Information on the Platypus Finance exploit for the Aave Community

Overview

The Platypus Finance exploit (the exploit) occurred on February 16, 2023 during the time period of 07:16:54 PM +UTC to 07:51:08 PM +UTC at the blockchain network Avalanche C-Chain (the blockchain). The attacker, who is unknown in identity, exploited and transferred away $9.194 million of stablecoins in USD value to multiple wallet addresses from Platypus Finance, a decentralized stablecoin exchange platform (the platform). Through taking advantage of a vulnerability in the underlying smart contract code of the platform, the attacker deployed three flash loan capable smart contracts, and executed a series of three transactions calling each of the deployed contracts to execute the exploit. In one of the three transactions, the attacker implemented a logic in the exploit contract such that around $381k worth of stablecoins exploited from the platform's LP pools were directly transferred to Aave's Pool contract deployed on Avalanche during the transaction.

The vulnerability exists in a function that was neither accessible from the platform's front-end interface, nor it was officially documented. The attacker exploits the vulnerability, draining the LP pools where the stablecoins were deposited by the platform's users, without the platform or the platform's users' authorization.

In this report, we will focus on detailing the transaction that led to funds being maliciously drained to Aave's Pool contract from the platform, hence "the exploit" will subsequently be referring exclusively to the transaction where funds were drained to Aave's Pool contract, and not the other transactions initiated by the attacker. We will also explain the vulnerability in the platform, and how it was taken advantage of by the attacker.

Important details relevant to the exploit

1. Time of the exploit: Feb-16-2023 07:38:51 PM +UTC

2. Block number where the exploit happened: 26344274

3. The main blockchain address of the attacker that executed the exploit of the platform:

Address	Chain	Explorer
0xeff003d64046a6f521ba31f39405cb720e953958	Avalanche C-Chain	https://snowtrace.io/address/0xeff003d64046a6f521ba31f39405cb720e953958

4. The addresses of the smart contract containing the exploit logic, and were subsequently called by the attacker to execute the exploit:

Address	Explorer
0xf5d6007abb615654a95d33614a059fa59bcff390	https://snowtrace.io/address/0xf5d6007abb615654a95d33614a059fa59bcff390

5. The transaction hashes directly related to the exploit:

Transaction hash	Description	Explorer
0x8b47bec698b338205e3b520d91f236af9d1692bda765104a20ef063ed5bf0aa2	Deployment of the exploit logic (Creating a smart contract)	https://snowtrace.io/tx/0x8b47bec698b338205e3b520d91f236af9d1692bda765104a20ef063ed5bf0aa2
0x919266aa66d7c9a6af02dead5effc1cc68ab7b87890b52e5fc1e20a7041aa84d	Execution of the exploit logic (Calling the smart contract, the second exploit)	https://snowtrace.io/tx/0x919266aa66d7c9a6af02dead5effc1cc68ab7b87890b52e5fc1e20a7041aa84d

6. Blockchain addresses involved in the exploit:

Address	Description	Explorer	Note
0xf5d6007abb615654a95d33614a059fa59bcff390	Smart contract containing the exploit logic	https://snowtrace.io/address/0xf5d6007abb615654a95d33614a059fa59bcff390	Deployed by the attacker, responsible for the exploit logic
0xefF003D64046A6f521BA31f39405cb720E953958	EOA address used to call the smart contracts to initiate the exploit	https://snowtrace.io/address/0xeff003d64046a6f521ba31f39405cb720e953958	Responsible for exploiting Platypus Finance and sent the exploited funds to Aave's Pool contract
0x66357dCaCe80431aee0A7507e2E361B7e2402370	Platypus Finance's Pool router contract	https://snowtrace.io/address/0x66357dCaCe80431aee0A7507e2E361B7e2402370	Responsible for fulfilling stablecoin swapping requests.
0xff6934aac9c94e1c39358d4fdcf70aeca77d0ab0	Platypus Finance's MasterPlatypusV4 LP staking contract	https://snowtrace.io/address/0xff6934aac9c94e1c39358d4fdcf70aeca77d0ab0	The "emergencyWithdraw()" logic here was wrongly implemented and subsequently exploited by the attacker.
0x061da45081ACE6ce1622b9787b68aa7033621438	Platypus Finance's PlatypusTreasure USP stablecoin module	https://snowtrace.io/address/0x061da45081ACE6ce1622b9787b68aa7033621438	Responsible for main logic of the USP stablecoin system
0xaef735b1e7ecfaf8209ea46610585817dc0a2e16	Platypus Finance's LP-USDC pool contract	https://snowtrace.io/address/0xaef735b1e7ecfaf8209ea46610585817dc0a2e16	LP contract where USDC deposited by users here were swapped out using maliciously minted USP
0x909b0ce4fac1a0dca78f8ca7430bbafeeca12871	Platypus Finance's LP-USDC.e pool contract	https://snowtrace.io/address/0x909b0ce4fac1a0dca78f8ca7430bbafeeca12871	LP contract where USDC.e deposited by users here were swapped out using maliciously minted USP

