Sablier Finance

Security Review

Cantina Managed review by:
Zach Obront, Security Researcher
RustyRabbit, Security Researcher

December 15, 2023

Contents

1 Introduction **2**
1.1 About Cantina . . . 2
1.2 Disclaimer . . . 2
1.3 Risk assessment . . . 2
1.3.1 Severity Classification . . . 2

2 Security Review Summary **3**

3 Findings **4**
3.1 Critical Risk . . . 4
3.1.1 Plugins can be maliciously overridden by colliding signatures . . . 4
3.2 High Risk . . . 6
3.2.1 Malicious asset can be used to inject Javascript into Sablier front end . . . 6
3.2.2 Wrapped native asset address can be freely chosen and is assumed to return equal amount of tokens on deposit . . . 7
3.2.3 Any plugins using permit2 can steal funds with frontrunning attack . . . 8
3.2.4 Lack of reasonable value check in setMinGasReserve() can lead to bricked proxy . . . 9
3.2.5 Any envoy with access to SablierV2ProxyTarget can steal all funds in proxy . . . 10
3.2.6 Owner can be temporarily changed within proxy calls, allowing complete control of proxy . . . 11
3.3 Medium Risk . . . 13
3.3.1 Create2 salt based on tx.origin alone . . . 13
3.3.2 Token allowances stay in effect on proxy ownership transfer . . . 14
3.3.3 Time dependent data in NFT metadata . . . 14
3.3.4 Plugin or target with selfdestruct . . . 15
3.3.5 Permission and plugins not reset on proxy ownership transfer . . . 15
3.3.6 Plugins and permissions storage variables are unprotected . . . 16
3.3.7 Deployment front running protection not effective . . . 17
3.3.8 `refundableAmountOf` will return a value when `isCancelable` is false . . . 18
3.4 Low Risk . . . 19
3.4.1 Permissions and plugins stored as mappings . . . 19
3.4.2 No protection against non plugin or non target contracts . . . 19
3.4.3 Infinite approval for Sablier lockup contracts . . . 19
3.4.4 Proxies cannot be burned, and thus must be sent to potentially unsafe addresses . . . 20
3.4.5 `batchCreate` functions can overflow total amount and revert . . . 21
3.4.6 Receiver using proxy cannot cancel individual stream . . . 22
3.5 Informational . . . 22
3.5.1 Atypical transferOwnership() function on PRBProxyRegistry . . . 22
3.5.2 No default function for transferring ETH or tokens . . . 23
3.5.3 No mirror function for `burn()` . . . 23
3.5.4 Malicious asset contract can force Sablier's `tokenURI()` function to revert if called on chain . . . 23

4 Additional Comments **25**
4.1 On-chain contracts . . . 25

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Directly* exploitable security vulnerabilities that need to be fixed.
High	Security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All high issues should be addressed.
Medium	Objective in nature but are not security vulnerabilities. Should be addressed unless there is a clear reason not to.
Low	Subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. When determining the severity one first needs to determine whether the finding is subjective or objective. All subjective findings are considered of Minor severity.

Next it is determined whether the finding can be regarded as a security vulnerability. Some findings might be objective improvements that need to be fixed, but do not impact the project's security overall (Medium).

Finally, objective findings of security vulnerabilities are classified as either critical or major. Critical findings should be directly vulnerable and have a high likelihood of being exploited. Major findings on the other hand may require specific conditions that need to be met before the vulnerability becomes exploitable.

2 Security Review Summary

Sablier is a token streaming protocol available on Ethereum, Optimism, Arbitrum, Polygon, Ronin, Avalanche, and BSC. It's the first of its kind to have ever been built in crypto, tracing its origins back to 2019. Similar to how you can stream a movie on Netflix or a song on Spotify, so you can stream tokens by the second on Sablier.

From May 31st to June 9th the Cantina team conducted a review of v2-core on commit hash 07014ac, v2-periphery on commit hash cc9434 and prb-proxy on commit hash 6fbb6d.

The team identified a total of **25** issues in the following risk categories:

- Critical Risk: 1
- High Risk: 6
- Medium Risk: 8
- Low Risk: 6
- Gas Optimizations: 0
- Informational: 4

The post review commit hashes for the final version tags (v1.0.0, v1.0.0, and v4.0.0) are:

- v2-core on commit 412ec3d
- v2-periphery on commit 0c389e7
- prb-proxy on commit 27594d5

3 Findings

3.1 Critical Risk

3.1.1 Plugins can be maliciously overridden by colliding signatures

Severity: Critical Risk

Context:

- PRBProxyAnnex.sol#L44-L64

Description: Plugins are saved as a mapping from a 4byte selector to a given plugin address:

```
mapping(bytes4 method => IPRBProxyPlugin plugin) public plugins;
```

When new plugins are installed, we call `methodList()` on the plugin, and then iterate through the list of selectors, and save the plugin address for each selector:

```
function installPlugin(IPRBProxyPlugin plugin) external override {
    // Get the method list to install.
    bytes4[] memory methodList = plugin.methodList();

    // The plugin must have at least one listed method.
    uint256 length = methodList.length;
    if (length == 0) {
        revert PRBProxy_NoPluginMethods(plugin);
    }

    // Enable every method in the list.
    for (uint256 i = 0; i < length;) {
        plugins[methodList[i]] = plugin;
        unchecked {
            i += 1;
        }
    }

    // Log the plugin installation.
    emit InstallPlugin(plugin);
}
```

As a result, an innocent looking plugin can be crafted to intentionally override an existing plugin. When this happens, it will replace the existing plugin as the place where control flow is sent when this plugin in called.

This is extremely dangerous, as it allows an attacker to skirt around the plugin's protections logic and get complete control over the proxy mid execution.

Proof of Concept:

There are many ways this could cause harm in various protocols using PRBProxy, but the simplest is to look at the Sablier integration.

When a stream is cancelled by a receiver, Sablier sends the refund to the `sender` and then calls `onStreamCanceled()` on the `sender` contract. The Sablier defined plugin is used to forward these funds along to the owner of the proxy.

However, a malicious plugin could be installed with a colliding 4byte selector that, instead, sends the funds to an attacker. Even worse, once the attacker has control flow on behalf of the plugin, they would be able to cancel all other active streams and steal the refundable amount of all of them, which could be devastating.

Here is a test that can be dropped into `periphery/test/integration/plugin/on-stream-canceled` that emulates installing an innocent plugin to collect fees from an unrelated protocol, but results in the refund being sent to an attacker when a stream is canceled.

As you will see, the `test_PluginOverride` function emulates the exact behavior of `test_OnStreamCanceled` in your own integration tests, but because the malicious plugin is installed first, an attacker is able to steal the funds.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity >=0.8.19 <0.9.0;
```

```
import { LockupLinear } from "@sablier/v2-core/types/DataTypes.sol";
import { ISablierV2Lockup } from "@sablier/v2-core/interfaces/ISablierV2Lockup.sol";
import { ISablierV2ProxyPlugin } from "src/interfaces/ISablierV2ProxyPlugin.sol";
import { IPRBProxyPlugin } from "@prb/proxy/interfaces/IPRBProxyPlugin.sol";
import { Errors } from "src/libraries/Errors.sol";
import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol";

import { Integration_Test } from "../../Integration.t.sol";

interface FeeController {
    function getAsset(uint) external view returns (IERC20);
}

contract InnocentLookingPlugin {
    address constant TREASURY = address(420);

    function methodList() external pure returns (bytes4[] memory methods) {
        methods = new bytes4[](1);
        methods[0] = this.onAddictionFeesRefunded.selector; // same as onStreamCanceled
    }

    function onAddictionFeesRefunded(uint248 loanId, int168, uint192 feeAmount, int248) public {
        // Get the asset of the loan.
        IERC20 asset = FeeController(msg.sender).getAsset(loanId);

        // Send fees to the treasury.
        asset.transfer({ to: TREASURY, amount: feeAmount });
    }
}

contract PluginOverrideTest is Integration_Test {
    uint256 internal streamId;

    function setUp() public virtual override {
        Integration_Test.setUp();
        installPlugin();
        streamId = createWithRange();

        // Lists the lockupLinear contract in the archive.
        changePrank({ msgSender: users.admin.addr });
        archive.list(address(lockupLinear));
        changePrank({ msgSender: users.alice.addr });
    }

    function test_PluginOverride() external {
        address ATTACKER = address(420);

