WHITEPAPER

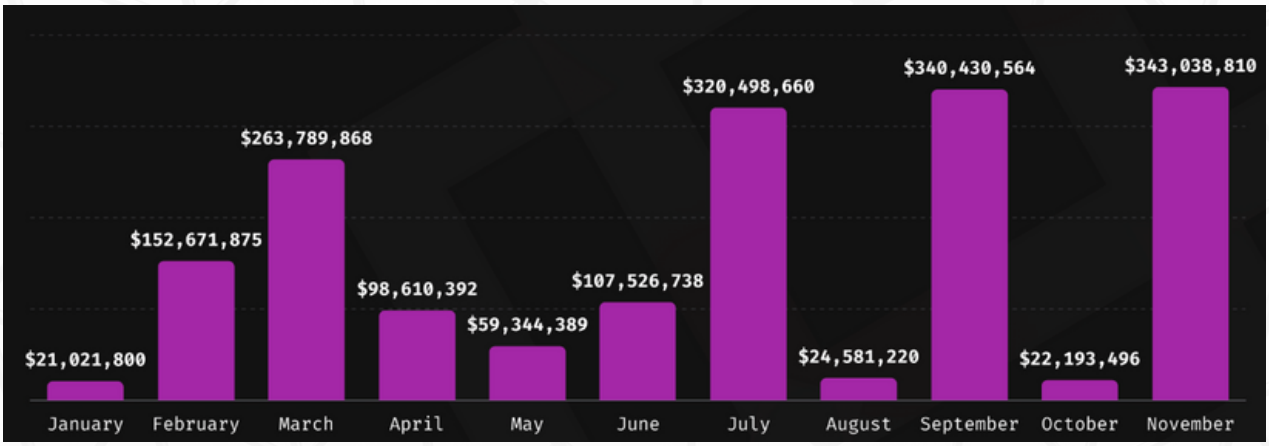
How The Crypto Industry Could Have Saved $1.75B From Hacks This Year

December 2023

Prepared By :
Cyberscope Team

CONTENTS

Introduction

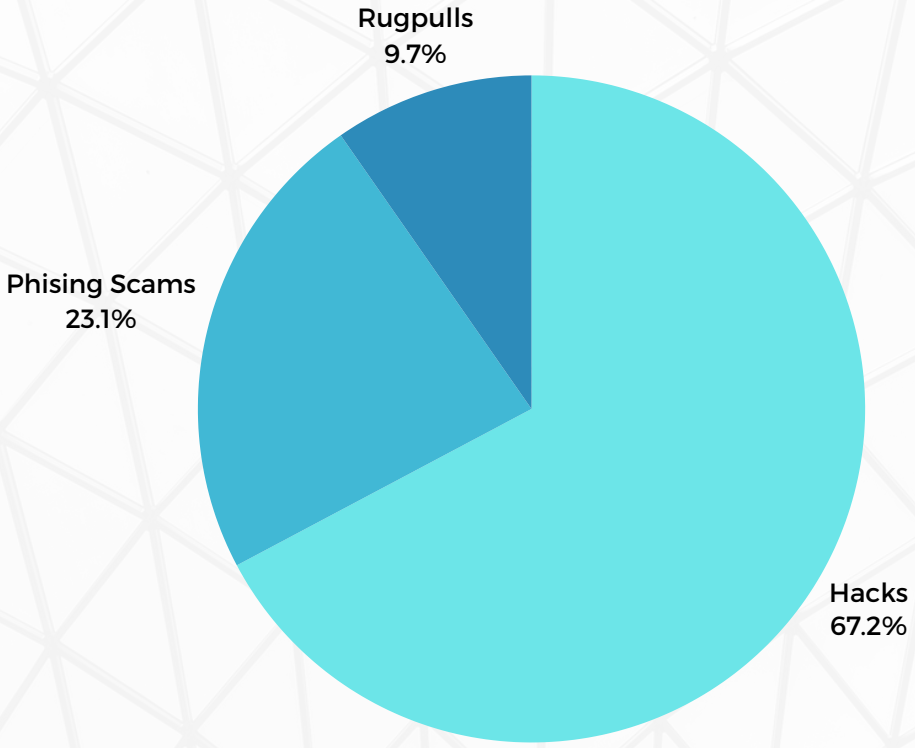


So far in 2023, we have seen more than $1.3B lost from crypto hacks, scams and rug pulls.

