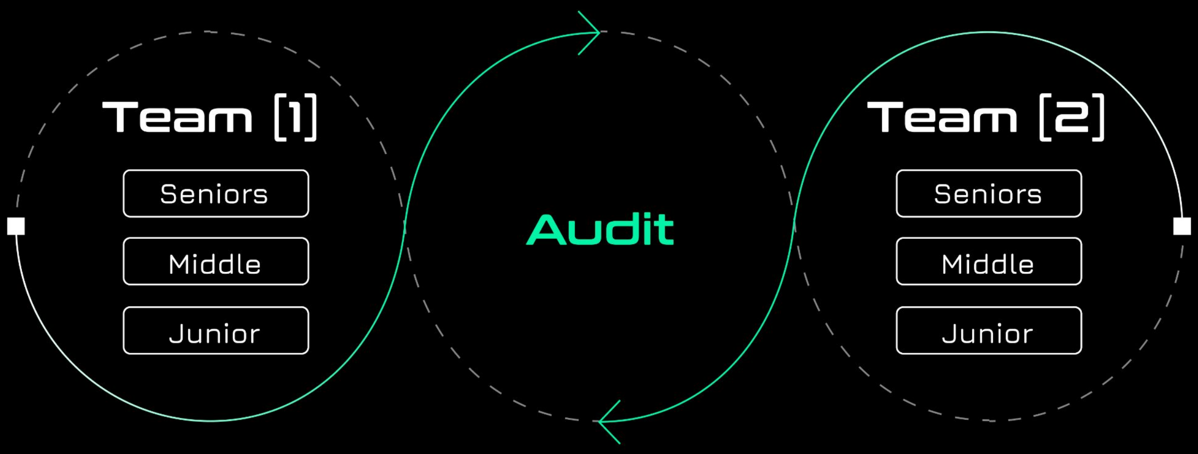
COMMON AUDIT PROCESS

Companies often assign just one engineer to one security assessment with no specified level. Despite the possible impeccable skills of the assigned engineer, it carries risks of the human factor that can affect the product's lifecycle.

Auditor*
Audit

HEXENS METHODOLOGY

Hexens methodology involves 2 teams, including multiple auditors of different seniority, with at least 5 security engineers. This unique cross-checking mechanism helps us provide the best quality in the market.



SEVERITY STRUCTURE

The vulnerability severity is calculated based on two components

- Impact of the vulnerability
- Probability of the vulnerability

IMPACT	PROBABILITY			
	Rare	Unlikely	Likely	Very Likely
Low / Info	Low / Info	Low / Info	Medium	Medium
Medium	Low / Info	Medium	Medium	High
High	Medium	Medium	High	Critical
Critical	Medium	High	Critical	Critical

SEVERITY CHARACTERISTICS

Vulnerabilities can range in severity and impact, and it's important to understand their level of severity in order to prioritize their resolution. Here are the different types of severity levels of vulnerabilities:

CRITICAL

Vulnerabilities with this level of severity can result in significant financial losses or reputational damage. They often allow an attacker to gain complete control of a contract, directly steal or freeze funds from the contract or users, or permanently block the functionality of a protocol. Examples include infinite mints and governance manipulation.

HIGH

Vulnerabilities with this level of severity can result in some financial losses or reputational damage. They often allow an attacker to directly steal yield from the contract or users, or temporarily freeze funds. Examples include inadequate access control integer overflow/underflow, or logic bugs.

MEDIUM

Vulnerabilities with this level of severity can result in some damage to the protocol or users, without profit for the attacker. They often allow an attacker to exploit a contract to cause harm, but the impact may be limited, such as temporarily blocking the functionality of the protocol. Examples include uninitialized storage pointers and failure to check external calls.

LOW

Vulnerabilities with this level of severity may not result in financial losses or significant harm. They may, however, impact the usability or reliability of a contract. Examples include slippage and front-running, or minor logic bugs.

INFORMATIONAL

Vulnerabilities with this level of severity are regarding gas optimizations and code style. They often involve issues with documentation, incorrect usage of EIP standards, best practices for saving gas, or the overall design of a contract. Examples include not conforming to ERC20, or disagreement between documentation and code.

It's important to consider all types of vulnerabilities, including informational ones, when assessing the security of the project. A comprehensive security audit should consider all types of vulnerabilities to ensure the highest level of security and reliability.

EXECUTIVE SUMMARY

OVERVIEW

This audit covered the new V2 NFT Vault contracts as developed by Slash. This contracts allow for deposits through NFTs and offers integration with Slash Payment for both users and merchants.

Our security assessment was a full review of the smart contracts, spanning a total of 2 weeks.

During our audit, we have identified 2 High severity vulnerabilities. These vulnerabilities could allow for stealing of funds from users that make use of gasless transactions using the Forwarder.

We have also identified 5 Medium severity vulnerabilities, various minor vulnerabilities and code optimisations.

Finally, all of our reported issues were fixed or acknowledged by the development team and consequently validated by us.

We can confidently say that the overall security and code quality of the project have increased after completion of our audit.

SCOPE

The analyzed resources were sent in an archive with the following SHA256 hash:
81829474d325618e7361268dde1b069e54eff368a9d6df27963d0ae0e69cf332

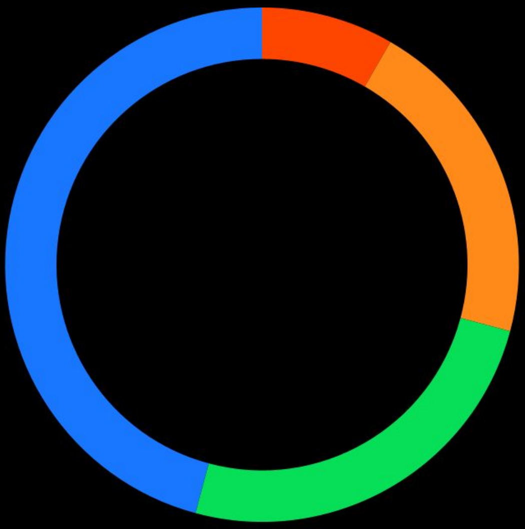
The issues described in this report were fixed. Corresponding commits are mentioned in the description.

SUMMARY

SEVERITY	NUMBER OF FINDINGS
CRITICAL	0
HIGH	2
MEDIUM	5
LOW	6
INFORMATIONAL	11

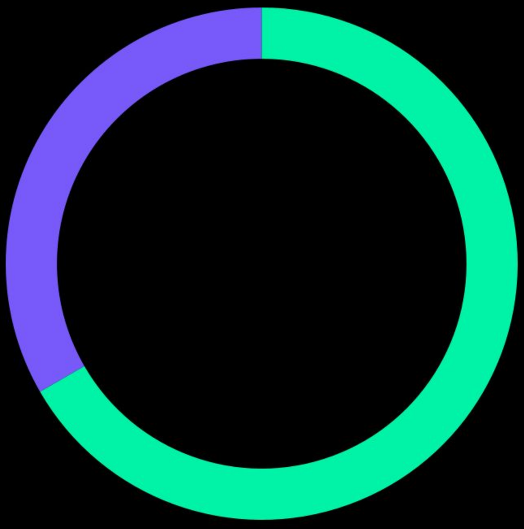
TOTAL: 24

SEVERITY



● High ● Medium ● Low ● Informational

STATUS



● Fixed ● Acknowledged

WEAKNESSES

This section contains the list of discovered weaknesses.

