SHERLOCK

SHERLOCK SECURITY REVIEW FOR

SHERLOCK

Prepared for:	Nouns
Prepared by:	Sherlock
Lead Security Expert:	WATCHPUG
Dates Audited:	December 1 - December 4, 2022
Prepared on:	December 9, 2022

Introduction

This project solves the problem of allowing payers of token streams to create streams and fund them after the fact, i.e. first create the stream, then fund it later when funds are available.

Scope

- `StreamFactory.sol`
- `Stream.sol`
- `IStream.sol`
- `solady/utils/LibClone.sol`
- `solady/utils/Clone.sol`

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

Issues found

Medium	High
7	0

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

WATCHPUG
KingNFT
rvierdiiev
hansfriese
pashov
cccz
bin2chen
Zarf
DecorativePineapple

joestakey
chainNue
csanuragjain
jonatascm
ctf_sec
adriro
dic0de
dipp
obront
0xZakk
neko_nyaa

Issue M-1: Unnecessary precision loss in `_recipientBalance()`

Source: https://github.com/sherlock-audit/2022-11-nounsdao-judging/issues/70

Found by

WATCHPUG

Summary

Using `ratePerSecond()` to calculate the `_recipientBalance()` incurs an unnecessary precision loss.

Vulnerability Detail

The current formula in `_recipientBalance()` to calculate the vested amount (`balance`) incurs an unnecessary precision loss, as it includes div before mul:

```
balance = elapsedTime_ * (RATE_DECIMALS_MULTIPLIER * tokenAmount_ / duration) /
↪   RATE_DECIMALS_MULTIPLIER
```

This can be avoided and the improved formula can also save some gas.

Impact

Precision loss in `_recipientBalance()`.

Code Snippet

https://github.com/sherlock-audit/2022-11-nounsdao/blob/main/src/Stream.sol#L341-L344

Tool used

Manual Review

Recommendation

Consdier changing to:

```
balance = elapsedTime_ * tokenAmount_ / duration
```

Issue M-2: Lack of sanity check for `stoptime`

Source: https://github.com/sherlock-audit/2022-11-nounsdao-judging/issues/66

Found by

pashov, rvierdiiev, ctf_sec, chainNue, hansfriese, jonatascm, csanuragjain, WATCHPUG

Summary

We believe when the stoptime is before the current time, then it might be wrong.

Vulnerability Detail

Creating a stream with a `stopTime` earlier than the current time is the equivalent of making a simple ERC20 transfer, except that a stream will be much more expensive in terms of gas cost.

Therefore, we believe there is no such use case and the system should include a sanity check to ensure `stoptime>block.timestamp`.

Impact

If the user calls `createAndFundStream()` with a wrong/unintended `stoptime` that is earlier than the current time, the stream will not be able to be canceled.

Code Snippet

https://github.com/sherlock-audit/2022-11-nounsdao/blob/main/src/StreamFactory.sol#L184-L213

Tool used

Manual Review

Recommendation

```
function createStream(
        address payer,
        address recipient,
        uint256 tokenAmount,
        address tokenAddress,
        uint256 startTime,
        uint256 stopTime,
```

```
        uint8 nonce
    ) public returns (address stream) {
        // These input checks are here rather than in Stream because these
↪   parameters are written
        // using clone-with-immutable-args, meaning they are already set when
↪   Stream is created and can't be
        // verified there. The main benefit of this approach is significant gas
↪   savings.
        if (payer == address(0)) revert PayerIsAddressZero();
        if (recipient == address(0)) revert RecipientIsAddressZero();
        if (tokenAmount == 0) revert TokenAmountIsZero();
        if (stopTime <= startTime) revert DurationMustBePositive();
        if (tokenAmount < stopTime - startTime) revert
↪   TokenAmountLessThanDuration();
+       if (stopTime <= block.timestamp) revert StopTimeLessThanCurrentTime();

        stream = streamImplementation.cloneDeterministic(
            encodeData(payer, recipient, tokenAmount, tokenAddress, startTime,
↪   stopTime),
            salt(
                msg.sender, payer, recipient, tokenAmount, tokenAddress,
↪   startTime, stopTime, nonce
            )
        );
        IStream(stream).initialize();

        emit StreamCreated(
            msg.sender, payer, recipient, tokenAmount, tokenAddress, startTime,
↪   stopTime, stream
            );
    }
```