0x776628a5c37335608dd2a9538807b9bba3869e14	Platypus Finance's LP-USDT pool contract	https://snowtrace.io/address/0x776628a5c37335608dd2a9538807b9bba3869e14	LP contract where USDT deposited by users here were swapped out using maliciously minted USP
0x0d26d103c91f63052fbca88aaf01d5304ae40015	Platypus Finance's LP-USDT.e pool contract	https://snowtrace.io/address/0x0d26d103c91f63052fbca88aaf01d5304ae40015	LP contract where USDT.e deposited by users here were swapped out using maliciously minted USP
0xc1daa16e6979c2d1229cb1fd0823491ea44555be	Platypus Finance's LP-DAI.e pool contract	https://snowtrace.io/address/0xc1daa16e6979c2d1229cb1fd0823491ea44555be	LP contract where DAI.e deposited by users here were swapped out using maliciously minted USP
0xe23f8ccdeb4e8ce5d9fe76782718cd85d12689c8	Platypus Finance's LP-BUSD pool contract	https://snowtrace.io/address/0xe23f8ccdeb4e8ce5d9fe76782718cd85d12689c8	LP contract where BUSD deposited by users here were swapped out using maliciously minted USP
0xa16bbab03b61810ba8633343d9ffc04b086506b5	Platypus Finance's LP-USP pool contract	https://snowtrace.io/address/0xa16bbab03b61810ba8633343d9ffc04b086506b5	LP contract where attacker deposits maliciously minted USP to exchange for other stablecoin assets
0x794a61358d6845594f94dc1db02a252b5b4814ad	Aave's Pool V3 contract	https://snowtrace.io/address/0x794a61358d6845594f94dc1db02a252b5b4814ad	Destination where the exploited funds were sent to

7. Amount of stablecoins stolen in the exploit: $380,594 approximately in USD value, of which includes the following digital stablecoin assets (rounded off):