SLSF-21. FORWARDER SOFT ERROR CAN LEAD TO STOLEN ETH AND CENSORED TRANSACTIONS

SEVERITY: High

PATH: Forwarder.sol:execute:L45-66

REMEDIATION: remove the soft error and instead do a full revert if the call returns false. The requesting user should have some way to increase their nonce, so the reverting request does not block any future requests of the user

STATUS: fixed

DESCRIPTION:

The **execute** function of the Forwarder won't revert if the forwarded call reverts, instead it only emits an event.

In the case where the request has some value (and assuming **execute** is now **payable**), then if the forwarded call reverts by means of the requester, attacker or other unrelated causes, then the ETH that was sent will be left in the Forwarder contract. That ETH is then up for anyone to steal with another call to **execute** and a self-signed request.

Furthermore, the **execute** function will increase the nonce even if the forwarded call fails. Any attacker can therefore execute a signer forward request but force a revert (e.g. by specifying too little gas in the actual transaction). The nonce would still be increased and the forward request can no longer be executed, censoring the user's forward request.

```
function execute(
    ForwardRequest calldata req,
    bytes calldata sig
) external override returns (bool success, bytes memory ret) {
    if (req.validUntilTime < block.timestamp) {
        revert RequestExpired(req.validUntilTime);
    }
    if (nonces[req.from]++ != req.nonce) {
        revert InvalidNonce(req.from, req.nonce);
    }
    _verifySig(req, sig);

    (success, ret) = req.to.call{gas: req.gas, value: req.value}(
        abi.encodePacked(req.data, req.from)
    );

    if (!success) {
        emit ExecutionFailed(req.from, req.to, req.nonce, ret);
    }

    return (success, ret);
}
```

SLSF-28. LACK OF TARGET AND FUNCTION WHITELIST IN FORWARDER

SEVERITY: High

PATH: Forwarder.sol

REMEDIATION: to mitigate these security risks, it is recommended to add a whitelist to the Forwarder contract so that only trusted values are allowed for req.to and the function signature inside req.data

STATUS: fixed

DESCRIPTION:

The **execute** method in the **Forwarder** contract does not impose any restrictions or sanitize the **req.to** and **req.data** parameters that are subsequently used in the external call. This lack of parameter limitation or sanitization in the **execute** method allows threat actors to exploit the following scenarios:

Scenario 1. Currently, the **NftVault** contract always withdraws tokens from **msg.sender** when a deposit occurs. This behavior is described in the SLSF-20 issue, which can be observed in lines 165-166 of **NftVault** and line 80 of **UniversalERC20**. Consequently, if a trusted forwarder is used for **NftVault**, tokens will be transferred from the **Forwarder** contract balance to **NftVault** during the deposit. Therefore, the user must transfer funds to the **Forwarder** contract directly, or there should be a preliminary call to **execute()** of the **Forwarder** to transfer approved funds from the user to the

Forwarder contract. In either case, a threat actor can front-run the main call to **execute()** while funds are sitting in the **Forwarder** balance and steal them by calling the **transfer()** function for the token.

Scenario 2. A user may purposely or inadvertently approve some amount of token for the **Forwarder** contract. This allows a threat actor to steal the approved amount by calling the **transferFrom()** function for the token utilizing the **execute()** method of the **Forwarder** contract.

Scenario 3. Currently, the **NftVault** contract allows the deposit of any ERC20 token unless it is explicitly blacklisted in the **_tokenBlacklist** set. There is a wide variety of ERC-777 tokens, and a user may want to deposit such a token in the vault. Again, if a trusted forwarder is used with **NftVault**, a threat actor can call **execute()** for the **Forwarder** contract, subsequently calling the ERC1820 Registry to register a hook for all ERC-777 tokens on behalf of the Forwarder contract. As a result, every time the **Forwarder** contract receives or sends ERC-777 tokens, an attacker-controlled hook will be called. The attacker may choose to revert inside the hook, thus censoring the usage of ERC-777 tokens for the **Forwarder**.

```
function execute(
    ForwardRequest calldata req,
    bytes calldata sig
) external override returns (bool success, bytes memory ret) {
    if (req.validUntilTime < block.timestamp) {
        revert RequestExpired(req.validUntilTime);
    }
    if (nonces[req.from]++ != req.nonce) {
        revert InvalidNonce(req.from, req.nonce);
    }
    _verifySig(req, sig);

    (success, ret) = req.to.call{gas: req.gas, value: req.value}(
        abi.encodePacked(req.data, req.from)
    );

    if (!success) {
        emit ExecutionFailed(req.from, req.to, req.nonce, ret);
    }

    return (success, ret);
}
```

SLSF-2. NFT VAULT DOES NOT HANDLE REBASE TOKENS

SEVERITY: Medium

PATH: NftVault.sol

REMEDIATION: in the current implementation it would be rather difficult to track which tokens are rebase tokens and correctly assign the token amounts to those users

one solution could be to instead of saving the absolute amount of deposited tokens, it can save an amount of shares for that token. So when a user wants to withdraw, the correct amount of tokens is calculated using the user's share balance, the total amount of shares and the current balance of the specific token. A share is also token-specific

STATUS: acknowledged, see commentary

DESCRIPTION:

The NFT Vault allows for depositing any ERC20 token and the balances are tracked using the **IterableLock** library. The balances are the exact tokens that were sent during the deposit and only this exact amount can consequently be withdrawn.

However, rebase tokens like Lido's stETH or Aave's aToken distribute rewards through rebasing, which increases the token holders' balance automatically.