Issue M-3: The rather harsh requirement of `tokenAmount` makes it inapplicable for certain tokens

Source: https://github.com/sherlock-audit/2022-11-nounsdao-judging/issues/63

Found by

WATCHPUG, rvierdiiev

Summary

The requirements for `tokenAmount>=stopTime-startTime` will not be suitable for all tokens and therefore need to be made less applicable for certain tokens like WBTC and EURS.

Vulnerability Detail

Requiring the `tokenAmount>=stopTime-startTime` is suitable for USDC and WETH.

However, such requirements will be a bit too harsh for other popular tokens, eg, WBTC (`decimals:8`) and EURS (`decimals:2`). Therefore, make the system less applicable for those tokens.

For WBTC, it must be `0.31536WBTC` per year (worth about $5,400) to meet this requirement, and for EURS, it must be at least `315,360EURS` per year (worth about $315,000).

Impact

The system will be inapplicable for certain tokens with higher per wei value, eg, WBTC and EURS.

Code Snippet

https://github.com/sherlock-audit/2022-11-nounsdao/blob/main/src/StreamFactory.sol#L200

Tool used

Manual Review

Recommendation

Consider changing to `tokenAmount*RATE_DECIMALS_MULTIPLIER>=stopTime-startTime`.

Issue M-4: Two address tokens can be withdrawn by the payer even when the stream has began

Source: https://github.com/sherlock-audit/2022-11-nounsdao-judging/issues/52

Found by

hansfriese, pashov, rvierdiiev, DecorativePineapple

Summary

It has been identified that if a stream has begun with a two address token the payer can withdraw the full amount via the `rescueERC20` function.

Vulnerability Detail

Two address tokens exists in the blockchain. For example, Synthetix's ProxyERC20 contract is such a token which exists in many forms (sUSD, sBTC...). A stream can be created with such tokens, but the payer can withdraw the full amount via the `rescueERC20` function. The only check in the `rescueERC20` function is that `tokenAddress==address(token()`, which is irrelevant in the case of two address tokens.

Impact

This can make the payer be able to withdraw the funds that are deposited in the stream and break the system, because the balance of the stream contract is zero and the `recipient()` cannot withdraw the fair amount based on the rate and the time elapsed.

Code Snippet

The payer can withdraw the full amount of token via the `rescueERC20` function:

```
function rescueERC20(address tokenAddress, uint256 amount) external onlyPayer {
    if (tokenAddress == address(token())) revert CannotRescueStreamToken();

    IERC20(tokenAddress).safeTransfer(msg.sender, amount);
}
```

Tool used

Manual Review

Recommendation

Replace the address check with a balance check - record the balance of the token that's deposited in the stream before and after the transfer and assert that they are equal.

Issue M-5: The `Stream` contract is designed to receive ETH but not implement function for withdrawal

Source: https://github.com/sherlock-audit/2022-11-nounsdao-judging/issues/47

Found by

KingNFT

Summary

The `Stream` contract instances can receive ETH but can not withdraw, ETH occasionally sent by users will be stuck in those contracts.

Vulnerability Detail

Shown as the test case, it can receive ETH normally.

```
contract StreamReceiveETHTest is StreamTest {
    function setUp() public override {
        super.setUp();
    }

    function test_receiveETH() public {
        s = Stream(
            factory.createStream(
                payer, recipient, STREAM_AMOUNT, address(token), startTime,
↪   stopTime
            )
        );

        vm.deal(payer, 10 ether);
        vm.prank(payer);
        (bool success, ) = address(s).call{value: 1 ether}("");
        assertEq(success, true);
        assertEq(address(s).balance, 1 ether);
    }
}
```