        // install an innocent looking plugin, but that overrides the onStreamCanceled selector
        IPRBProxyPlugin innocentLookingPlugin = IPRBProxyPlugin(address(new InnocentLookingPlugin()));
        bytes memory data = abi.encodeCall(proxyAnnex.installPlugin, (innocentLookingPlugin));
        proxy.execute(address(proxyAnnex), data);

        // Retrieve the initial asset balances of the proxy owner and the attacker.
        uint256 aliceInitBalance = asset.balanceOf(users.alice.addr);
        uint256 attackerInitBalance = asset.balanceOf(ATTACKER);

        // Simulate the passage of time.
        vm.warp(defaults.CLIFF_TIME());

        // Make the recipient the caller so that Sablier calls the hook implemented by the plugin.
        changePrank({ msgSender: users.recipient.addr });

        // // Asset flow: Sablier contract  proxy  ATTACKER!
        // // Expect transfers from the Sablier contract to the proxy, and then from the proxy to the ATTACKER!
        expectCallToTransfer({ to: address(proxy), amount: defaults.REFUND_AMOUNT() });
        expectCallToTransfer({ to: ATTACKER, amount: defaults.REFUND_AMOUNT() });

        // Cancel the stream and trigger the plugin.
        lockupLinear.cancel(streamId);

        // Assert that Alice received no funds from the cancelation.
        uint256 aliceFinalBalance = asset.balanceOf(users.alice.addr);
        assertEq(aliceFinalBalance, aliceInitBalance, "balances do not match");

        // Assert that the ATTACKER got the refund amount.
```

```
        uint256 attackerFinalBalance = asset.balanceOf(ATTACKER);
        uint256 attackerExpectedBalance = attackerInitBalance + defaults.REFUND_AMOUNT();
        assertEq(attackerExpectedBalance, attackerFinalBalance, "balances do not match");

    }
}
```

Recommendation: When plugins are installed, ensure that the 4byte selectors do not collide with any existing plugins.

```
function installPlugin(IPRBProxyPlugin plugin) external override {
    // Get the method list to install.
    bytes4[] memory methodList = plugin.methodList();

    // The plugin must have at least one listed method.
    uint256 length = methodList.length;
    if (length == 0) {
        revert PRBProxy_NoPluginMethods(plugin);
    }

    // Enable every method in the list.
    for (uint256 i = 0; i < length;) {
+       if (plugins[methodList[i]] != address(0)) revert PRBProxy_SelectorCollision(methodList[i]);
        plugins[methodList[i]] = plugin;
        unchecked {
            i += 1;
        }
    }

    // Log the plugin installation.
    emit InstallPlugin(plugin);
}
```

Sablier: Changes applied in PR 121.

Cantina: Confirmed that the current storage layout (pulling methods from storage rather than from methodList() call solves this problem, so this fix is solid.

3.2 High Risk

3.2.1 Malicious asset can be used to inject Javascript into Sablier front end

Severity: High Risk

Context:

- SablierV2NFTDescriptor.sol#L316-L325
- SablierV2NFTDescriptor.sol#L63-L79
- NFTSVG.sol#L120-L144

Description: The NFT Descriptor contract fetches the `symbol` from the asset contract using the `safeAssetSymbol()` function. As long as that symbol has a length of over 64 bytes (ie it is a string), it is returned as is.

```
function safeAssetSymbol(address asset) internal view returns (string memory) {
    (bool success, bytes memory symbol) = asset.staticcall(abi.encodeCall(IERC20Metadata.symbol, ()));

    // Non-empty strings have a length greater than 64, and bytes32 has length 32.
    if (!success || symbol.length <= 64) {
        return "ERC20";
    }

    return abi.decode(symbol, (string));
}
```

This `symbol` is passed along to the SVG, where it is inserted directly into the SVG text that is returned to the front end to display.

```
function generateFloatingText(
    string memory sablierAddress,
    string memory streamingModel,
    string memory assetAddress,
    string memory assetSymbol
)
    internal
    pure
    returns (string memory)
{
    return string.concat(
        '<text text-rendering="optimizeSpeed">',
        SVGElements.floatingText({
            offset: "-100%",
            text: string.concat(sablierAddress, unicode"  ", "Sablier V2 ", streamingModel)
        }),
        SVGElements.floatingText({
            offset: "0%",
            text: string.concat(sablierAddress, unicode"  ", "Sablier V2 ", streamingModel)
        }),
        SVGElements.floatingText({ offset: "-50%", text: string.concat(assetAddress, unicode"  ", assetSymbol)
↪   }),
        SVGElements.floatingText({ offset: "50%", text: string.concat(assetAddress, unicode"  ", assetSymbol)
↪   }),
        "</text>"
    );
}
```

There are no checks on the value of this `symbol`, and it is passed directly to any front end displaying Sablier NFTs to display.

However, SVGs can be used to execute arbitrary Javascript (see: Using-JavaScript-in-SVG and svg-xss-injection-attacks). This means that if an attacker were to create an asset with a symbol that contained malicious Javascript, they could use this to inject Javascript into the Sablier front end. This could be used, for example, to create a fake Metamask pop up that asks a user to sign a message.

Recommendation: Symbol strings are not typically longer than a few characters. Sensible upper limits should be placed on the maximum length of a string that can be returned from the `safeAssetSymbol()` function, which will prevent an attacker from including code with any complexity.

To be extra safe, it may be worth adding an input sanitation check to the `safeAssetSymbol()` function so that only Aa-Zz, 0-9 is allowed (preventing < / > ' ' ;).

Sablier: Changes applied in PR 555.

Cantina: Fixed.

3.2.2 Wrapped native asset address can be freely chosen and is assumed to return equal amount of tokens on deposit

Severity: High Risk

Context:

- /v2-periphery/src/SablierV2ProxyTarget.sol#L329-L338

Description: The `wrapAndCreate` functions in the `SablierV2ProxyTarget` contract allow the user to specify the address of the wrapped native asset (e.g. WETH).

```
    function wrapAndCreateWithDurations(
        ISablierV2LockupLinear lockupLinear,
        LockupLinear.CreateWithDurations memory createParams
    )
        external
        payable
        override
        onlyDelegateCall
        returns (uint256 streamId)
    {
        createParams.totalAmount = uint128(msg.value);
        // Wrap the native asset payment in ERC-20 form.
        IWrappedNativeAsset(address(createParams.asset)).deposit{ value: msg.value }();
        // Approve the Sablier contract to spend funds.
        _approve(address(lockupLinear), createParams.asset, createParams.totalAmount);
        // Create the stream.
        streamId = lockupLinear.createWithDurations(createParams);
    }
```

The ETH sent with the transaction is deposited in the contract via the `deposit()` function. The amount of ETH deposited is `msg.value` which is used to overwrite the `createParams.totalAmount` used to create the stream.

However, as the WETH address is specified by the user they can choose any other ERC20 contract that can handle a `deposit()`. If that contract returns less tokens than the ETH deposited this can be used to steal any of those tokens from the proxy.

The attack could look like this:

- The returned amount of ERC20 token is less than the `msg.value`.
- The `_approve()` then does an infinite approval on the token contract.
- The totalAmount of the stream is set to the `msg.value`.
- The difference will be taken from the proxy contract's balance when creating the stream.

We haven't found a contract that meets these requirements (although stETH comes close except for the empty calldata check in the fallback function)

Recommendation: Limit the acceptable wrapped native asset contracts with an allow list or an immutable variable in case only one contract is to be allowed.

Sablier: Changes applied in commit b7c426.

Cantina: Fixed.

3.2.3 Any plugins using permit2 can steal funds with frontrunning attack

Severity: High Risk

Context:

- SablierV2ProxyTarget.sol#L662

Description: As stated in the Uniswap Permit2 documentation here and here, the calling contract should check the intent of the signer on how the signature is used.

In `SablierV2ProxyTarget`, any time funds are transferred the permit2 signature is used to permit the proxy to spend the tokens on behalf of the user.

```
permit2Params.permit2.permit({
    owner: owner_,
    permitSingle: permit2Params.permitSingle,
    signature: permit2Params.signature
});

// Transfer funds from the proxy owner to the proxy.
permit2Params.permit2.transferFrom({ from: owner_, to: address(this), amount: amount, token: address(asset) });
```

This signature however only specifies the proxy address, token contract and token amount; not the target/plugin contract or the function being called, which more precisely covers the intent of the owner when they sign the permit2 allowance.