In this case, the extra balance would be unaccounted for, the rewards would be lost by the users and can only be withdrawn using the owner's recovery mechanism.

```
function _deposit(
    address erc20Token_,
    address nftAddress_,
    uint256 nftTokenId_,
    uint256 amount_,
    uint64 unlockAt_
) internal {
    uint256 buffer = 1825 days; // 5 years
    if (unlockAt_ > block.timestamp + buffer) revert InvalidUnlockTime();
    if (tokenInBlacklist(erc20Token_)) revert InvalidToken(erc20Token_);
    if (
        !IDepositValidator(_depositValidator).isValid(
            _msgSender(),
            nftAddress_,
            nftTokenId_
        ) ||
        (_isUseNftWhitelist && !_nftWhitelist.contains(nftAddress_)) ||
        tokenIdInBlacklist(nftAddress_, nftTokenId_)
    ) revert InvalidNft();

    ItLock storage itLock = _balances[
        _hash3(nftAddress_, nftTokenId_, erc20Token_)
    ];

    // Validate the condition for the time-lock
    if (
        unlockAt_ > block.timestamp &&
        itLock.exists(unlockAt_) &&
        IERC721(nftAddress_).ownerOf(nftTokenId_) != _msgSender()
    ) revert AlreadyLocked();

    // Iterate the locks and mark the expired locks as unlocked
    itLock.deposit(unlockAt_, amount_);

    ItAddressMap storage itAddressMap = _lockedTokens[
        _hash2(nftAddress_, nftTokenId_)
    ];
    itAddressMap.insert(erc20Token_);

    _balancePerToken[erc20Token_] += amount_;
    emit Deposited(
        erc20Token_,
        nftAddress_,
        nftTokenId_,
        amount_,
        unlockAt_
    );
}
```

Commentary from the client:

“ - Document the official stance to state we do not support rebase tokens, and to use wrapped rebase tokens if required.
Also, for rebase, there are 2 types, positive and negative rebase.
For positive rebase, users do not lose any tokens, and any excess tokens from the rebase can be withdrawn by the vault owner.
For negative rebase, there would be insufficient tokens for 100% of the users to withdraw. However, this could be the same issue for any token contract that has a backdoor which can reduce the token balance of any address."

SLSF-4. MAXIMUM LOCK TIME IS NOT ENFORCED IN LOCK AND RELOCK

SEVERITY: Medium

PATH: NftVault.sol:lock, relock (L309-332, L336-359)

REMEDIATION: enforce the same maximum lock time as in _deposit in both lock and relock

STATUS: fixed

DESCRIPTION:

In the NFT Vault, the **deposit** function limits the user's lock time to a maximum of 5 years on lines 116-117:

```
uint256 buffer = 1825 days; // 5 years
if (unlockAt_ > block.timestamp + buffer) revert InvalidUnlockTime();
```

However, the **lock** and **relock** functions do not have this limit and the user can create or extend locks to maximum **uint64**.

If there is some reward or governance power derived from the balance and lock time, then a small balance with an enormous lock time would cause large deviations and potential manipulation of those rewards or governance.

```
function lock(
    address erc20Token_,
    address nftAddress_,
    uint256 nftTokenId_,
    uint256 amount_,
    uint64 unlockAt_
) external {
    ItLock storage itLock = _balances[
        _hash3(nftAddress_, nftTokenId_, erc20Token_)
    ];
    // Check the user is the honest holder
    if (IERC721(nftAddress_).ownerOf(nftTokenId_) != _msgSender())
        revert Unpermitted();
    itLock.lock(unlockAt_, amount_);

    emit Locked(
        _msgSender(),
        erc20Token_,
        nftAddress_,
        nftTokenId_,
        amount_,
        unlockAt_
    );
}

/// @notice Relock the tokens with new lock duration
/// @dev Only nft holder can relock tokens
function relock(
    address erc20Token_,
    address nftAddress_,
    uint256 nftTokenId_,
    uint64 lockId_,
    uint64 unlockAt_
) external {
    ItLock storage itLock = _balances[
        _hash3(nftAddress_, nftTokenId_, erc20Token_)
    ];
    // Check the user is the honest holder
    if (IERC721(nftAddress_).ownerOf(nftTokenId_) != _msgSender())
        revert Unpermitted();
    itLock.relock(lockId_, unlockAt_);
    emit Relocked(
        _msgSender(),
        erc20Token_,
        nftAddress_,
        nftTokenId_,
        lockId_,
        unlockAt_
    );
}
```

SLSF-8. NFT BLACKLIST IS NOT CHECKED UPON WITHDRAWAL

SEVERITY: Medium

PATH: NftVault.sol:withdrawForPayment:L220-251

REMEDIATION: add a check to verify if the token being used for payment has been blacklisted. When a user attempts to use a blacklisted token for payments, the protocol should promptly reject the transaction and provide a clear error message indicating that the token has been blacklisted. This will ensure that users can only withdraw their blacklisted tokens and cannot employ them for payments within the protocol

STATUS: acknowledged, see commentary

DESCRIPTION:

While the protocol appropriately checks for blacklisted tokens during deposits, it does not perform the same checks when users attempt to make payments using these tokens through the **withdrawForPayment** function. This means that if a user deposits a specific token into the vault, and that token subsequently becomes blacklisted, the user can still utilize the blacklisted token for making payments within the protocol.

For instance, suppose Alice has deposited a token into the vault, and this token later becomes blacklisted. Under the current implementation, Bob cannot deposit the same blacklisted token due to the checks in place. However, Bob can purchase Alice's NFT, effectively "depositing" a blacklisted token without going through the standard deposit process.

```
function withdrawForPayment(
    address account_,
    address erc20Token_,
    address nftAddress_,
    uint256 nftTokenId_,
    uint256 amount_
) external override {
    // If called via a forwarder, _msgSender() may not be a wrapper, so check with msg.sender.
    if (!_wrapperList.contains(msg.sender)) revert Unpermitted();
    bytes32 key = keccak256(abi.encodePacked(account_, msg.sender));
    if (!_withdrawForPaymentApproved[key]) revert UnapprovedWrapper();

    // Check the user who wants to make the slash payment has owned this nft token id
    // We check account_, not _msgSender() because _msgSender() is Payment wrapper contract
    if (
        IERC721(nftAddress_).ownerOf(nftTokenId_) != account_ ||
        !IWithdrawValidator(_withdrawValidator).isValid(
            account_,
            nftAddress_,
            nftTokenId_
        )
    ) revert Unpermitted();

    _withdraw(
        true,
        msg.sender, // Withdraw to Payment wrapper contract
        erc20Token_,
        nftAddress_,
        nftTokenId_,
        amount_
    );
}
```

Commentary from the client:

“ - This is intended behavior to only check for blacklist on deposits, so if blacklist is done after a deposit is done, the blacklisted owner can still withdraw, but not make new deposits for the blacklisted NFT.”

SLSF-20. INCORRECT USAGE OF UNIVERSALTRANSFERFROMSENDERTOTHIS IN NFT VAULT

SEVERITY: Medium

PATH: NftVault.sol:_transferToken:L160-180

REMEDIATION: use UniversalERC20.universalTransferFrom instead with _msgSender() as from parameter

STATUS: fixed

DESCRIPTION:

The NFT vault uses the **_transferToken** function is **deposit** and **batchDeposit** to pull in funds from the user. This function uses the **UniversalERC20** library's function **universalTransferFromSenderToThis** on line 166, which uses **msg.sender** directly to transfer tokens from.