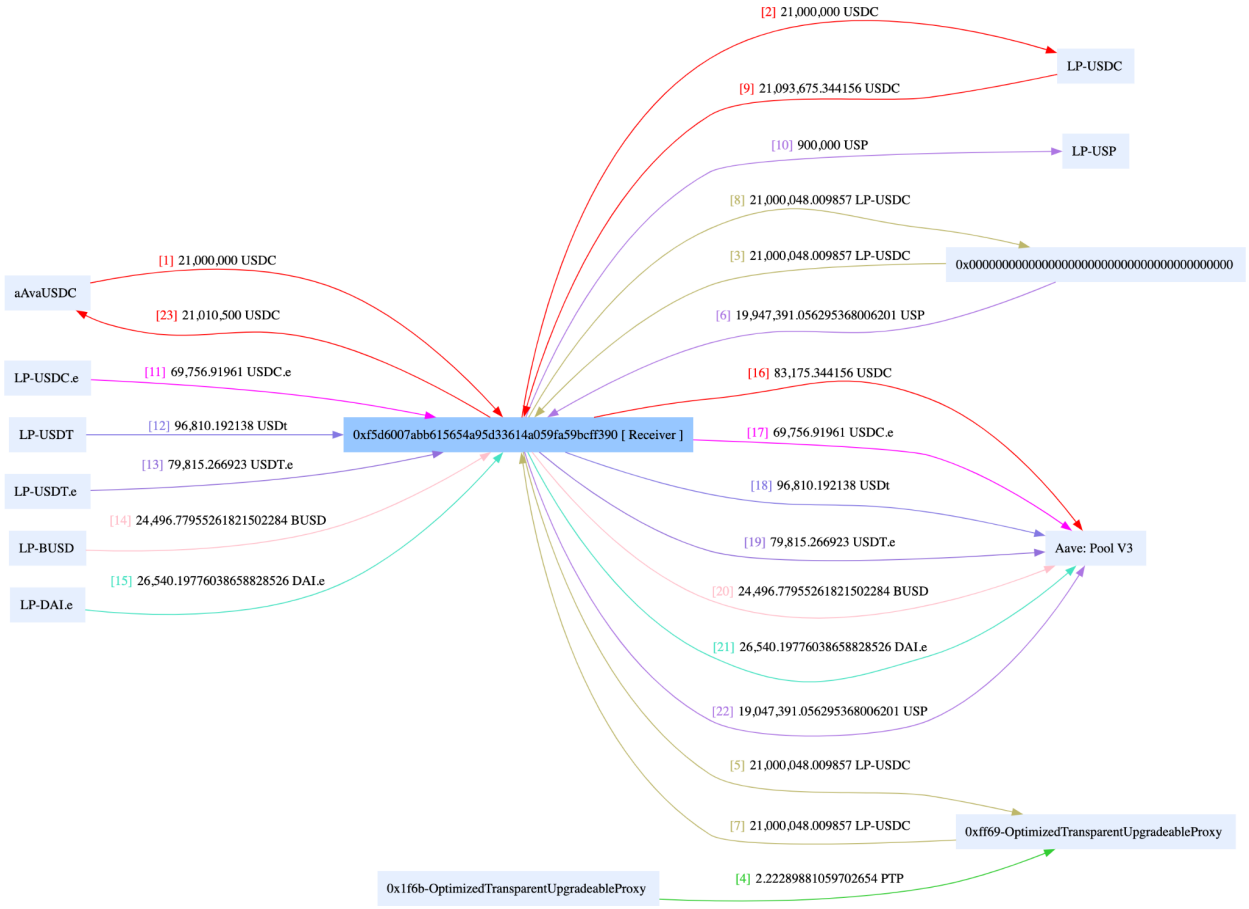
Name of the asset	Ticker	Contract address	Quantity
TetherToken	USDt	0x9702230A8Ea53601f5cD2dc00fDBc13d4dF4A8c7	96,810.192138
Tether USD (Bridged)	USDT.e	0xc7198437980c041c805A1EDcbA50c1Ce5db95118	79,815.266923
USD Coin	USDC	0xB97EF9Ef8734C71904D8002F8b6Bc66Dd9c48a6E	83,175.344156
USD Coin (Bridged)	USDC.e	0xA7D7079b0FEaD91F3e65f86E8915Cb59c1a4C664	69,756.91961
Dai Stablecoin (Bridged)	DAI.e	0xd586E7F844cEa2F87f50152665BCbc2C279D8d70	26,540.19776038658828526
Binance-Peg BUSD	BUSD	0x9C9e5fD8bbc25984B178FdCE6117Defa39d2db39	24,496.77955261821502284
USP Stablecoin	USP	0xdaCDe03d7Ab4D81fEDdc3a20fAA89aBAc9072CE2	19,047,391.056295368006201

Note:

- “Bridged” here means the asset was issued by the Avalanche Bridge, a cross-blockchain asset transfer solution built by Avalanche.
- Exact amounts can be verified through the transfer logs available here: https://snowtrace.io/tx/0x919266aa66d7c9a6af02dead5effc1cc68ab7b87890b52e5fc1e20a7041aa84d

8. Blockchain networks involved in the exploit: Avalanche C-Chain, EVM Chain ID 43114

Overall fund flow graph of the exploit (Generated with BlockSec's explorer):