```
struct PermitSingle {
    // the permit data for a single token allowance
    PermitDetails details;
    // address permissioned on the allowed tokens
    address spender;
    // deadline on the permit signature
    uint256 sigDeadline;
}

struct PermitDetails {
    // ERC20 token address
    address token;
    // the maximum amount allowed to spend
    uint160 amount;
    // timestamp at which a spender's token allowances become invalid
    uint48 expiration;
    // an incrementing value indexed per owner, token, and spender for each signature
    uint48 nonce;
}
```

This means the proxy is able to use the signature to transfer funds from the permit2 contract for any reason.

In the event that there is a plugin installed on the proxy that uses permit2, a user can front run the proxy's use of the signature and use the same signature to spend the funds for it's own purposes.

Recommendation: 1) Documentation should be clear that `permit2` signatures are not allowed in plugins, as it would be impossible for the proxy code to enforce verification of the witness before forwarding to the plugin contract.

2) Although less likely, this same risk can exist in other situations besides plugins. For example, envoys may be permitted to send transactions to a contract with `permit2`, assuming they would be able to use it without the owner's signature.

To solve this, use the more restrictive permitWitnessTransferFrom which allows limiting the validity of the signature to the action intended by the owner. This should at minimum include the target contract address and the function called on the target contract, but ideally would also include the target contract's function parameters.

Note that this witness data should be validated by the proxy code itself before forwarding the call to the target contract. Therefore a dedicated function `executeWithPermit2(address target, bytes calldata data, Permit2Params calldata permit2Params)` for permit2 usage is needed to uniformly be able to extract the permit2 signature from the calldata.

Sablier: Acknowledged. We will accept this issue as an inescapable consequence of the computational universality of the plugin/ target execution flow. But we will make sure to document the dangers of using Permit2 signatures in proxy plugins, as you have recommended.

Cantina: Acknowledged.

3.2.4 Lack of reasonable value check in setMinGasReserve() can lead to bricked proxy

Severity: High Risk

Context:

- /prb-proxy/src/PRBProxyAnnex.sol#L72
- /prb-proxy/src/PRBProxy.sol#L66-L77
- /prb-proxy/src/PRBProxy.sol#L106-L118

Description: There is no sanity check on the input parameter for the new `minGasReserve` in `setMinGasReserve()`. The `minGasReserve` is deducted from the gas stipend to be sent in the `delegatecall` to a target or plugin contract.

```
    function _safeDelegateCall(address to, bytes memory data) internal returns (bool success, bytes memory
↪   response) {
        ...
        uint256 stipend = gasleft() - minGasReserve;
        (success, response) = to.delegatecall{ gas: stipend }(data);
        ...
    }
```

If the `minGasReserve` is set to a high enough value the total required gas amount could be so high it would exceed the block size limit and never be able to get included in a block. This also applies to a transaction to set the `minGasReserve` back to a sane value resulting in an unusable proxy locking all the funds with it.

The storage slot used for the `minGasReserve` can also accidentally (or intentionally by a malicious plugin or target contract) be overwritten with the same result.

Additionally reserving a fixed amount of gas for post-processing the `delegatecall` serves no purpose. If we would rely on the normal 1/64th amount of gas reserved and it would not be sufficient for post-processing (which is limited to checking the success of the `delegatecall` and possibly reverting with the underlying reason) the transaction would revert due to insufficient gas. This behavior however is no different than when the `minGasReserve` is applied and the stipend for the `delegatecall` is reduced in which case the `delegatecall` would revert due to insufficient gas. In both situations the transaction reverts and the user needs to send the transaction again with an increased gas limit.

Recommendation: Consider removing the stipend altogether. Alternatively, perform a check for the sane value of `minGasReserve` when setting a new value and store it in an external contract to avoid plugins or target contracts accidentally overwriting the storage slot.

Sablier: Changed applied in PR 114.

Cantina: Fixed.

3.2.5 Any envoy with access to SablierV2ProxyTarget can steal all funds in proxy

Severity: High Risk

Context:

- SablierV2ProxyTarget.sol#L643-L666

Description: Envoys are given permissions on a target-by-target basis, with the assumption that they are not fully trusted with the assets of the proxy.

When a proxy owner gives an envoy permissions on the `SablierV2ProxyTarget`, it is safe for them to assume that the user cannot steal their funds. This is because the only way to create a stream is to have a signed `permit2` order from the owner, which allows the assets to be transferred to the proxy so the stream can be created.

```
function _transferAndApprove(
    address sablierContract,
    IERC20 asset,
    uint160 amount,
    Permit2Params calldata permit2Params
)
    internal
{
    // Retrieve the proxy owner.
    address owner_ = owner;

    // Permit the proxy to spend funds from the proxy owner.
    permit2Params.permit2.permit({
        owner: owner_,
        permitSingle: permit2Params.permitSingle,
        signature: permit2Params.signature
    });

    // Transfer funds from the proxy owner to the proxy.
    permit2Params.permit2.transferFrom({ from: owner_, to: address(this), amount: amount, token:
↪  address(asset) });

    // Approve the Sablier contract to spend funds.
    _approve(sablierContract, asset, amount);
}
```

However, this `_transferAndApprove()` function allows the caller to input the `Permit2Params`, which includes the `permit2` address that is used for the approval and transfer.

This allows a malicious envoy to deploy their own dummy contract (with `permit()` and `transferFrom()` functions that pass but don't do anything), and then create a stream on behalf of the owner with this contract passed as the `permit2` address.

The result is that the `createStream()` function will call `_transferAndApprove()`, no checks will be performed in `permit2`, the asset will be approved for spending by Sablier, and the stream will be created.

This stream could be uncancellable and send assets to themselves over a short time period, stealing all the funds in the proxy from the owner.

Recommendation: A list of approved `permit2` contracts should be maintained on the contract, and the inputted `permit2` address should be checked against this list before approving the transfer.

Sablier: Addressed via PR 102.

Cantina: Fixed.

3.2.6 Owner can be temporarily changed within proxy calls, allowing complete control of proxy

Severity: High Risk

Context:

- PRBProxy.sol#L137-L151

Description: When a proxy owner uses the `execute()` function to perform a delegate call, the `_safeDelegateCall()` function is used:

```
function _safeDelegateCall(address to, bytes memory data) internal returns (bool success, bytes memory
↪  response) {
    // Save the owner address in memory so that this variable cannot be modified during the DELEGATECALL.
    address owner_ = owner;

    // Reserve some gas to ensure that the contract call will not run out of gas.
    uint256 stipend = gasleft() - minGasReserve;

    // Delegate call to the provided contract.
    (success, response) = to.delegatecall{ gas: stipend }(data);