In the case where the Forwarder is used, the msg.sender will always be the Forwarder and instead the tokens should be pulled from **_msgSender**.

In the current situation, any deposit made using the Forwarder would fail, as it would try to transfer from the Forwarder.

```
function _transferToken(address token_, uint256 amount_) internal {
    uint256 feeAmount = depositFee(token_, amount_);
    amount_ += feeAmount;

    // Transfer total deposit amount to the NftVault contract
    uint256 amountTransferred = IERC20(token_)
        .universalTransferFromSenderToThis(amount_);
    // Fee token is not supported in the NftVault
    if (amountTransferred < amount_)
        revert InsufficientTransfer(amount_, amountTransferred);

    // Transfer fee to the treasury account
    if (feeAmount == 0) return;
    uint256 feeTransferred = IERC20(token_).universalTransfer(
        _treasury,
        feeAmount
    );

    if (feeTransferred < feeAmount)
        revert InsufficientTransfer(feeAmount, feeTransferred);
}

function universalTransferFromSenderToThis(
    IERC20 token,
    uint256 amount
) internal returns (uint256) {
    if (amount == 0) {
        return 0;
    }

    if (isETH(token)) {
        require(msg.value >= amount, "Insufficient msg.value");
        // Return remainder if exist
        if (msg.value > amount)
            payable(msg.sender).sendValue(msg.value - amount);
        return amount;
    }
    uint256 balanceBefore = token.balanceOf(address(this));
    token.safeTransferFrom(msg.sender, address(this), amount);
    return token.balanceOf(address(this)) - balanceBefore;
}
```

SLSF-29. SLASH PAYMENTS INTO NFT VAULT CAN BE BLOCKED

SEVERITY: Medium

PATH: NftVaultSlashExtension.sol:receivePayment:L35-63

REMEDIATION: it could be possible to allow for deposits into any existing lock, as there is no harm in donating free tokens and it would not increase the lock time

another solution would be to give a privileged role to the Slash extension contract to be allowed to deposit into existing locks

STATUS: fixed

DESCRIPTION:

The Slash payment core contract will call the **receivePayment** function to deposit paid assets into the NFT vault for a merchant (or potentially as cash back for users). This will call the **batchDeposit** function of the NFT vault, which internally calls **_deposit**.

In the **_deposit** function, the vault checks that if the lock already exists, that the sender is indeed the owner on lines 134-138:

```
if (
    unlockAt_ > block.timestamp &&
    itLock.exists(unlockAt_) &&
    IERC721(nftAddress_).ownerOf(nftTokenId_) != _msgSender()
) revert AlreadyLocked();
```

The Slash extension contract is not the owner of the merchant's NFT, but the **unlockAt_** parameter will be a timestamp and so this call will normally succeed.

However, an attacker can front-run such a payment call (or pre-calculate if deterministic) an deposit the smallest amount possible (e.g. 1 Wei) in the the NFT token ID and timestamp of the merchant. This will cause the call to **receivePayment** to revert with the **AlreadyLocked** error, effectively blocking the payment.

```
function _makeDeposit(
    address token_,
    uint256 amount_,
    bytes calldata reserved_
) internal returns (uint256) {
    // Extract params from `reserved_` arg. They are in abi-encoded format.
    (
        address[] memory nftAddresses,
        uint256[] memory nftTokenIds,
        uint256[] memory amounts,
        uint64 unlockAt
    ) = abi.decode(reserved_, (address[], uint256[], uint256[], uint64));
    INftVault nftVault_ = INftVault(_nftVault);
    IERC20(token_).universalApprove(address(nftVault_), amount_);

    uint256 balanceBefore = IERC20(token_).universalBalanceOf(
        address(this)
    );
    {
        nftVault_.batchDeposit{value: IERC20(token_).isETH() ? amount_ : 0}(
            token_,
            nftAddresses,
            nftTokenIds,
            amounts,
            unlockAt
        );
    }

    return balanceBefore - IERC20(token_).universalBalanceOf(address(this));
}
```

SLSF-6. BLACKLISTED NFT ADDRESS CAN STILL BE USED TO DEPOSIT

SEVERITY: Low

PATH: NftVault.sol:_deposit:L109-157

REMEDIATION: in the case that blacklisting or a full collection is desired, then we would recommend to check the nftAddress against the _nftBlacklist in tokenIdInBlacklist such that a fully blacklisted NFT cannot be used to deposit. Similarly, in updateTokenIdBlacklist should not add the NFT address to the blacklist if blacklisting of a single specific ID is desired

otherwise, the list _nftBlacklist and setter function updateNftBlacklist are redundant, as _nftBlacklist is not used anywhere else

STATUS: acknowledged, see commentary

DESCRIPTION:

The **BaseVault** contracts exposes functionality to both blacklist and whitelist NFT token addresses, blacklist ERC20 token addresses and blacklist specific NFT token IDs.

However, we found that the **_nftBlacklist** list is not used at all in the **NftVault** contract to check for blacklisted NFTs during a deposit.

For example, if the owner blacklists some **nftAddress_** using the function **updateNftBlacklist**, then this should mean that all token IDs can't be used in the deposit function. But this is not checked in **_deposit**, instead it only checks the NFT token ID against the blacklist on line 126 using **tokenIdInBlacklist**:

```
/// @notice Check tokenId is blacklisted or not
  function tokenIdInBlacklist(
      address nftAddress_,
      uint256 tokenId_
  ) public view returns (bool) {
      return _tokenIdBlacklist[nftAddress_].contains(tokenId_);
  }
```

Furthermore, if a specific NFT token ID gets blacklisted, then the NFT token address gets added to the **_nftBlacklist**, which if the above check was done correctly would result in all token IDs getting blacklisted.

```
  /// @notice Add / remove the exact tokenId in nft collection for the blacklist deposit
  function updateTokenIdBlacklist(
      address nftAddress_,
      uint256 tokenId_,
      bool flag_
  ) external onlyOwner {
      if (_tokenIdBlacklist[nftAddress_].contains(tokenId_) == flag_)
          revert AlreadyConfigured();

      if (flag_) {
          _tokenIdBlacklist[nftAddress_].add(tokenId_);
          if (!_nftBlacklist.contains(nftAddress_))
              _nftBlacklist.add(nftAddress_);
      } else {
          _tokenIdBlacklist[nftAddress_].remove(tokenId_);
          if (_tokenIdBlacklist[nftAddress_].length() == 0)
              _nftBlacklist.remove(nftAddress_);
      }

      emit TokenIdBlacklistUpdated(nftAddress_, tokenId_, flag_);
  }
```