Tx Hash	Blockchain explorer	BlockSec's transaction explorer	Download fund flow graph
0x919266aa66d7c9a6af02dead5effc1cc68ab7b87890b52e5fc1e20a7041aa84d	https://snowtrace.io/tx/0x919266aa66d7c9a6af02dead5effc1cc68ab7b87890b52e5fc1e20a7041aa84d	https://phalcon.blocksec.com/tx/avax/0x919266aa66d7c9a6af02dead5effc1cc68ab7b87890b52e5fc1e20a7041aa84d	https://i.imgur.com/YP2gfET.png

Technical details on the execution of the exploit

The concept of the exploit, in layman terms, was to leverage a vulnerability in the Platypus Finance smart contracts where the system logic has not implemented a proper account balance checking mechanics, leading the exploiter to essentially be able to “withdraw deposits from a bank and the bank proceeds the withdrawal, without noting that the deposits were used in a collateralized loan that is still yet to be repaid”, and therefore lead to a double-spending problem.

Below are the links containing the full logic for the smart contracts involved in the vulnerability:

Name	Related functions	Implementation address	Block Explorer	Proxy address
MasterPlatypusV4.sol	withdraw(), emergencyWithdraw()	0xc007f27b757a782c833c568f5851ae1dfe0e6ec7	https://snowtrace.io/address/0xc007f27b757a782c833c568f5851ae1dfe0e6ec7#code (File 1 of 18)	0xfF6934aAC9C94E1C39358D4fDCF70aeca77D0AB0
PlatypusTreasure.sol	isSolvent()	0xbcd6796177ab8071f6a9ba2c3e2e0301ee91bef5	https://snowtrace.io/address/0xbcd6796177ab8071f6a9ba2c3e2e0301ee91bef5#code (File 57 of 69)	0x061da45081ace6ce1622b9787b68aa7033621438

Note:

- “MasterPlatypusV4.sol” is responsible for the deposit and withdrawal of LP shares, where rewards will be accrued to depositors over time. It has a similar functionality as MasterChef from SushiSwap
- “PlatypusTreasure.sol” is responsible for the main logic of the USP over-collateralized stablecoin system by Platypus Finance

Below is the LP withdrawal logic in the platform that is implemented in the platform’s frontend application, and is the default process when users interact with the platform’s frontend to request for a withdrawal for LP tokens:

```
544     /// @notice Withdraw LP tokens from MasterPlatypus.
545     /// @notice Automatically harvest pending rewards and sends to user
546     /// @param _pid the pool id
547     /// @param _amount the amount to withdraw
548     function withdraw(uint256 _pid, uint256 _amount)
549         external
550         override
551         nonReentrant
552         whenNotPaused
553         returns (uint256 reward, uint256[] memory additionalRewards)
554     {
555         (reward, additionalRewards) = _withdrawFor(_pid, msg.sender, msg.sender, _amount);
556 
557         if (address(platypusTreasure) != address(0x00)) {
558             (bool isSolvent, ) = platypusTreasure.isSolvent(msg.sender, address(poolInfo[_pid].lpToken), true);
559             require(isSolvent, 'remaining amount exceeds collateral factor');
560         }
561     }
```

Below is the LP withdrawal logic in the platform's system that poses a vulnerability for the exploit to become viable:

```
576     /// @notice Withdraw without caring about rewards. EMERGENCY ONLY.
577     /// @param _pid the pool id
578     function emergencyWithdraw(uint256 _pid) public nonReentrant {
579         PoolInfo storage pool = poolInfo[_pid];
580         UserInfo storage user = userInfo[_pid][msg.sender];
581 
582         if (address(platypusTreasure) != address(0x00)) {
583             (bool isSolvent, ) = platypusTreasure.isSolvent(msg.sender, address(poolInfo[_pid].lpToken), true);
584             require(isSolvent, 'remaining amount exceeds collateral factor');
585         }
586 
587         // reset rewarder before we update lpSupply and sumOfFactors
588         IBoostedMultiRewarder rewarder = pool.rewarder;
589         if (address(rewarder) != address(0)) {
590             rewarder.onPtpReward(msg.sender, user.amount, 0, user.factor, 0);
591         }
592 
593         // SafeERC20 is not needed as Asset will revert if transfer fails
594         pool.lpToken.transfer(address(msg.sender), user.amount);
595 
596         // update non-dialuting factor
597         pool.sumOfFactors -= user.factor;
598 
599         user.amount = 0;
600         user.factor = 0;
601         user.rewardDebt = 0;
602 
603         emit EmergencyWithdraw(msg.sender, _pid, user.amount);
604     }
```