    // Check that the owner has not been changed.
    if (owner_ != owner) {
        revert PRBProxy_OwnerChanged(owner_, owner);
    }
}
```

Because the contract that receives the delegate call has the ability to manipulate the storage slots of the proxy, a check is performed at the end to ensure that the owner hasn't changed over the course of the call.

However, this check does not ensure that the owner does not change during the call, which could be just as dangerous, allowing a malicious user to temporarily become the owner of the contract for the duration of the call.

Proof of Concept:

This vulnerability is very dangerous in almost any use of the proxy, as it allows an attacker to gain control of the proxy mid execution.

As a simple example, let's look at how this could be used to steal funds from a Sablier user.

1) An owner calls out to a contract (let's call it `Contract A`) using the `execute()` function.
2) The target contract updates storage slot 0 (`owner`) to the address of another owned contract (let's call it `Contract B`).
3) The target contract then calls (not delegate calls) `Contract B`, passing control flow over.
4) `Contract B` uses the `execute()` function on the proxy (which it is enabled to do because it is the owner) to cancel all the proxies streams.
5) `SablierV2ProxyTarget`'s `cancel()` function sends the refunded amount back to the owner of the proxy (`Contract B`).
6) When control flow returns to `Contract A`, it rewrites the storage slot to transfer ownership of the proxy back to the original owner.
7) When control flow returns to the proxy, the `owner_ == owner` check passes, and the function is concluded successfully.

Recommendation: There is no easy fix to this problem, and it likely requires some rethinking of the proxy storage architecture.

Our opinion is that the best solution to this problem (along with many of the other issues reported) is to make proxies non-transferrable. This would allow you to set the owner as an immutable value, which removes the risk of the owner ever being changed.

(Note that this would require target contracts to call an `owner()` function on the proxy to get the owner address, rather than reading it directly from storage.)

In the event that proxies must remain transferrable, the PRBProxyRegistry contract should be maintained as the final authority on ownership. Each proxy should have the Registry address saved as an immutable value, and should query the Registry for ownership checks when `execute()` is called.

Sablier: Changes applied for the 1st part in PR 119 and for the 2nd part in PR 104.

Cantina: Fixed.

3.3 Medium Risk

3.3.1 Create2 salt based on tx.origin alone

Severity: Medium Risk

Context:

- /prb-proxy/src/PRBProxyRegistry.sol#L134
- /prb-proxy/src/PRBProxyRegistry.sol#L68

Description: When deploying a proxy from the `PRBProxyRegistry` contract the `tx.origin` is used together with an incremental `nonce` (tracked per `tx.origin`) to form the salt for the CREATE2 when deploying the `PRBProxy` contract.

However, this means the proxy address is determined solely by the gas payer of the transaction (and its relative position in a sequence of deploying proxies). It does not depend on the state of the proxy created in any way, not even the owner. The assumption is that in most cases the owner is the `tx.origin` and in these cases, no one can front-run the transaction to hijack the proxy address. The Auditor's Handbook for PRBProxy mentions the following about Account Abstraction:

> We are aware of a potential "feature clash" with Account Abstraction (if and when it gets implemented in Ethereum).

This seems to refer to the Account Abstraction efforts that require a change in the Ethereum protocol (like EIP2938 and EIP3074).

However, there are other situations where the choice of using (only) `tx.origin` heavily impacts the deployment process:

- EIP-2771 AA using meta-transactions
- EIP-4337 AA using UserOperations
- Multisigs where the last signer is the `tx.origin` and as such the predicted proxy address may or may not be correct.

Some major problems present themselves in a relayer/gasMaster type scenario.

- Scenario 1 Imagine when the relayer is asked to predict a proxy address for a user (Alice). The relayer uses `nonce` = 1 to predict the proxy address and request Alice to sign some type of meta-transaction (e.g. for refunding gas in a gasless transaction, but this could include an interaction with a third party contract). Then Alice delays her signature and a second user (Bob) asks the relayer to predict their proxy (using `nonce` = 2). The relayer does so and asks Bob to sign a second meta-transaction. Now Bob may be assured the predicted address is his based on the assumption the relayer will send them out in the order they were predicted. However, if the sequence of Alice's and Bob's signatures are sent out in reverse order they end up with each other's predicted proxy address as the `SablierV2ProxyTarget` uses the nonce in the order they are included in the block. This is obviously an implementation issue of the relayer but it points to the design flaw of not including the owner in the salt of the CREATE2.
- Scenario 2 Imagine the same scenario but Alice never signs the transactions and the relayer is correctly implemented. The relayer is now in a state where they can't proceed because there is no way to deploy Bob's proxy without first deploying Alice's proxy. They can't honestly deploy Alice's without reputation damage as they would have to deploy her proxy designating themselves or a dead address as owner which looks like they maliciously provided Alice with the wrong address.
- Scenario 3 Imagine Bob having requested the deployment of a proxy to a relayer. Firstly Bob has to trust the relayer to present the correct address (there is no way Bob can validate the given address). Bob then may sign a message to the relayer to initiate the gasless deployment.

The malicious relayer however decides to use the `deplyAndExecuteFor()` to deploy the proxy but in an altered state (e.g. plugins installed, envoy permission sets, ERC20/NFT token allowances set, basically anything they want relating to interactions with third party contracts). The address of this proxy would be identical to one deployed using the simple `deployFor()` function and hence Bob will accept this address without realizing the altered state of the proxy.

Note that this issue is similar to the issues surrounding `transferOwnership()`, but even if they are resolved the `deplyAndExecuteFor()` would still allow a relayer to maliciously deploy a proxy on behalf of a user.

Recommendation: Instead of using only `tx.origin` as the basis for the salt and nonce tracking it is better to use the `owner` and in case `deployandExecuteFor()` is used combine the `owner` with the target contract address and the `data` executed on the target contract.

Additionally, provide a view function to allow a user to predict and validate the address based on these same parameters. In this way a user can be assured they receive a proxy in the state they expect it to be and it doesn't matter who or what the `tx.origin` or the `msg.sender` is.

Note that this also solves the "potential feature clash with Account Abstraction" described in the auditor's handbook and the blocked relayer scenario described in scenario 3.

Sablier: Changes applied in PR 123.

Fixed: Fixed.

3.3.2 Token allowances stay in effect on proxy ownership transfer

Severity: Medium Risk

Context:

- /prb-proxy/src/PRBProxy.sol#L121-L129

Description: The proxy contract acts as a smart contract wallet for it's owner and it is a reasonable scenario that the proxy will at some point have (infinite) approvals set for other contracts on some (ERC20) tokens.

When the ownership of a proxy for some reason is transferred to another entity (person or contracts as part of a protocol) the existing approvals stay in effect as they are based on the proxy's address.

In such a scenario the new owner (if a person) is unaware of the existing approvals and this would allow the old owner to use those approvals to potentially steal the tokens via the approved contracts.

Note that this is also true for approval on the permit2 contract although they have an expiration and in the case of `AllowanceTransfer` are reset on each new approval.

Recommendation: Reconsider the proxy ownership transfer use case as many external contracts depend on the address as an indication of ownership and access control.

Sablier: Fixed with the inclusion of 07fed42 resetting all the `constructorParams` in `_deploy`.

Cantina: Fixed.

3.3.3 Time dependent data in NFT metadata

Severity: Medium Risk

Context:

- /sablier-labs/v2-core//src/SablierV2NFTDescriptor.sol#L73-L76

Description: The SablierV2NFTDescriptor contract creates an SVG image and embeds it in the `image` property of the JSON metadata returned by the `tokenURI()` function. However, the generated image contains time dependent data of the stream including the status, streamed amount, and progress as a percentage.

Although this information is up to date every time the `tokenURI()` function is called, marketplace platforms such as OpenSea cache this information as explained here (Although this is OS specific, most marketplaces will have similar caching optimizations in place)

> `image`: This is the URL to the image of the item. Can be just about any type of image (including SVGs, which will be cached into PNGs by OpenSea), and can be IPFS URLs or paths. We recommend using a 350 x 350 image.| `image_data`: Raw SVG image data, if you want to generate images on the fly (not recommended). Only use this if you're not including the `image` parameter.

Not only does OpenSea cache the image as a rasterized version but the complete metadata is cached as there is no way of knowing when the metadata has changed (except when ERC4906 events are used). To refresh the metadata the user needs to manually request to refresh the metadata for the NFT. Their API does allow for refreshing specific tokens programmatically but this is for obvious reasons heavily rate limited.

This means that whenever the image is displayed on a marketplace the time dependent data in the image will not be up to date and potential buyers will be misinformed about the status of the stream.

Recommendation: We recommend either removing the time dependent data from the image or including a timestamp when the image was generated. Alternatively, a solution making use of the `animation_url` could allow for a more time aware display of the image while keeping the time dependent data out of the standard `image` property.

Sablier: Changes applied via PR 568.

Cantina: Fixed. This solves the issue mostly except for the amount/percentage streamed between withdrawals. The bot triggering the 3rd party APIs can mitigate that.

3.3.4 Plugin or target with selfdestruct

Severity: Medium Risk

Context: General

Description: Until the 'Cancun' upgrade scheduled for later this year, whenever a `delegatecall` is made to a target contract or plugin that would do a `SELFDESTRUCT` the proxy would be destroyed at the end of the transaction and all ETH in the proxy would be sent to the address designated in the `SEFLDESTRUCT` call. This would obviously be problematic but would also mean that the current owner of the proxy could not create a new proxy as the registry's `proxies` mapping for the user still points to the destroyed proxy address.

After the upgrade, the `seflдestruct` opcode will not destroy the proxy contract but still send the ETH to the designated address.

Although target and plugin contracts are gated by the plugins and permissions access control mechanisms it is still possible for an owner to unsuspectedly call a contract that selfdestructs.

Recommendation: There's no easy way to detect whether a `selfdestruct` has been called in the `delegatecall`, but this does emphasize the need to assure target and plugin contracts are well vetted before being used. An optional (specified during the proxy's creation) allow list with audited target contracts could be used to aid in this assurance.

Sablier: Providing an allowlist of target contracts at deployment time would run counter to the intended computational universality of the proxy. The proxy owner should retain the ability to call any arbitrary computation performed in any target and any plugin (at their own risk).

Given the above, Cancun's imminence, and that it is very difficult (or impossible) to detect SELFDESTRUCT, we will mark this finding as "Acknowledged".

Cantina: Acknowledged.

3.3.5 Permission and plugins not reset on proxy ownership transfer

Severity: Medium Risk

Context:

- /prb-proxy/src/PRBProxyRegistry.sol#L104-L122
- /prb-proxy/src/abstracts/PRBProxyStorage.sol#L17-L20

Description: When a plugin is installed the mapping between function selectors and plugin contract is stored in the `plugins` mapping of the proxy. Similarly, the `permissions` mapping holds the target contracts envoys are allowed to call.