This makes either the check incomplete or the **_nftBlacklist** list redundant.

```
/// @notice Update balance data for the deposit operation
  function _deposit(
    address erc20Token_,
    address nftAddress_,
    uint256 nftTokenId_,
    uint256 amount_,
    uint64 unlockAt_
  ) internal {
    uint256 buffer = 1825 days; // 5 years
    if (unlockAt_ > block.timestamp + buffer) revert InvalidUnlockTime();
    if (tokenInBlacklist(erc20Token_)) revert InvalidToken(erc20Token_);
    if (
      !IDepositValidator(_depositValidator).isValid(
        _msgSender(),
        nftAddress_,
        nftTokenId_
      ) ||
      (_isUseNftWhitelist && !_nftWhitelist.contains(nftAddress_)) ||
      tokenIdInBlacklist(nftAddress_, nftTokenId_)
    ) revert InvalidNft();

    ItLock storage itLock = _balances[
      _hash3(nftAddress_, nftTokenId_, erc20Token_)
    ];

    // Validate the condition for the time-lock
    if (
      unlockAt_ > block.timestamp &&
      itLock.exists(unlockAt_) &&
      IERC721(nftAddress_).ownerOf(nftTokenId_) != _msgSender()
    ) revert AlreadyLocked();

    // Iterate the locks and mark the expired locks as unlocked
    itLock.deposit(unlockAt_, amount_);

    ItAddressMap storage itAddressMap = _lockedTokens[
      _hash2(nftAddress_, nftTokenId_)
    ];
    itAddressMap.insert(erc20Token_);

    _balancePerToken[erc20Token_] += amount_;
```

```
        emit Deposited(
            erc20Token_,
            nftAddress_,
            nftTokenId_,
            amount_,
            unlockAt_
        );
    }
```

Commentary from the client:

“ - Intended to blacklist based on collection address + token id. Not to blacklist entire collections.”

SLSF-10. TREASURY UPDATE RECEIVER CHECK IS INSUFFICIENT

SEVERITY: Low

PATH: BaseVault.sol:updateTreasury:L77-86

REMEDIATION: it is not specified if the wallet is intended to be an EOA or a smart contract wallet

assuming it is intended to be an EOA, a better mitigation would be to require the new address to sign a confirming transaction

otherwise, if the wallet address is a smart contract wallet, there should be a check to ensure address(treasury).code.length > 0 when updating the address

if the wallet could be either of of either type, the recommended fix is to require a the new address to sign a confirming transaction before finalising the update

STATUS: acknowledged, see commentary

DESCRIPTION:

The function **updateTreasury** might update the **treasury** address to a non-existent address.

Because the EVM considers a low-level **call** to a non-existing contract to always succeed, sending "**0 ETH to the new treasury address to check if it can receive ETH**" is not sufficient to ensure the address is legitimate.

```
  function updateTreasury(address payable treasury_) external onlyOwner {
    if (treasury_ == address(0)) revert InvalidZeroAddress();
    if (_treasury == treasury_) revert AlreadyConfigured();

    treasury_.sendValue(0); // We try to send 0 ETH to the new treasury address to check if it can receive ETH
    _treasury = treasury_;

    emit TreasuryUpdated(treasury_);
  }
```

From OpenZeppelin's AddressUpgradeable:

```
function sendValue(address payable recipient, uint256 amount) internal {
  if (address(this).balance < amount) {
    revert AddressInsufficientBalance(address(this));
  }

  (bool success, ) = recipient.call{value: amount}("");
  if (!success) {
    revert FailedInnerCall();
  }
}
```

Commentary from the client:

" - Even if we check for the address no have non-zero code length, reverts could still happen if the contract does not have a fallback function or reverts in it.
Also, if we add a non-zero code length check, EOA cannot be set as treasury."

SLSF-11. UINT256 COULD BE MORE GAS EFFICIENT THAN SMALLER TYPES

SEVERITY: Low

PATH: see description

REMEDIATION: refactor any smaller uint variables such as uint64 or uint16 that are not loaded in the same block to uint256. In doing so less gas will be used when accessing such variables

STATUS: fixed

DESCRIPTION:

The EVM works with 32 bytes by default, which translates to **uint256**. Smaller uint types such as **uint64** require more gas because the EVM needs to perform extra operations to cast such values.

Unless such variables are packed in the same slot also used in the same block, it is actually cheaper to use **uint256** for most use cases.

For example, in the **NftVault** contract the **_batchTxLimit**, **_depositFee** and **_withdrawFee** are both **uint16** and so might share a slot. However, they are not used in the same functions and so an extra conversion is required each time and no **SLOAD** is saved anywhere, increasing the gas cost.

```
/// @notice Array limit in the batch tx functions. Default 100 items
uint16 internal _batchTxLimit = 20;
/// @notice deposit fee percentage. denominator 10000
uint16 internal _depositFee;
/// @notice Withdraw fee percentage. denominator 10000
uint16 internal _withdrawFee;
```

SLSF-15. VAULT CONTRACT SUPPORTS ANY TYPE OF TOKENS INSTEAD OF SPECIFIC TOKENS

SEVERITY: Low

PATH: NftVault.sol

REMEDIATION: consider a whitelist for ERC20 tokens with the purpose of having screened tokens and clearly outline which token standards are compatible with the NFT Vault contract

STATUS: acknowledged, see commentary

DESCRIPTION:

The current NFT Vault contract documentation outlines the protocol's ability to support a short list of tokens, including native tokens from multiple blockchains:

"The native tokens of each of the 4 chains mentioned above, USDT, USDC, DAI, and JPYC can be deposited to Vault Contracts. The tokens stored on the vault contract can only be withdrawn with a transaction by the NFT holder. There is no platform fee for the deposit, but a small network fee (gas fee) is needed for the transaction to be executed."

While the protocol should only support some list of tokens, in **deposit** and **batchDeposit** functions it is possible to pass any ERC20 token, if it isn't included in **_tokenBlacklist**.

```
/// @notice Update balance data for the deposit operation
  function _deposit(
      address erc20Token_,
      address nftAddress_,
      uint256 nftTokenId_,
      uint256 amount_,
      uint64 unlockAt_
  ) internal {
      uint256 buffer = 1825 days; // 5 years
      if (unlockAt_ > block.timestamp + buffer) revert InvalidUnlockTime();
      if (tokenInBlacklist(erc20Token_)) revert InvalidToken(erc20Token_);
      if (
         !IDepositValidator(_depositValidator).isValid(
            _msgSender(),
            nftAddress_,
            nftTokenId_
         ) ||
         (_isUseNftWhitelist && !_nftWhitelist.contains(nftAddress_)) ||
         tokenIdInBlacklist(nftAddress_, nftTokenId_)
      ) revert InvalidNft();

      ItLock storage itLock = _balances[
         _hash3(nftAddress_, nftTokenId_, erc20Token_)
      ];

      // Validate the condition for the time-lock
      if (
         unlockAt_ > block.timestamp &&
         itLock.exists(unlockAt_) &&
         IERC721(nftAddress_).ownerOf(nftTokenId_) != _msgSender()
      ) revert AlreadyLocked();

      // Iterate the locks and mark the expired locks as unlocked
      itLock.deposit(unlockAt_, amount_);

      ItAddressMap storage itAddressMap = _lockedTokens[
         _hash2(nftAddress_, nftTokenId_)
      ];
      itAddressMap.insert(erc20Token_);

      _balancePerToken[erc20Token_] += amount_;
```

```
        emit Deposited(
            erc20Token_,
            nftAddress_,
            nftTokenId_,
            amount_,
            unlockAt_
        );
    }
```

Commentary from the client:

“ - No whitelist for ERC20 tokens by default, only blacklist."