Both the above logic aims to achieve the purpose of allowing users to withdraw their LP tokens from a LP staking contract. Platypus Finance allows users to deposit their stablecoins to become a liquidity provider, which in exchange will receive LP tokens representing their share of the deposits. As liquidity providers, they can further stake their LP into the platform to receive rewards for liquidity provisioning.

The platform also has a feature to allow stakers to mint a new token called "USP" based on the underlying value of the LP tokens they have staked to the platform. Hence, to obtain USP, the user flow for example is: Deposit USDC -> Receive USDC-LP -> Deposit USDC-LP -> Receive USP. The redemption (withdrawal) flow is therefore: Repay USP -> Redeem USDC-LP -> Repay USDC-LP -> Redeem USDC.

For the two withdrawal logic screenshotted above, "function withdraw()" is what normally users experience when they choose to redeem their USDC-LP. "function emergencyWithdraw()" is a function that straightforwardly helps users to redeem their USDC-LP and forfeit any LP staking reward incentives accrued in the platform. The platform never implemented the "function emergencyWithdraw()" on the frontend user-interface, and never intended it to be part of the user experience the platform was designed to serve.

As mentioned above, the vulnerability lies on an improper account balance checking mechanics in "function emergencyWithdraw()", which the logic first calls "platypusTreasure.isSolvent", a checker for solvency in the USP system, where it aims to make sure that the user has sufficient collateral in the form of LP tokens to back the USP the user has minted (created), subsequent

the check, the logic goes on by transferring the deposited LP tokens to the user as part of the redemption process:

```
582         if (address(platypusTreasure) != address(0x00)) {
583             (bool isSolvent, ) = platypusTreasure.isSolvent(msg.sender, address(poolInfo[_pid].lpToken), true);
584             require(isSolvent, 'remaining amount exceeds collateral factor');
585         }
586
587         // reset rewarder before we update lpSupply and sumOfFactors
588         IBoostedMultiRewarder rewarder = pool.rewarder;
589         if (address(rewarder) != address(0)) {
590             rewarder.onPtpReward(msg.sender, user.amount, 0, user.factor, 0);
591         }
592
593         // SafeERC20 is not needed as Asset will revert if transfer fails
594         pool.lpToken.transfer(address(msg.sender), user.amount);
```

^snippet of “function emergencyWithdraw()” at MasterPlatypus

The “platypusTreasure.isSolvent” check will look up the amount of LP tokens a user has deposited, if the deposited value of LP tokens exceeds the USP the user has minted, “isSolvent” will be “True”.

This poses the problem that when a user calls “emergencyWithdraw”, the check logic (Line 583) comes **before** the transfer logic (Line 594), which the check will pass, as there are no collateral movements (transferred out), and the user will only become insolvent **after** the LP tokens have transferred away from the platform to the user, because the LP tokens are no longer deposited in the platform and therefore insufficient collateral posed by the user.

Hence, the “function emergencyWithdraw()” poses a vulnerability where a user can bypass the solvency check to withdraw their LP tokens collateral while having a USP stablecoin collateralized position opened, which was unfortunately taken advantage of by the attacker.

```
555         (reward, additionalRewards) = _withdrawFor(_pid, msg.sender, msg.sender, _amount);
556
557         if (address(platypusTreasure) != address(0x00)) {
558             (bool isSolvent, ) = platypusTreasure.isSolvent(msg.sender, address(poolInfo[_pid].lpToken), true);
559             require(isSolvent, 'remaining amount exceeds collateral factor');
560         }
561     }
```

^snippet of “function withdraw()” at MasterPlatypus

While in “function withdraw()”, the logic first withdraw the LP tokens for the user (Line 555), only after the withdrawal will the logic check for the user’s solvency (Line 558), because the LP tokens have already been transferred away, there are no LP tokens deposited, without collateral posted the user is insolvent, which “isSolvent” will be “false”, and will not pass the “require” checking (Line 559). Therefore, the transaction will revert (reverse) and throw an error. In the EVM’s case, any function calls that cannot be fully executed, will result in an error which the whole operation will be reverted, therefore nothing will happen as if the user did not withdraw any LP tokens. There is no possibility for a partial execution where a user calls “function withdraw()” and receives the LP tokens successfully while failing the solvency check implemented.