Now, as shocking as that number sounds, it actually represents a significant drop from the havoc wreaked in 2022. That year, we were looking at close to $3 billion in crypto loss due to the same types of digital attacks.

So, the good news is that we are moving in the right direction, but we still have a lot of room for improvement.

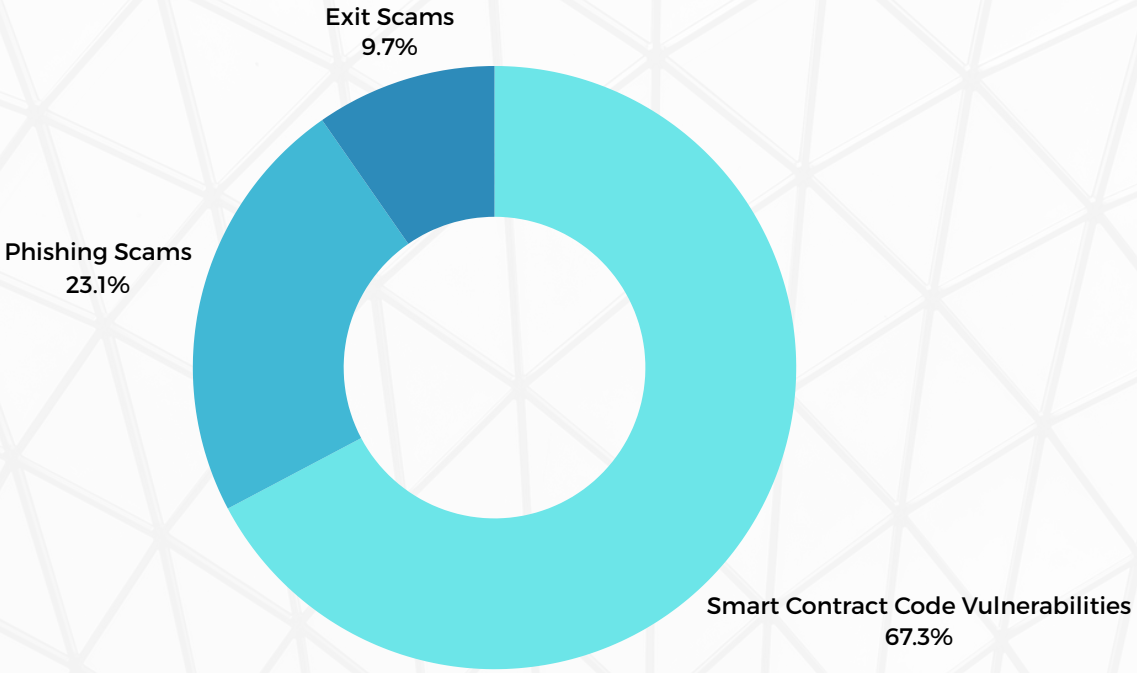
Types of Exploits:



Before we start analysing some of the attacks and scams that contributed to the losses, we want to briefly explain the different types of exploits that we usually see in the space.

There are usually 3 ways we record losses in the crypto industry. Firstly we have smart contract code vulnerabilities. This means that a third party, usually a hacker, finds a vulnerability in a smart contract of a project and exploits it for his gain. Secondly we have exit scams, where the project owners will take the money of investors on false promises and disappear. The most common way we see this happening is through rug pulls, as you probably know. And lastly, we have phishing scams. A phishing scam takes place when someone pretends to be someone else and tricks the users to share private information with the attacker.

Biggest Exploits in 2023



Now, as we mentioned earlier $1.3 Billion were lost in these types of attacks, more specifically here's the breakdown:

- $910M were lost to smart contract code vulnerabilities
- $312M were lost because of phishing scams and
- $131M were due to exit scams

As you can see a staggering 67% of the total losses are due to vulnerabilities.

Now, let's look at some of these attacks more closely

Hack 1: Euler Finance

$197,000,000+
Total Assets Lost

As you might have probably heard already, on March 13 Euler Finance was exploited for around $197 millions.

This was the biggest attack of the year and the 6th highest of all time.

The attacker engaged in a series of flash loan attacks that drained various tokens from the protocol. For those not familiar with the term, a flash loan is a type of loan where a user borrows assets with no upfront collateral and returns the borrowed assets within the same blockchain transaction.