SLSF-18. FORWARDER SUPPORTS CALL VALUE BUT HAS NO WAY OF RECEIVING ETH

SEVERITY: Low

PATH: Forwarder.sol:execute:L45-66

REMEDIATION: make the execute a payable function and check the msg.value against the req.value to make sure that the exact value has been sent

STATUS: fixed

DESCRIPTION:

The Forwarder contract supports setting **msg.value** in the ForwardRequest and also sets the value in the external call on line 57.

However, the function **execute** is not payable nor does the contract have a receive or fallback function. As a result, the forwarder has no way to use ETH in forwarded calls and all those requests will fail.

```
function execute(
    ForwardRequest calldata req,
    bytes calldata sig
) external override returns (bool success, bytes memory ret) {
    if (req.validUntilTime < block.timestamp) {
        revert RequestExpired(req.validUntilTime);
    }
    if (nonces[req.from]++ != req.nonce) {
        revert InvalidNonce(req.from, req.nonce);
    }
    _verifySig(req, sig);

    (success, ret) = req.to.call{gas: req.gas, value: req.value}(
        abi.encodePacked(req.data, req.from)
    );

    if (!success) {
        emit ExecutionFailed(req.from, req.to, req.nonce, ret);
    }

    return (success, ret);
}
```

SLSF-27. FLOATING PRAGMA

SEVERITY: Low

PATH: see description

REMEDIATION: use the specific version of Solidity 0.8.19, which is compatible with other EVM chains and does not implement the new opcode opcode PUSH0

STATUS: fixed

DESCRIPTION:

Solidity version 0.8.20 introduces a new opcode **PUSH0**, which is currently, not compatible with any EVM chains other than Ethereum. According to Slash Project White Paper Ver.2.0 page 10: “Slash Vaults supports whitelisted NFTs on ETH, BNB, Avalanche C-chain, and Polygon blockchains as of December 2022”.

Using a floating pragma might lead to the deployment of contracts using solidity 0.8.20 to EVM chains other than Ethereum. This would lead to malfunction of protocol, rendering it mostly useless on most chains.

```
pragma solidity ^0.8.0;
```

SLSF-3. CONSTANT VARIABLES SHOULD BE MARKED AS PRIVATE

SEVERITY: Informational

PATH: BaseVault.sol

REMEDIATION: see description

STATUS: fixed

DESCRIPTION:

The MAX_DEPOSIT_FEE and MAX_WITHDRAW_FEE parameters on lines 18 and 20 should be **private**. Setting constants to private will save deployment gas. This is because the compiler won't have to create non-payable getter functions for deployment calldata, won't need to store the bytes of the values outside of where it's used, and won't add another entry to the method ID table. The values can still be read from the verified contract source code if necessary.

```
/// @notice max deposit fee 10%
uint16 public constant MAX_DEPOSIT_FEE = 1000;
/// @notice max withdraw fee 10%
uint16 public constant MAX_WITHDRAW_FEE = 1000;
```

SLSF-5. DOCUMENTATION INCONSISTENCIES

SEVERITY: Informational

PATH: see description

REMEDIATION: fix the documentation inconsistencies

STATUS: fixed

DESCRIPTION:

We have identified various locations where implementation is inconsistent with the documentation:

1. According to the protocol documentation, the **fee** should not be charged while depositing in the **NftVault**. However, upon calling function **deposit()**, it subsequently invokes function **_transferToken()**, within which a procedure that levies **fee** is present.
2. In the **BaseVault** contract, the constant **_batchTxLimit** is set to 20, however the documentation comment string specifies a limit of 100.
3. The function **WithdrawValidator:isValid** function makes no use of the parameter **owner**.

However, the documentation says it should check if the owner holds the NFT. Instead this check is implemented upstream in **NftVault.withdraw**.

1.

```
function _transferToken(address token_, uint256 amount_) internal {
    uint256 feeAmount = depositFee(token_, amount_);
    amount_ += feeAmount;

    // Transfer total deposit amount to the NftVault contract
    uint256 amountTransferred = IERC20(token_)
        .universalTransferFromSenderToThis(amount_);
    // Fee token is not supported in the NftVault
    if (amountTransferred < amount_)
        revert InsufficientTransfer(amount_, amountTransferred);

    // Transfer fee to the treasury account
    if (feeAmount == 0) return;
    uint256 feeTransferred = IERC20(token_).universalTransfer(
        _treasury,
        feeAmount
    );

    if (feeTransferred < feeAmount)
        revert InsufficientTransfer(feeAmount, feeTransferred);
}
```

2.

```
/// @notice Array limit in the batch tx functions. Default 100 items
uint16 internal _batchTxLimit = 20;
```

3.

```
/// @notice Check if the owner is able to withdraw tokens from nft vault
/// @dev First, the nft must be ERC721-standard
/// @dev Second, `owner` must hold the given `tokenId` of the `nftAddress`
/// @dev Third, `nftAddress` & `tokenId` must not denied from withdrawal
function isValid(
    address owner_,
    address nftAddress_,
    uint256 tokenId_
) external view returns (bool) {
    IERC721Upgradeable nft = IERC721Upgradeable(nftAddress_);
    // This function may be called with contract address which does not have supportsInterface function
    try nft.supportsInterface(0x80ac58cd) returns (bool result) {
        if (!result) return false;
    } catch {
        return false;
    }

    return withdrawEnabled(nftAddress_, tokenId_);
}
```

SLSF-12. MAGIC NUMBERS SHOULD BE REPLACED WITH CONSTANTS

SEVERITY: Informational

PATH: see description

REMEDIATION: change hard coded values such as 10_000 with a constant that explains the purpose of such variable, for example as BASIS_POINTS_DENOMINATOR in BaseVault function withdrawFee line 173

STATUS: fixed

DESCRIPTION:

Magic numbers, e.g **10_000** hurt readability and use more gas.