```
mapping(bytes4 method => IPRBProxyPlugin plugin) public plugins;
mapping(address envoy => mapping(address target => bool permission)) public permissions;
```

On ownership transfer of the proxy, these mappings are not reset, meaning any plugins installed or envoy permissions stay in effect. As these are mapping there is no easy way for the new owner to detect or list installed plugins and envoy permissions. As such in a scenario where ownership of a proxy is effectively transferred to another person (i.e. not to another account of the same person) this would allow the previous owner to maliciously install a plugin or set themselves as envoy to a malicious target contract before the transfer and potentially steal funds or cancel streams.

Recommendation: Consider either completely removing the ownership transfer functionality or resetting the plugins and permissions on ownership transfer. Alternatively, as any owner can only own one proxy these permissions could be stored in the registry based on the owner address instead of the proxy address.

Sablier: Changes applied in PR 119 and PR 113.

Cantina: Fixed.

3.3.6 Plugins and permissions storage variables are unprotected

Severity: Medium Risk

Context:

- /prb-proxy/src/abstracts/PRBProxyStorage.sol#L17-L20
- /prb-proxy/blob//src/PRBProxy.sol#L53
- /prb-proxy/src/PRBProxy.sol#L89
- /prb-proxy/src/PRBProxy.sol#L148

Description: The installed plugins and the permissions given to the envoys are stored in mappings defined in the `PRBProxyStorage` contract which is inherited by he proxy and as such they are stored in the proxy. Any target or plugin contract is supposed to inherit the same `PRBProxyStorage` contract to avoid storage slot collision.

```
abstract contract PRBProxyStorage is IPRBProxyStorage {
    address public override owner;
    uint256 public override minGasReserve;
    mapping(bytes4 method => IPRBProxyPlugin plugin) public plugins;
    mapping(address envoy => mapping(address target => bool permission)) public permissions;
```

This however can lead to accidental or malicious manipulation of the mappings which represent important access control features of the proxy. Some examples of possible scenarios are:

- A faulty target or plugin contract accidently overwriting these storages slots.
- A malicious target or plugin contract assigning the attacker as envoy for other malicious contracts.
- A malicious envoy using faulty target contracts to try and manipulate the mappings to gain unauthorized control.

This problem also applies to the `owner` and `minGasReserve` storage variables, but they have additional impact which is addressed in other findings.

Recommendation: It is best to not store these variables inside the proxy contract. They are better stored in the `PRBProxyRegistry` based on the proxy address or as discussed in the "Permission and plugins are not reset on proxy ownership transfer" finding based on the owner address.

Alternatively to at least protect against accidental storage collision use EIP1967 type storage with a custom namespace.

Sablier: Applied your recommendation here by moving both plugins and permissions to the registry, where only the owner is allowed to update them:

- PR 120.

Basically, the proxy contract itself doesn't have any storage layout anymore now, since we've also removed `transferOwnership` and minGasReserve:

- PR 119.
- PR 114.

Cantina: Fixed.

3.3.7 Deployment front running protection not effective

Severity: Medium Risk

Context:

- PRBProxyRegistry.sol#L115
- PRBProxyRegistry.sol#L142
- PRBProxyRegistry.sol#LL67C41-L67C60

Description: Deployment of a proxy is protected from frontrunning by salting the CREATE2 with the `tx.origin` of the deployer.

```
function _deploy(address owner) internal returns (IPRBProxy proxy) {
    bytes32 seed = nextSeeds[tx.origin];

    // Prevent front-running the salt by hashing the concatenation of "tx.origin" and the user-provided seed.
    bytes32 salt = keccak256(abi.encode(tx.origin, seed));

    // Deploy the proxy with CREATE2.
    transientProxyOwner = owner;
    proxy = new PRBProxy{ salt: salt }();
    delete transientProxyOwner;

    // Set the proxy for the owner.
    proxies[owner] = proxy;
}
```

While this does protect users from having their proxy address sniped from the mempool (which would cause a revert with `CREATE`), it does not protect against other opportunities for a frontrunning attacker to cause deployment to revert.

This is due to the `noProxy()` modifier, which will revert if the owner already has a proxy deployed:

```
modifier noProxy(address owner) {
    IPRBProxy proxy = proxies[owner];
    if (address(proxy) != address(0)) {
        revert PRBProxyRegistry_OwnerHasProxy(owner, proxy);
    }
    _;
}
```

However, an attacker can force a proxy on another user at any time using `deployFor()` or `transferOwnership()` to the victim before their transaction is processed.

In the least impactful case this will just mean the victim will have to burn the proxy (for which there is no function, so would involve transferring the proxy to a random address) and deploy the proxy again.

In the worst case, the user may not notice the revert (or upon noticing the revert, checks for a proxy they own via the public `proxies` mapping in the registry and determine that it must have worked, since they have one).

Unbeknownst to the victim, the attacker could have installed a malicious plugin or given themselves permission to execute on a (malicious) target contract during their frontrunning. Neither of these changes are easily detectable (because they are stored as mappings), but would put the proxy in extreme danger.

Recommendation: Make the `transferOwnership()` and `deployFor()` a two step process in which the new owner has to accept the transfer before assigning the ownership in the `proxies` mapping.

Alternatively, as recommended in other issues, it may be preferable to make ownership of proxies immutable, which would solve this issue.

Sablier: Changes applied in PR 119 and PR 113.

Cantina: Fixed.

3.3.8 `refundableAmountOf` will return a value when `isCancelable` is false

Severity: Medium Risk

Context:

- SablierV2LockupLinear.sol#L209-L222
- SablierV2LockupDynamic.sol#L226-L239

Description: The `refundableAmountOf()` function is an external view function that returns the amount of tokens that can be refunded to the sender of the transaction.

It specifically checks that, if the stream is either depleted or canceled, it will return 0. Otherwise, it will return the amount deposited minus the streamed amount.

```
if (!_streams[streamId].isDepleted && !_streams[streamId].wasCanceled) {
    refundableAmount = _streams[streamId].amounts.deposited - _calculateStreamedAmount(streamId);
}
```

However, there is no check whether the stream is cancelable. In the case that `stream.isCancelable = false`, the amount that is refundable is actually zero, and it would be more accurate to return this value.

This can be particularly dangerous when interacting with other areas in DeFi, where, for example, a lending protocol might use this return value to determine the amount that the stream could be liquidated for, which in fact it cannot be liquidated at all.

Recommendation:

```
function refundableAmountOf(uint256 streamId)
    external
    view
    override
    notNull(streamId)
    returns (uint128 refundableAmount)
{
    // If the stream is neither depleted nor canceled, subtract the streamed amount from the deposited amount.
    // Both of these checks are needed because {_calculateStreamedAmount} does not look up the stream's status.
-    if (!_streams[streamId].isDepleted && !_streams[streamId].wasCanceled) {
+    if (!_streams[streamId].isDepleted && !_streams[streamId].wasCanceled && _streams[streamId].isCancelable)
{
        refundableAmount = _streams[streamId].amounts.deposited - _calculateStreamedAmount(streamId);
    }
    // Otherwise, if the stream is either depleted or canceled, the result is implicitly zero.
}
```

Sablier: Implemented via PR 547.

It's worth mentioning that we removed the `wasCanceled` check due to a protocol invariant: if `wasCanceled` is true then `isCancelable` is false.

Cantina: Fixed, but just for precision, the invariant is the other way around. If `wadCanceled` is true, then `isCancelable` is false.

3.4 Low Risk

3.4.1 Permissions and plugins stored as mappings

Severity: Low Risk

Context:

- /prb-proxy//src/abstracts/PRBProxyStorage.sol#L17-L20

Description: Both the `permission` and the `plugins` are stored in the proxy in the form of mappings.