Recall the user flow to obtain USP is: Deposit USDC -> Receive USDC-LP -> Deposit USDC-LP -> Receive USP.
An example exploit on the platform could be through the following steps (assume no interest payment for the flash loan):

1. Flash Loan borrow $10 million USDC
2. Deposit $10 million USDC to the platform
3. Receive $10 million worth of USDC-LP from the platform
4. Deposit $10 million worth of USDC-LP to the platform
5. Mint $9.5 million worth of USP (with an example LTV of 95%)
6. Call “function emergencyWithdraw()” to withdraw $10 million worth of USDC-LP
7. Repay $10 million worth of USDC-LP
8. Receive $10 million USDC
9. Repay flash loan of $10 million USDC

The person would end up with a balance of 9.5 million USP with no collateral backing locked in the platform.

To perform all the above steps in three transactions, The attacker created their own smart contract logic to interact with the platform’s smart contracts, and utilized Aave’s flash loan feature to maximize the efficiency of the exploit where the platform’s pools can be drained in a single transaction.

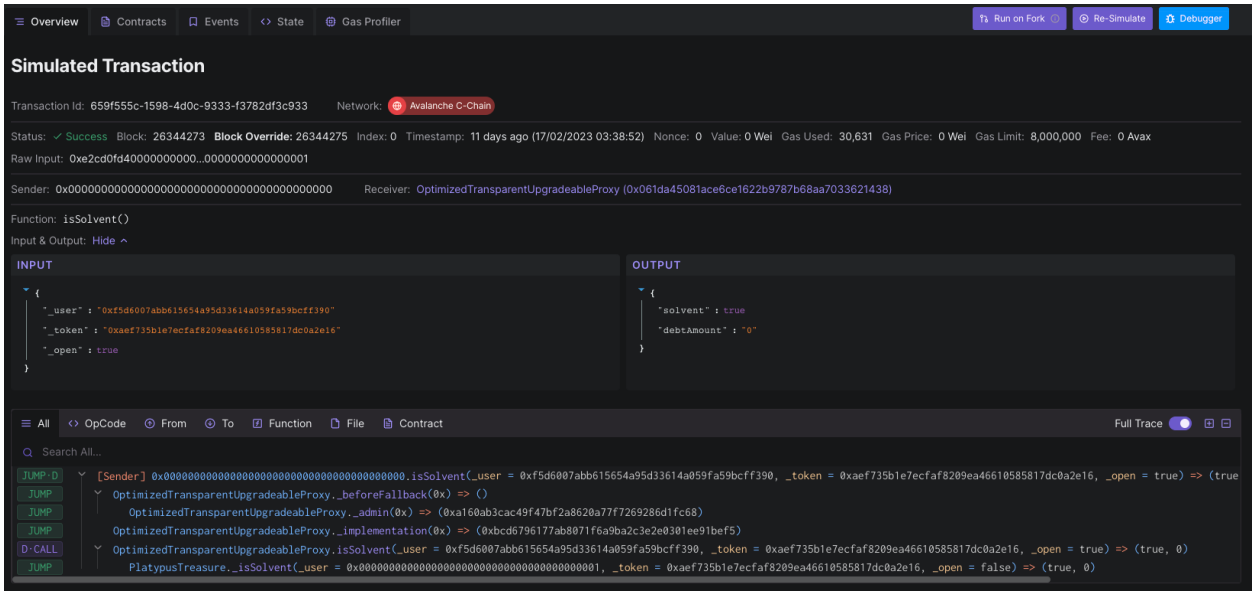
The attacker was able to repay the interest payments accrued from flash loaning USDC through exchanging the maliciously obtained USP to USDC in a liquidity pool (AMM) operated by the platforms’ liquidity providers. USP were also subsequently exchanged by the attacker to various stablecoins such as USDC, USDT, etc on the platform.