Replacing them with a constant like BASIS_POINTS_DENOMINATOR improves readability and gas cost.

```
function withdrawFee(
    address token_,
    uint256 amount_
) public view override returns (uint256) {
    if (_withdrawFeeExemptedTokenList[token_]) return 0;
    return (amount_ * _withdrawFee) / 10000;
}
```

SLSF-14. REDUNDANT ZERO WITHDRAWAL AMOUNT CHECK FOR WRAPPER WITHDRAWALS

SEVERITY: Informational

PATH: NftVault.sol:_withdraw:L253-305

REMEDIATION: see description

STATUS: fixed

DESCRIPTION:

The **_withdraw** function is called only from **withdraw** and **withdrawForPayment** functions.

To optimise for gas usage, the check for a zero withdrawal amount (L275) when **fromWrapper** is false should be moved from the **_withdraw** function to the **withdraw** function:

```
if (amount_ == 0) {
  revert InvalidZeroAmount();
}
```

By performing this check directly in the **withdraw** function, gas can be saved since the redundant check won't be executed during wrapper withdrawals.

```
function _withdraw(
    bool fromWrapper,
    address recipient_,
    address erc20Token_,
    address nftAddress_,
    uint256 nftTokenId_,
    uint256 amount_
) internal nonReentrant {
    ItLock storage itLock = _balances[
        _hash3(nftAddress_, nftTokenId_, erc20Token_)
    ];

    // Deposited tokens in the vault for this nft + token id + erc20 token
    uint256 tokenAmountInVault = itLock.availableAmount();

    if (amount_ > tokenAmountInVault) {
        if (fromWrapper)
            amount_ = tokenAmountInVault; // In case of the withdrawal request for the slash payment, we just withdraw
available amount in the vault
        else revert TooMuchWithdrawals(tokenAmountInVault, amount_); // In case of the normal withdrawal, it is
reverted
    }

    // For the withdrwal of zero amount, it is reverted unless it is requested for the slash payment
    if (amount_ == 0) {
        if (fromWrapper) return;
        revert InvalidZeroAmount();
    }

    itLock.withdraw(amount_);
    _balancePerToken[erc20Token_] -= amount_;

    // If this token does not have amount left per this nft id, remove it from lockedTokens list
    if (itLock.empty())
        _lockedTokens[_hash2(nftAddress_, nftTokenId_)].remove(erc20Token_);

    uint256 feeAmount;
    if (!fromWrapper) {
        feeAmount = withdrawFee(erc20Token_, amount_);
        IERC20(erc20Token_).universalTransfer(_treasury, feeAmount);
        amount_ -= feeAmount;
    }

    IERC20(erc20Token_).universalTransfer(recipient_, amount_);

    emit Withdrawn(
        recipient_,
        erc20Token_,
        nftAddress_,
        nftTokenId_,
        amount_,
        feeAmount
    );
}
```

SLSF-16. GAS OPTIMISATIONS IN LOOPS

SEVERITY: Informational

PATH: BaseVault.sol, WithdrawValidator.sol, NftVault.sol

REMEDIATION: refactor all for loops and implement the recommended changes

STATUS: fixed

DESCRIPTION:

We have identified several loops where gas optimisations in loops are possible:

1. Reading array length at each iteration of the loop takes 6 gas (three for **mload** and three to place **memory_offset**) in the stack. Caching the array length in the stack saves around 3 gas per iteration. We would suggest storing the array's length in a variable before the for-loop.
2. **for** loop's **i++** involves checked arithmetic, which is not required. **i++** in the for loop can never overflow so that increment operation can be moved in an **unchecked** scope (30--40 gas per loop iteration).
3. **++i** costs less gas compared to **i++** or **i += 1** (about 5 gas per iteration).

```
for (; i < nftAddresses_.length; i++) {
    if (_nftWhitelist.contains(nftAddresses_[i]) == flag_)
        continue;
    if (flag_) _nftWhitelist.add(nftAddresses_[i]);
    else _nftWhitelist.remove(nftAddresses_[i]);
}
```

```
for (; i < nftAddresses_.length; i++) {
    if (_nftBlacklist.contains(nftAddresses_[i]) == flag_)
        continue;
    if (flag_) _nftBlacklist.add(nftAddresses_[i]);
    else _nftBlacklist.remove(nftAddresses_[i]);
}
```

```
for (; i < tokens_.length; i++) {
    if (_tokenBlacklist.contains(tokens_[i]) == flag_)
        continue;
    if (flag_) _tokenBlacklist.add(tokens_[i]);
    else _tokenBlacklist.remove(tokens_[i]);
}
```

```
    for (; i < limit_; i++)
        nftAddresses[i] = nftWhitelist.at(i + offset_);
```

```
    for (; i < limit_; i++)
        tokens[i] = tokenBlacklist.at(i + offset_);
```

```
    for (; i < limit_; i++) wrappers[i] = wrapperList.at(i + offset_);
```

```
    for (; i < limit_; i++)
        nftTokenIds[i] = tokenIdBlacklist.at(i + offset_);
```

```
for (; i < batchCount; i++) {
    _deposit(
        erc20Token_,
        nftAddresses_[i],
        nftTokenIds_[i],
        amounts_[i],
        unlockAt_
    );
    sumAmount += amounts_[i];
}
```

```
for (; i < count; i++)
    enableWithdraw(nftAddresses_[i], tokenIds_[i], flag_);
```

SLSF-17. WITHDRAWAL CENTRALISATION RISK

SEVERITY: Informational

PATH: WithdrawValidator.sol:withdrawEnabled, WithdrawValidator.sol:enableWithdraw

REMEDIATION: ensure proper key security and have the owner be multisig wallet where not all keys are controlled by the same entity

STATUS: acknowledged

DESCRIPTION:

The contract WithdrawValidator is used to check for ownership and blacklists on withdrawals from the NFT vault.

When a user invokes the **withdraw()** or **withdrawPayment()** function in **NftVault.sol**, these functions subsequently trigger a call to the **isValid()** function within **WithdrawValidator.sol**. The **isValid()** function assesses whether the **ERC721** token implements the **supportInterface()** function and verifies whether the owner of the **ERC721** token is eligible to withdraw their assets.

Moreover, by utilizing the **enableWithdraw()** function from **WithdrawValidator.sol**, the **owner** gains the ability to deactivate withdrawal functionality for specific users, identified by their **ERC721 address** and **ERC721 id**.