```
mapping(bytes4 method => IPRBProxyPlugin plugin) public plugins;
mapping(address envoy => mapping(address target => bool permission)) public permissions;
```

This means it is hard for an owner to detect which permissions exist and which plugins have been installed, as they'd have to know which method or envoy/target contract to query, which is exactly the information they would be querying.

As this pertains to access control settings on the proxy it is critical users are able to easily determine who has access to their proxy.

Recommendation: Consider using another storage type like OpenZeppelin's EnumerableMap

Sablier: Acknowledged. Strictly speaking, the issue is not that the permissions and the plugins are mappings - it's that there are no enumerable arrays for each.

Taking plugins as an example, there's no way to see, at a glance, what all plugins a particular proxy has installed. However, the plugins themselves would still be stored in a mapping.

Cantina: Acknwoledged.

3.4.2 No protection against non plugin or non target contracts

Severity: Low Risk

Context: No protection against non plugin or non target contracts

Description: PRBProxy allows any type of contract to be `delegatecall`ed into. When a user specifies a contract as target that was not specifically coded to be used in this way this can cause problems at the proxy level. One example of what can go wrong is the possible storage collision between proxy and targeted contract.

The proxy is in effect a smart contract wallet for the user but with the difference that the `execute()` function. As a user it's easy to forget that the `execute()` function does a `delegatecall` an not a standard `call`. For this reason it would be prudent to build in a protection that only contracts that are specifically built to integrate with PRBProxy are used.

Recommendation: Consider using an EIP165 style `supportsInterface(bytes4)` or `isPRBProxyTarget()` call to assure the target is compliant with PRBProxy. If `delegatecall`ing into more generalized libraries consider creating a secondary `execute()` function that does not impose this restriction.

Sablier: Acknowledged.

Cantina: Acknowledged.

3.4.3 Infinite approval for Sablier lockup contracts

Severity: Low Risk

Context:

- /v2-periphery/src/SablierV2ProxyTarget.sol#L604

Description: The SablierV2ProxyTarget uses infinite approval for funds used to create streams.

```
    function _approve(address sablierContract, IERC20 asset, uint256 amount) internal {
        uint256 allowance = asset.allowance({ owner: address(this), spender: sablierContract });
        if (allowance < amount) {
            asset.forceApprove({ spender: sablierContract, value: type(uint256).max });
        }
    }
```

Once a stream has been created for a certain ERC20 the allowance will stay in effect and the Sablier lockup contracts can use the funds in the proxy contract to create streams. Depending on the functionality provided by other installed plugins or envoy/target contract permissions the funds in the contracts could be used to create streams on the Sablier lockup contracts directly without requiring a permit2 allowance signature from the owner.

Recommendation: Consider setting the approval to the exact amount to avoid leaving an infinite approval active on tokens for the SablierLockup contracts.

Sablier: Implemented via PR 97.

Cantina: Fixed.

3.4.4 Proxies cannot be burned, and thus must be sent to potentially unsafe addresses

Severity: Low Risk

Context:

- PRBProxyRegistry.sol#L104-L122
- PRBProxy.sol#L121-L129

Description: There is no mechanism for users to burn a proxy they own.

This might be a common need since it is currently possible for any user to thrust a proxy upon another user (via `deployFor()` or `transferOwnership()`), and each user can only have one proxy. Thus, users need a way to discard of proxies they have been sent.

It is possible for users to simply transfer their proxies to dead addresses. However, the `transferOwnership()` function contains the `noProxy(newOwner)` modifier. This means that we cannot send a proxy to an address that already has one.

```
function transferOwnership(address newOwner) external override noProxy(newOwner) {
    // Check that the caller has a proxy.
    IPRBProxy proxy = proxies[msg.sender];
    if (address(proxy) == address(0)) {
        revert PRBProxyRegistry_OwnerDoesNotHaveProxy({ owner: msg.sender });
    }

    // Delete the proxy for the caller.
    delete proxies[msg.sender];

    // Set the proxy for the new owner.
    proxies[newOwner] = proxy;

    // Transfer the proxy.
    proxy.transferOwnership(newOwner);

    // Log the transfer of the proxy ownership.
    emit TransferOwnership({ proxy: proxy, oldOwner: msg.sender, newOwner: newOwner });
}
```

```
modifier noProxy(address owner) {
    IPRBProxy proxy = proxies[owner];
    if (address(proxy) != address(0)) {
        revert PRBProxyRegistry_OwnerHasProxy(owner, proxy);
    }
    _;
}
```

As a result, only one user can burn their proxy by sending it to the 0 address, one user can send to the 1 address, etc.

The result is that many users will need to input random addresses until they find one that's not used.

Beyond being a poor user experience, it is well known that human attempts at randomness aren't very random, and inputting a random address to send a proxy to might result in some risk (especially given that allowances are not reset on transfer).

Recommendation: `PRBProxyRegistry` should have a `burnProxy()` function, which resets the callers' ownership to 0, while not assigning the proxy to a new owner.

Sablier: Changes applied in PR 119.

Cantina: Fixed.

3.4.5 `batchCreate` functions can overflow total amount and revert

Severity: Low Risk

Context:

- SablierV2ProxyTarget.sol#L164-L173

Description: When streams are batch created through the proxy target contract, the total amount needed to fund the streams is calculated in the following loop:

```
// Calculate the sum of all of stream amounts. It is safe to use unchecked addition because one of the create
// transactions will revert if there is overflow.
uint256 i;
uint128 transferAmount;
for (i = 0; i < batchSize;) {
    unchecked {
        transferAmount += batch[i].totalAmount;
        i += 1;
    }
}
```

The comment above the code states that the overflow can only happen if the create transactions overflow, but this isn't the case. Since the `totalAmount` of a given batch can be as high as uint128 without reverting and the `transferAmount` is a uint128, it is possible that the sum of multiple batches will overflow.

This can lead to the `transferAmount` being lower than the real sum of each `totalAmount`.

The result is that, when `_transferAndApprove()` is called, only the lower `transferAmount` is transferred into the proxy. Then, as the create functions are called, there will eventually be insufficient funds to create the streams, and the transaction will revert.

This overflow cannot be abused by a user in any malicious way, but it could impact users of obscure tokens who plan to send multiple streams for near the maximum amount of uint128.

Recommendation: The `amount` argument to the `_transferAndApprove()` function is a uint160.

This is the only place where the `transferAmount` is used, so there is no need to limit it to a uint128 and risk overflow in the loop.

Instead, the original `transferAmount` declaration should be changed to a uint160.

```
uint256 i;
-uint128 transferAmount;
+uint160 transferAmount;
for (i = 0; i < batchSize;) {
    unchecked {
        transferAmount += batch[i].totalAmount;
        i += 1;
    }
}
```

Sablier: Changes applied in PR 99.

Cantina: Fixed.

3.4.6 Receiver using proxy cannot cancel individual stream

Severity: Low Risk

Context: SablierV2ProxyTarget.sol#L82-L95

Description: When a stream is canceled using the proxy target, the `cancel()` function attempts to forward the refund back to the proxy owner with the following logic:

```
function cancel(ISablierV2Lockup lockup, uint256 streamId) public onlyDelegateCall {
    // Retrieve the asset used for streaming.
    IERC20 asset = lockup.getAsset(streamId);

    // Retrieve the refunded amount.
    uint256 refundedAmount = lockup.refundableAmountOf(streamId);

    // Cancel the stream.
    lockup.cancel(streamId);