Below the exact step by step play by the attacker in the exploit transaction where funds were sent to Aave’s Pool contract:

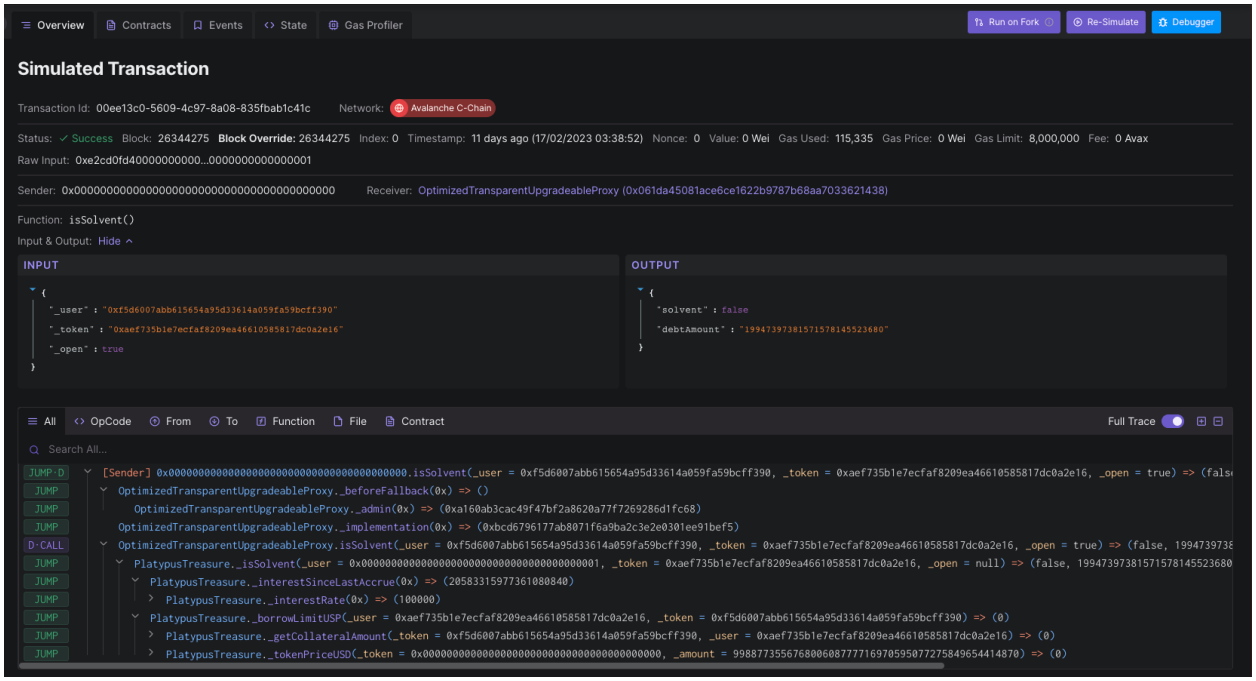
1. Flash Loan borrow $21 million USDC
2. Deposit $21 million USDC to the platform
3. Receive $21 million worth of USDC-LP from the platform
4. Deposit $21 million worth of USDC-LP to the platform
5. Mint ~19.95 million USP
6. Call “function emergencyWithdraw()” to withdraw $21 million worth of USDC-LP
7. Repay $21 million worth of USDC-LP
8. Receive ~$20.95 million USDC
9. Swap 19.95 million USP to various stablecoin assets in the Main Pool of the platform
10. Transfer all stablecoin assets swapped with USP plus unexchanged USP to Aave’s Pool contract, minus a small amount of USDC for flash loan interest payment
11. Repay flash loan of ~$21.01 million USDC (borrowed + interest payment)

We have also conducted a simulation to show that the attacker's contract manages to bypass the normal solvency check flow in "function withdraw()" through the exploit transaction, and created a USP borrowing position with zero collateral posted, which was not what the platform intended to allow.