This leads to a centralisation risk, as the owner is able to completely block users from withdrawing their funds.

```
function enableWithdraw(
    address nftAddress_,
    uint256 nftTokenId_,
    bool flag_
) public onlyOwner {
    bytes32 _hash = _hash2(nftAddress_, nftTokenId_);
    if (_withdrawDenyList[_hash] == !flag_)
        revert AlreadyConfigured();
    _withdrawDenyList[_hash] = !flag_;

    emit WithdrawEnabled(nftAddress_, nftTokenId_, flag_);
}
```

```
function withdrawEnabled(
    address nftAddress_,
    uint256 nftTokenId_
) public view returns (bool) {
    return
        _canWithdraw &&
        !_withdrawDenyList[_hash2(nftAddress_, nftTokenId_)];
}
```

SLSF-19. UNUSED EVENT

SEVERITY: Informational

PATH: PaymentWrapper.sol

REMEDIATION: remove any redundant events

STATUS: fixed

DESCRIPTION:

In the PaymentWrapper, the **NewTrustedForwarder** event is never emitted and is therefore redundant.

```
event NewTrustedForwarder(address forwarder, bool flag);
```

SLSF-22. PAYMENTWRAPPER TRUSTED FORWARDER SHOULD BE IMMUTABLE

SEVERITY: Informational

PATH: PaymentWrapper.sol:_trustedForwarder

REMEDIATION: this is a potential risk of error and we would recommend to instead make the _trustedForwarder immutable and set it through a parameter in the constructor

STATUS: fixed

DESCRIPTION:

The PaymentWrapper contract makes use of the Forwarder contract for meta transactions. However, it does not allow for setting the _trustedForwarder variable (like in NftVault), instead it relies on the deployer manually setting the variable to the right value before deployment.

```
function _msgSender() internal view override returns (address sender) {
    if (isTrustedForwarder(msg.sender)) {
        // The assembly code is more direct than the Solidity version using `abi.decode`.
        /// @solidity memory-safe-assembly
        assembly {
            sender := shr(96, calldataload(sub(calldatasize(), 20)))
        }
    } else {
        return super._msgSender();
    }
}
```

```
// put the forwarder address when deploy this contract
address private constant _trustedForwarder = 0x0000000000000000000000000000000000000000;
```

SLSF-23. FUNCTION RENAMING SUGGESTION

SEVERITY: Informational

PATH: WithdrawValidator.sol

REMEDIATION: we recommend slightly changing repeated function names to better reflect their purpose

Examples:

function enableWithdraw(bool flag_) → function globalEnableWithdraw(bool flag_) (L65)

function enableWithdraw(address[] memory nftAddress_, uint256[] memory nftTokenId_, bool flag_) → function batchEnableWithdraw(address[] memory nftAddress_, uint256[] memory nftTokenId_, bool flag_) (L92)

function withdrawEnabled() → function globalWithdrawEnabled() (L105)

STATUS: acknowledged

DESCRIPTION:

In the WithdrawValidator there are multiple overloaded functions with the same name. Functions with the exact same name can hurt readability and decrease overall code quality.

```
function enableWithdraw(bool flag_) external onlyOwner {
      if (_canWithdraw == flag_) revert AlreadyConfigured();
      _canWithdraw = flag_;
      emit WithdrawEnabled(flag_);
   }

   /// @notice Enable / disable withdraw for indivisual nft token
   /// @param nftAddress_ NFT address
   /// @param nftTokenId_ Token id
   /// @param flag_ enable / disable flag
   function enableWithdraw(
      address nftAddress_,
      uint256 nftTokenId_,
      bool flag_
   ) public onlyOwner {
      bytes32 _hash = _hash2(nftAddress_, nftTokenId_);
      if (_withdrawDenyList[_hash] == !flag_)
         revert AlreadyConfigured();
      _withdrawDenyList[_hash] = !flag_;

      emit WithdrawEnabled(nftAddress_, nftTokenId_, flag_);
   }

   /// @notice Batch enable / disable withdraw for indivisual nft token
   /// @param nftAddresses_ NFT addresses
   /// @param tokenIds_ Token id
   /// @param flag_ enable / disable flag
   function enableWithdraw(
      address[] memory nftAddresses_,
      uint256[] memory tokenIds_,
      bool flag_
   ) external onlyOwner {
      uint256 i;
      uint256 count = nftAddresses_.length;
      require(nftAddresses_.length == tokenIds_.length, "Invalid size of array");
      for (; i < count; i++)
         enableWithdraw(nftAddresses_[i], tokenIds_[i], flag_);
   }
```

```
/// @notice Check global withdraw permission
function withdrawEnabled() external view returns (bool) {
    return _canWithdraw;
}

/// @notice Check withdraw permission for the nft token
function withdrawEnabled(
    address nftAddress_,
    uint256 nftTokenId_
) public view returns (bool) {
    return
        _canWithdraw &&
        !_withdrawDenyList[_hash2(nftAddress_, nftTokenId_)];
}
```

SLSF-24. TYPOGRAPHICAL ERRORS

SEVERITY: Informational

PATH: see description

REMEDIATION: refactor Check the user who wants to withdraw has owned this nft token id to Check the user who wants to withdraw owns this nft token id

STATUS: fixed

DESCRIPTION:

NftVault:withdraw (L195) documentation differs from intended functionality.

The sentence: **Check the user who wants to withdraw has owned this nft token id** implies the the message sender should have owned the NFT sometime in the past, however it actually checks if the message sender **currently** owns the NT. Therefore, it should instead be written as **Check the user who wants to withdraw owns this nft token id**.

The same error is also present on line 232.

```
function withdraw(
    address erc20Token_,
    address nftAddress_,
    uint256 nftTokenId_,
    uint256 amount_
) external {
    // Check the user who wants to withdraw has owned this nft token id
    if (
        IERC721(nftAddress_).ownerOf(nftTokenId_) != _msgSender() ||
        !IWithdrawValidator(_withdrawValidator).isValid(
            _msgSender(),
            nftAddress_,
            nftTokenId_
        )
```

SLSF-26. LACK OF CHECK FOR MSG.DATA LENGTH

SEVERITY: Informational

PATH: NftVault.sol, PaymentWrapper.sol

REMEDIATION: see description

STATUS: acknowledged, see commentary

DESCRIPTION:

In the contracts **NftVault.sol** and **PaymentWrapper.sol**, there is a lack of a check to ensure that **msg.data** length is greater than or equal to 20 bytes before proceeding with certain operations in the functions **_msgSender()** and **_msgData()** according to EIP2771. This omission could potentially lead to an underflow and revert.

```
function _msgSender() internal view override returns (address sender) {
    if (isTrustedForwarder(msg.sender)) {
        // The assembly code is more direct than the Solidity version using `abi.decode`.
        /// @solidity memory-safe-assembly
        assembly {
            sender := shr(96, calldataload(sub(calldatasize(), 20)))
        }
    } else {
        return super._msgSender();
    }
}

function _msgData() internal view override returns (bytes calldata) {
    if (isTrustedForwarder(msg.sender)) {
        return msg.data[:msg.data.length - 20];
    } else {
        return super._msgData();
    }
}
```

Add a check to ensure that msg.data.length >= 20.

```
function _msgSender() internal view override returns (address sender) {
    if (isTrustedForwarder(msg.sender) && msg.data.length >= 20) {

    [...]
```

Commentary from the client:

“ - It will revert either way (with overflow error), even without the check.”