    // Forward the refunded amount to the proxy owner. This cannot be zero because settled streams cannot be
    // canceled.
    asset.safeTransfer({ to: owner, value: refundedAmount });
}
```

If the user calling the function is the `sender`, this call to `refundableAmountOf()` will return the correct value and the function will work as intended.

However, if the user calling the function is the `receiver`, the call to `refundableAmountOf()` will return the amount refunded to the seller. No assets will be returned to the `receiver`, since they need to be withdrawn in a separate transaction.

Therefore, when `asset.safeTransfer()` is called, there will be no funds to perform this and the function will revert.

Recommendation: `cancel()` should use the same mechanism from the other cancellation functions on the proxy target, which looks up the asset balance in advance of the call, and then sends the difference after.

Sablier: Changed applied via PR 98.

Cantina: Fixed.

3.5 Informational

3.5.1 Atypical transferOwnership() function on PRBProxyRegistry

Severity: Informational

Context:

- /prb-proxy/src/PRBProxyRegistry.sol#L104-L118

Description: The `transferOwnership()` function transfers the ownership of a proxy to a new owner. To an uneducated user however the function name might be confusing as a typical `transferOwnership()` which transfers the ownership of the contract being called on.

Recommendation: Consider renaming the `transferOwnership()` function to `transferProxyOwnership()` as this more clearly represent the functionality it provides.

Sablier: Changes applied in PR 119.

Cantina: Fixed.

3.5.2 No default function for transferring ETH or tokens

Severity: Informational

Context: General

Description: None of the contracts (`PRBProxy`, `PRBAnnex` or `SablierV2ProxyTarget`) provide any way to transfer ETH or Tokens out of the proxy. This seems like a common use case where funds that somehow arrived in the proxy need to be sent out or withdrawn by the owner.

Although a contract can be deploy that performs this in a `delegatecall` it is inconvenient to not provide this by default as users would need to deploy their own contracts for doing this or trust contracts created by others.

Recommendation: Consider providing this as a default functionality for the owner of the contract.

Sablier: Acknowledged.

3.5.3 No mirror function for `burn()`

Severity: Informational

Context:

- SablierV2Lockup.sol#L104-L119

Description: Each of the functions callable by the sender or receipient have a "mirror" function on `SablierV2ProxyTarget.sol` so that proxy users are able to easily interact with the Sablier protocol.

While it appears to be the intention the the proxy will primarily be used by the sender, it is also likely that recipients will use the proxy as well.

All functions that a recipient will need are available in the proxy target, except `burn()` which is excluded.

Recommendation: Add the following function to `SablierV2ProxyTarget.sol`:

```
function burn(ISablierV2Lockup lockup, uint256 streamId) external onlyDelegateCall {
    lockup.burn(streamId);
}
```

Sablier: Fixed in PR 96.

Cantina: Fixed.

3.5.4 Malicious asset contract can force Sablier's `tokenURI()` function to revert if called on chain

Severity: Informational

Context:

- SablierV2NFTDescriptor.sol#L306-L311
- SablierV2NFTDescriptor.sol#L316-L325

Description: When `tokenURI()` is called on a Sablier contract, it pushes the logic to NFT Descriptor to return an SVG representing the stream.

While this `SablierV2NFTDescriptor#tokenURI()` function is expected to only be called by front ends, in the case that it is called during an on chain transaction, a malicious asset can force it to revert.

The function makes an effort to avoid this by not reverting when the calls out to the underlying asset fail, but these efforts do not cover all cases when a revert could happen.

The function can still be caused to revert in two ways:

1) `safeAssetDecimals()` and `safeAssetSymbol()` both don't specify amount of gas to send, so a malicious asset could construct these functions to burn all the gas passed (63/64ths of what's remaining), which would leave the `tokenURI()` function without enough gas to finish the call.
2) In the event that a specific amount of gas was sent, the malicious asset could return a large amount of data to force memory expansion, the cost of which is quadratic. Because there are two calls, it

could cause x**2 more gas cost than it incurs, which could be sufficient to force the `tokenURI()` function to run out of gas. More about this attack here.

Recommendation: Pass only a small amount of gas to these two calls (less than 20k), which will prevent either of these attacks.

Sablier: After pondering this issue for a while, we are happy to mark it as "Acknowledged".

1. Passing a hard-coded amount of gas would introduce other problems. For example, ERC-20 tokens on chains like Polygon are much more gas expensive than on Ethereum Mainnet, and so we would have to introduce an immutable variable to account for edge cases of this sort.
2. I can't think of any incentive for why anyone would want to block the execution of tokenURI, which is basically a visual asset data URI. If anything, not having one's SVG show up properly will play to their disadvantage because the asset would not appear on NFT marketplaces.

Cantina: Acknowledged.

4 Additional Comments

4.1 On-chain contracts

The reviewed codebase has been verified to correspond to the following on-chain contracts:

Ethereum Mainnet

Contract	Address
`SablierV2LockupLinear`	0xB10daee1FCF62243aE27776D7a92D39dC8740f95
`SablierV2LockupDynamic`	0x39EFdC3dbB57B2388CcC4bb40aC4CB1226Bc9E44
`SablierV2NFTDescriptor`	0x98F2196fECc01C240d1429B624d007Ca268EEA29
`SablierV2Comptroller`	0xC3Be6BffAeab7B297c03383B4254aa3Af2b9a5BA

Table 1: `v2-core` contracts on Ethereum Mainnet

Contract	Address
`SablierV2Archive`	0x0Be20a8242B0781B6fd4d453e90DCC1CcF7DBcc6
`SablierV2ProxyPlugin`	0x9bdebF4F9adEB99387f46e4020FBf3dDa885D2b8
`SablierV2ProxyTargetPermit2`	0x297b43aE44660cA7826ef92D8353324C018573Ef
`SablierV2ProxyTargetApprove`	0x638a7aC8315767cEAfc57a6f5e3559454347C3f6

Table 2: `v2-periphery` contracts on Ethereum Mainnet

Arbitrum One

Contract	Address
`SablierV2LockupLinear`	0x197D655F3be03903fD25e7828c3534504bfe525e
`SablierV2LockupDynamic`	0xA9EfBEf1A35fF80041F567391bdc9813b2D50197
`SablierV2NFTDescriptor`	0xc245d6C9608769CeF91C3858e4d2a74802B9f1bB
`SablierV2Comptroller`	0x17Ec73692F0aDf7E7C554822FBEAACB4BE781762

Table 3: `v2-core` contracts on Arbitrum One

Contract	Address
`SablierV2Archive`	0xDFa4512d07AbD4eb8Be570Cd79e2e6Fe21ff15C9
`SablierV2ProxyPlugin`	0x9aB73CA73c89AF0bdc69642aCeb23CC6A55A514C
`SablierV2ProxyTargetPermit2`	0xB7185AcAF42C4966fFA3c81486d9ED9633aa4c13
`SablierV2ProxyTargetApprove`	0x90cc23dc3e12e80f27c05b8137b5f0d2b1edfa20

Table 4: `v2-periphery` contracts on Arbitrum One

Arbitrum Nova

Contract	Address
SablierV2LockupLinear	0x18306C9550AbfE3F5900d1206FFdce9ce5763A89
SablierV2LockupDynamic	0xd6b66A8D797c1e83DdEcE8f483E7D1264B9DFDa6
SablierV2NFTDescriptor	0xE88d26d1E8802be5cc023264b3FccF63Bc38C20c
SablierV2Comptroller	0x203f1722d4adb9b67bf652c878d0dc3cc8099113

Table 5: `v2-core` contracts on Arbitrum Nova

Contract	Address
SablierV2Archive	0x17DE7707D0b25F878Ae4FaC03cdE2481CD616EDd
SablierV2ProxyPlugin	0x1f09ce4be5ad6e76cda6242af91921440df2306e
SablierV2ProxyTargetPermit2	0x4487F233bdf7d3C977F936891D5A0Ff1b275A2a8

Table 6: `v2-periphery` contracts on Arbitrum Nova

Avalanche

Contract	Address
SablierV2LockupLinear	0x610346E9088AFA70D6B03e96A800B3267E75cA19
SablierV2LockupDynamic	0x665d1C8337F1035cfBe13DD94bB669110b975f5F
SablierV2NFTDescriptor	0xFd050AFA2e04aA0596947DaD3Ec5690162aDc77F
SablierV2Comptroller	0x66F5431B0765D984f82A4fc4551b2c9ccF7eAC9C

Table 7: `v2-core` contracts on Avalanche

Contract	Address
SablierV2Archive	0x7b1ef644ce9a625537e9e0c3d7fef3be667e6159
SablierV2ProxyPlugin	0x17167A7e2763121e263B4331B700a1BF9113b387
SablierV2ProxyTargetPermit2	0x48B4889cf5d6f8360050f9d7606505F1433120BC
SablierV2ProxyTargetApprove	0x817fE1364A9d57d1fB951945B53942234163Ef10

Table 8: `v2-periphery` contracts on Avalanche

Base

Contract	Address
SablierV2LockupLinear	0x6b9a46C8377f21517E65fa3899b3A9Fab19D17f5
SablierV2LockupDynamic	0x645B00960Dc352e699F89a81Fc845C0C645231cf
SablierV2NFTDescriptor	0xEFc2896c29F70bc23e82892Df827d4e2259028Fd
SablierV2Comptroller	0x7Faaedd40B1385C118cA7432952D9DC6b5CbC49e

Table 9: `v2-core` contracts on Base

Contract	Address
SablierV2Archive	0x1C5Ac71dd48c7ff291743e5E6e3689ba92F73cC6
SablierV2ProxyPlugin	0x50E8B9dC7F28e5cA9253759455C1077e497c4232
SablierV2ProxyTargetPermit2	0x0648C80b969501c7778b6ff3ba47aBb78fEeDF39
SablierV2ProxyTargetApprove	0xf19576Ab425753816eCbF98aca8132A0f693aEc5

Table 10: `v2-periphery` contracts on Base

BNB Smart Chain

Contract	Address
SablierV2LockupLinear	0x3FE4333f62A75c2a85C8211c6AeFd1b9Bfde6e51
SablierV2LockupDynamic	0xF2f3feF2454DcA59ECA929D2D8cD2a8669Cc6214
SablierV2NFTDescriptor	0x3daD1bF57edCFF979Fb68a802AC54c5AAfB78F4c
SablierV2Comptroller	0x33511f69A784Fd958E6713aCaC7c9dCF1A5578E8

Table 11: `v2-core` contracts on BNB Smart Chain

Contract	Address
SablierV2Archive	0xeDe48EB173A869c0b27Cb98CC56d00BC391e5887
SablierV2ProxyPlugin	0xC43b2d8CedB71df30F45dFd9a21eC1E50A813bD6
SablierV2ProxyTargetPermit2	0x135e78B8E17B1d189Af75FcfCC018ab2E6c7b879
SablierV2ProxyTargetApprove	0xc9bf2A6bD467A813908d836c1506efE61E465761

Table 12: `v2-periphery` contracts on BNB Smart Chain

Gnosis

Contract	Address
SablierV2LockupLinear	0x685E92c9cA2bB23f1B596d0a7D749c0603e88585
SablierV2LockupDynamic	0xeb148E4ec13aaA65328c0BA089a278138E9E53F9
SablierV2NFTDescriptor	0x8CE9Cd651e03325Cf6D4Ce9cfa74BE79CDf6d530
SablierV2Comptroller	0x73962c44c0fB4cC5e4545FB91732a5c5e87F55C2

Table 13: `v2-core` contracts on Gnosis

Contract	Address
SablierV2Archive	0xF4A6F47Da7c6b26b6Dd774671aABA48fb4bFE309
SablierV2ProxyPlugin	0xc84f0e95815A576171A19EB9E0fA55a217Ab1536
SablierV2ProxyTargetPermit2	0x5B144C3B9C8cfd48297Aeb59B90a024Ef3fCcE92
SablierV2ProxyTargetApprove	0x89AfE038714e547C29Fa881029DD4B5CFB008454

Table 14: `v2-periphery` contracts on Gnosis

Optimism

Contract	Address
SablierV2LockupLinear	0xB923aBdCA17Aed90EB5EC5E407bd37164f632bFD
SablierV2LockupDynamic	0x6f68516c21E248cdDfaf4898e66b2b0Adee0e0d6
SablierV2NFTDescriptor	0xe0138C596939CC0D2382046795bC163ad5755e0E
SablierV2Comptroller	0x1EECb6e6EaE6a1eD1CCB4323F3a146A7C5443A10

Table 15: `v2-core` contracts on Optimism

Contract	Address
SablierV2Archive	0x9A09eC6f991386718854aDDCEe68647776Befd5b
SablierV2ProxyPlugin	0x77C8516B1F327890C956bb38F93Ac2d6B24795Ea
SablierV2ProxyTargetPermit2	0x194ed7D6005C8ba4084A948406545DF299ad37cD
SablierV2ProxyTargetApprove	0x8a6974c162fdc7Cb67996F7dB8bAAFb9a99566e0

Table 16: `v2-periphery` contracts on Optimism

Polygon

Contract	Address
SablierV2LockupLinear	0x67422C3E36A908D5C3237e9cFfEB40bDE7060f6E
SablierV2LockupDynamic	0x7313AdDb53f96a4f710D3b91645c62B434190725
SablierV2NFTDescriptor	0xA820946EaAceB2a85aF123f706f23192c28bC6B9
SablierV2Comptroller	0x9761692EDf10F5F2A69f0150e2fd50dcecf05F2E

Table 17: v2-core contracts on Polygon

Contract	Address
SablierV2Archive	0xA2f5B2e798e7ADd59d85d9b76645E6AC13fC4e1f
SablierV2ProxyPlugin	0xBe4cad0e99865CC62787Ecf029aD9DD4815d3d2e
SablierV2ProxyTargetPermit2	0x576743075fc5F771bbC1376c3267A6185Af9D62B
SablierV2ProxyTargetApprove	0xccA6dd77bA2cfcccEdA01A82CB309e2A17901682

Table 18: v2-periphery contracts on Polygon

Scroll

Contract	Address
SablierV2LockupLinear	0x80640ca758615ee83801EC43452feEA09a202D33
SablierV2LockupDynamic	0xde6a30D851eFD0Fc2a9C922F294801Cfd5FCB3A1
SablierV2NFTDescriptor	0xC1fa624733203F2B7185c3724039C4D5E5234fE4
SablierV2Comptroller	0x859708495E3B3c61Bbe19e6E3E1F41dE3A5C5C5b

Table 19: v2-core contracts on Scroll

Contract	Address
SablierV2Archive	0x94A18AC6e4B7d97E31f1587f6a666Dc5503086c3
SablierV2ProxyPlugin	0xED1591BD6038032a74D786A452A23536b3201490
SablierV2ProxyTargetPermit2	0x91154fc80933D25793E6B4D7CE19fb51dE6794B7
SablierV2ProxyTargetApprove	0x71CeA9c4d15fed2E58785cE0C05165CE34313A74

Table 20: v2-periphery contracts on Scroll

Arbitrum Goerli

Contract	Address
SablierV2LockupLinear	0x323B629635b6cFfe2453Aa2869c5957AfF55F445
SablierV2LockupDynamic	0xdc0a619fF975de6a08c7615ea383533fd265f2e3
SablierV2NFTDescriptor	0x740509d893BC15a31EAE8542683Ed32085c559cB
SablierV2Comptroller	0xECF737BDb9BB094489beCa39f0b9Ae66E0C14ba8

Table 21: `v2-core` contracts on Arbitrum Goerli testnet

Contract	Address
SablierV2Archive	0x4371d767Cd7991248D20eD61d425e1e70c6CEEab
SablierV2ProxyPlugin	0xD37832B8993bEe6F41A8183967a7488C6e2a3551
SablierV2ProxyTargetPermit2	0x2Ebd987e12432Ee3a74Fe0A55Afe5D866096e354

Table 22: `v2-periphery` contracts on Arbitrum Goerli testnet

Goerli

Contract	Address
SablierV2LockupLinear	0x6E3678c005815Ab34986D8d66A353Cd3699103DE
SablierV2LockupDynamic	0x4BE70EDe968e9dBA12DB42b9869Bec66bEDC17d7
SablierV2NFTDescriptor	0x1D83CDd66BCf0ea8c99E745cC868478d6C3633f0
SablierV2Comptroller	0x9B75F65bCCd05545C400145Cca29dA52DA57AC2b

Table 23: `v2-core` contracts on Goerli testnet

Contract	Address
SablierV2Archive	0xFd14E62e6fe4d96F033cf972556ae56D09Bd49cA
SablierV2ProxyPlugin	0x9CA1dFFC744318198bE9Cf92283A803CE16b698a
SablierV2ProxyTargetPermit2	0x0eE01680645c361B740ab4dCDdF238988eB20411
SablierV2ProxyTargetApprove	0x0e563B883dfe11469915194F8651a65212fdB96F

Table 24: `v2-periphery` contracts on Goerli testnet

Sepolia

Contract	Address
SablierV2LockupLinear	0xd4300c5bc0b9e27c73ebabdc747ba990b1b570db
SablierV2LockupDynamic	0x421e1E7a53FF360f70A2D02037Ee394FA474e035
SablierV2NFTDescriptor	0x3cb51943ebcea05b23c35c50491b3d296ff675db
SablierV2Comptroller	0x2006d43E65e66C5FF20254836E63947FA8bAaD68

Table 25: `v2-core` contracts on Sepolia testnet

Contract	Address
SablierV2Archive	0x83495d8DF6221f566232e1353a6e7231A86C61fF
SablierV2ProxyPlugin	0xa333c8233CfD04740E64AB4fd5447995E357561B
SablierV2ProxyTargetPermit2	0x5091900B7cF803a7407FCE6333A6bAE4aA779Fd4
SablierV2ProxyTargetApprove	0x105E7728C5706Ad41d194EbDc7873B047352F3d2

Table 26: `v2-periphery` contracts on Sepolia testnet