At block number 26344273, which is one block **before** the exploit, a simulation on Tenderly calling "isSolvent()" at "PlatypusTreasure", the contract responsible for the USP over-collateralized borrowing system, shows that the attacker's contract is **solvent** with a debt amount of 0 USP:



At block number 26344275, which is one block **after** the exploit, a simulation on Tenderly calling "function isSolvent()" at "PlatypusTreasure", the contract responsible for the USP over-collateralized borrowing system, shows that the attacker's contract is **not solvent** with a debt amount of 19 million USP, with the collateral amount posted being 0 (shown at the bottom of the screenshot):



The simulation can be reproduced on Tenderly through inputting the block number mentioned above mentioned to call 0x061da45081ace6ce1622b9787b68aa7033621438, the proxy address of PlatypusTreasure, and using the following parameters:

Parameter name (type)	Parameter value	Note
_user (address)	0xf5d6007abb615654a95d33614a059fa59bcff390	The address of the depositor (the exploiting contract)
_token (address)	0xaef735b1e7ecfaf8209ea46610585817dc0a2e16	The token address of the collateral (LP-USDC)
_open (bool)	true	The parameter used by “function withdraw()” and “function emergencyWithdraw()” to indicate checking of the borrow limit of the user.

To provide more information and perspectives of the exploit, below are the relevant links to the analysis of the exploit made by independent blockchain security organizations:

Name	Link
BlockSec	https://twitter.com/BlockSecTeam/status/1626429271614038016
BlockSec MetaSleuth	https://twitter.com/MetaSleuth/status/1626427932314054656
SlowMist	https://twitter.com/SlowMist_Team/status/1626536522500702208
PeckShield	https://twitter.com/peckshield/status/1626357011444269057
Omniscia	https://twitter.com/Omniscia_sec/status/1626599363110703104
Omniscia	https://medium.com/@omniscia.io/platypus-finance-incident-post-mortem-7b71a0a47a5e

Note: as the attacker launched a series of three transactions that caused a total of 9 million loss, the analysis listed above might not be directly reporting the exact exploit transaction where ~$380k was exploited and transferred to the Aave’s Pool, but rather with a focus on breaking down how the overall exploit happened.

Our request for Aave's help for asset recovery to our users

Based on the above analysis, we would like to request the Aave community for the courtesy to recover the below quantity of ERC-20 tokens to the Platypus Team's multisig address on Avalanche C-Chain:

Name of the asset	Ticker	Contract address	Quantity
TetherToken	USDt	0x9702230A8Ea53601f5cD2dc00fDBc13d4dF4A8c7	96,810.192138
Tether USD (Bridged)	USDT.e	0xc7198437980c041c805A1EDcbA50c1Ce5db95118	79,815.266923
USD Coin	USDC	0xB97EF9Ef8734C71904D8002F8b6Bc66Dd9c48a6E	83,175.344156
USD Coin (Bridged)	USDC.e	0xA7D7079b0FEaD91F3e65f86E8915Cb59c1a4C664	69,756.91961
Dai Stablecoin (Bridged)	DAI.e	0xd586E7F844cEa2F87f50152665BCbc2C279D8d70	26,540.1977603 8658828526
Binance-Peg BUSD	BUSD	0x9C9e5fD8bbc25984B178FdCE6117Defa39d2db39	24,496.7795526 1821502284
USP Stablecoin	USP	0xdaCDe03d7Ab4D81fEDdc3a20fAA89aBAc9072CE2	19,047,391.0562 95368006201

The amounts could be verified through the token transfer logs below:

Transaction hash	Blockchain Explorer
0x919266aa66d7c9a6af02dead5effc1cc68ab7b87890 b52e5fc1e20a7041aa84d	https://snowtrace.io/tx/0x919266aa66d7c9a6af02dead5effc1c c68ab7b87890b52e5fc1e20a7041aa84d

Platypus Team's multisig address on Avalanche C-Chain is:

Address	Blockchain Explorer
0x068e297e8FF74115C9E1C4b5B83B700FdA5aFdEB	https://snowtrace.io/address/0x068e297e8FF74115C9E1C4 b5B83B700FdA5aFdEB