Result

```
Running 1 test for test/Stream.t.sol:StreamReceiveETHTest
[PASS] test_receiveETH() (gas: 167691)
Test result: ok. 1 passed; 0 failed; finished in 1.25ms
```

Impact

See Summary

Code Snippet

https://github.com/Vectorized/solady/blob/db4857b4a1e17ad035668b588b41a1c90139b99d/src/utils/LibClone.sol#L193-L204

Tool used

Manual Review

Recommendation

Add a `rescueETH()` function which is similar with the existing `rescueERC20()`

Issue M-6: If the recipient is added to the USDC blacklist, then cancel() does not work

Source: https://github.com/sherlock-audit/2022-11-nounsdao-judging/issues/37

Found by

Zarf, cccz, bin2chen, joestakey

Summary

cancel() will send the vested USDC to the recipient, if the recipient is added to the USDC blacklist, then cancel() will not work

Vulnerability Detail

When cancel() is called, it sends the vested USDC to the recipient and cancels future payments. Consider a scenario where if the payer intends to call cancel() to cancel the payment stream, a malicious recipient can block the address from receiving USDC by adding it to the USDC blacklist (e.g. by doing something malicious with that address, etc.), which prevents the payer from canceling the payment stream and withdrawing future payments

```
function cancel() external onlyPayerOrRecipient {
    address payer_ = payer();
    address recipient_ = recipient();
    IERC20 token_ = token();

    uint256 recipientBalance = balanceOf(recipient_);

    // This zeroing is important because without it, it's possible for recipient
↪   to obtain additional funds
    // from this contract if anyone (e.g. payer) sends it tokens after
↪   cancellation.
    // Thanks to this state update, `balanceOf(recipient_)` will only return zero
↪   in future calls.
    remainingBalance = 0;

    if (recipientBalance > 0) token_.safeTransfer(recipient_, recipientBalance);
```

Impact

A malicious recipient may prevent the payer from canceling the payment stream and withdrawing future payments

Code Snippet

https://github.com/sherlock-audit/2022-11-nounsdao/blob/main/src/Stream.sol#L237-L249

Tool used

Manual Review

Recommendation

Instead of sending tokens directly to the payer or recipient in cancel(), consider storing the number of tokens in variables and having the payer or recipient claim it later

Issue M-7: Payer cannot withdraw accidental extra funds sent to the contract without canceling

Source: https://github.com/sherlock-audit/2022-11-nounsdao-judging/issues/28

Found by

0xZakk, rvierdiiev, obront, adriro, dic0de, neko_nyaa, WATCHPUG, cccz, dipp

Summary

If a different ERC20 is accidentally sent to the contract, the Payer can withdraw it using the `rescueERC20` function. However, if they accidentally send extra of the streaming token's funds to the contract, the only way to withdraw it is to cancel the stream.

Vulnerability Detail

The Nouns team seems to have made the decision that they should protect against accidental funds being sent into the contract. They implemented the `rescueERC20` function to accomplish this.

However, the `rescueERC20` function only works for non-stream tokens. If they accidentally send too much of the streaming token (which seems like a likely scenario), there is no similar rescue function to retrieve it.

Instead, their only option is to cancel the stream. In a protocol that's intended to be run via a governance system, canceling the stream could cause problems for the receiver (for example, if they are unable to pass a vote to restart the stream).

Impact

If too many stream tokens are sent into the contract, the whole stream will need to be canceled to retrieve them.

Code Snippet

https://github.com/sherlock-audit/2022-11-nounsdao/blob/main/src/Stream.sol#L237-L259

Tool used

Manual Review

Recommendation

Adjust the rescueERC20 function to also allow for withdrawing excess stream tokens, as follows:

```
function rescueERC20(address tokenAddress, uint256 amount) external onlyPayer {
-     if (tokenAddress == address(token())) revert CannotRescueStreamToken();
+     if (tokenAddress == address(token()) && amount < tokenBalance() -
 ↪  remainingBalance) revert AmountExceedsBalance;

    IERC20(tokenAddress).safeTransfer(msg.sender, amount);
}
```