Hack 1: Euler Finance

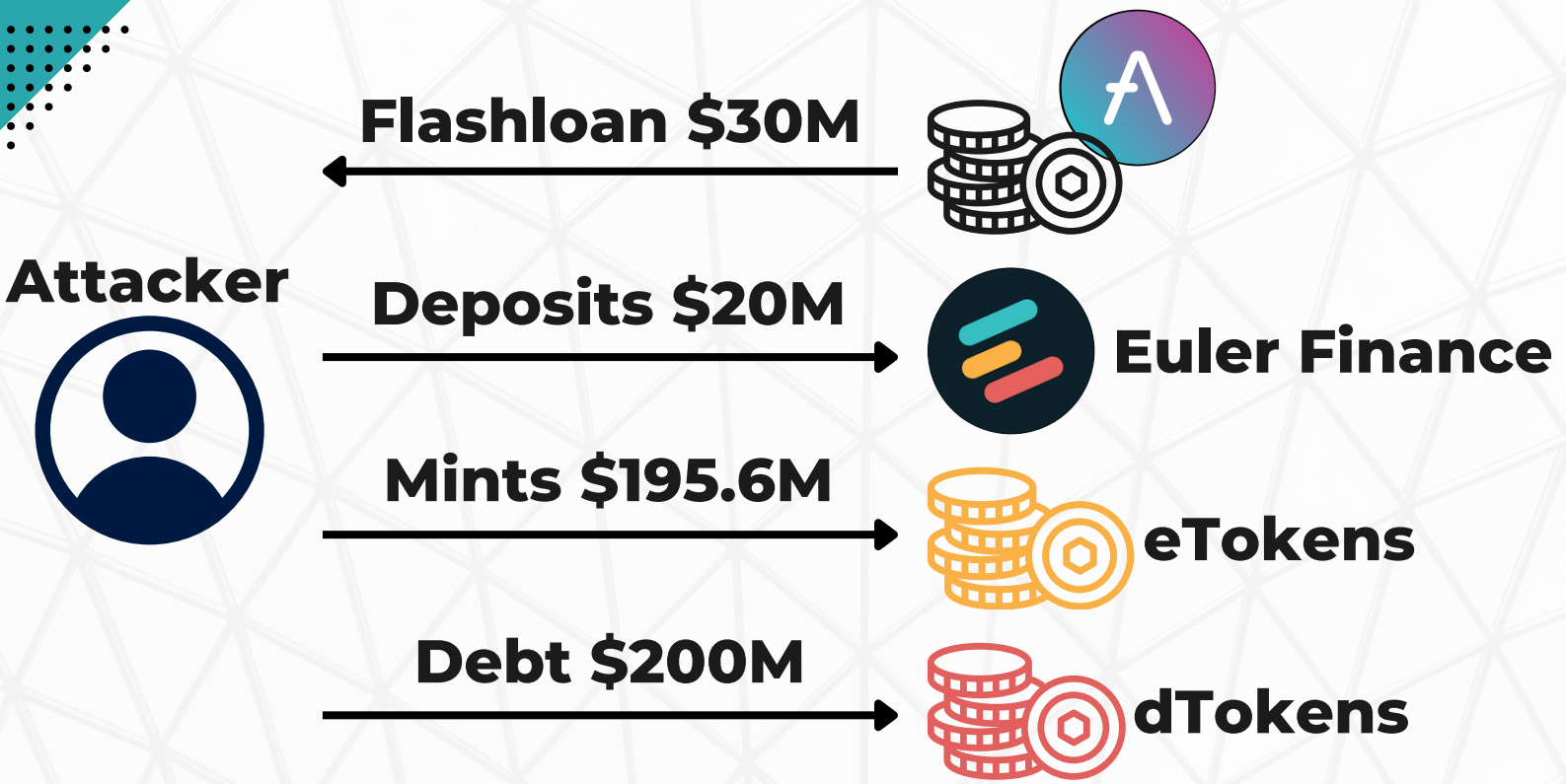
What you also need to know is that Euler is a lending platform similar to Compound or Aave.

Users can deposit crypto and allow the protocol to lend it to others, or they can use a deposit as collateral to borrow crypto. Euler also allows users to get debt in the form of tokens, so they can perform leveraged trades.

So a user can mint "eTokens" to perform trades and will get "dTokens" as a form of debt.

If users acquire more debt than collateral, then they get "liquidated". This means that liquidator bots can take their debt tokens to balance their account.

Hack 1: Euler Finance



How the attack happened:

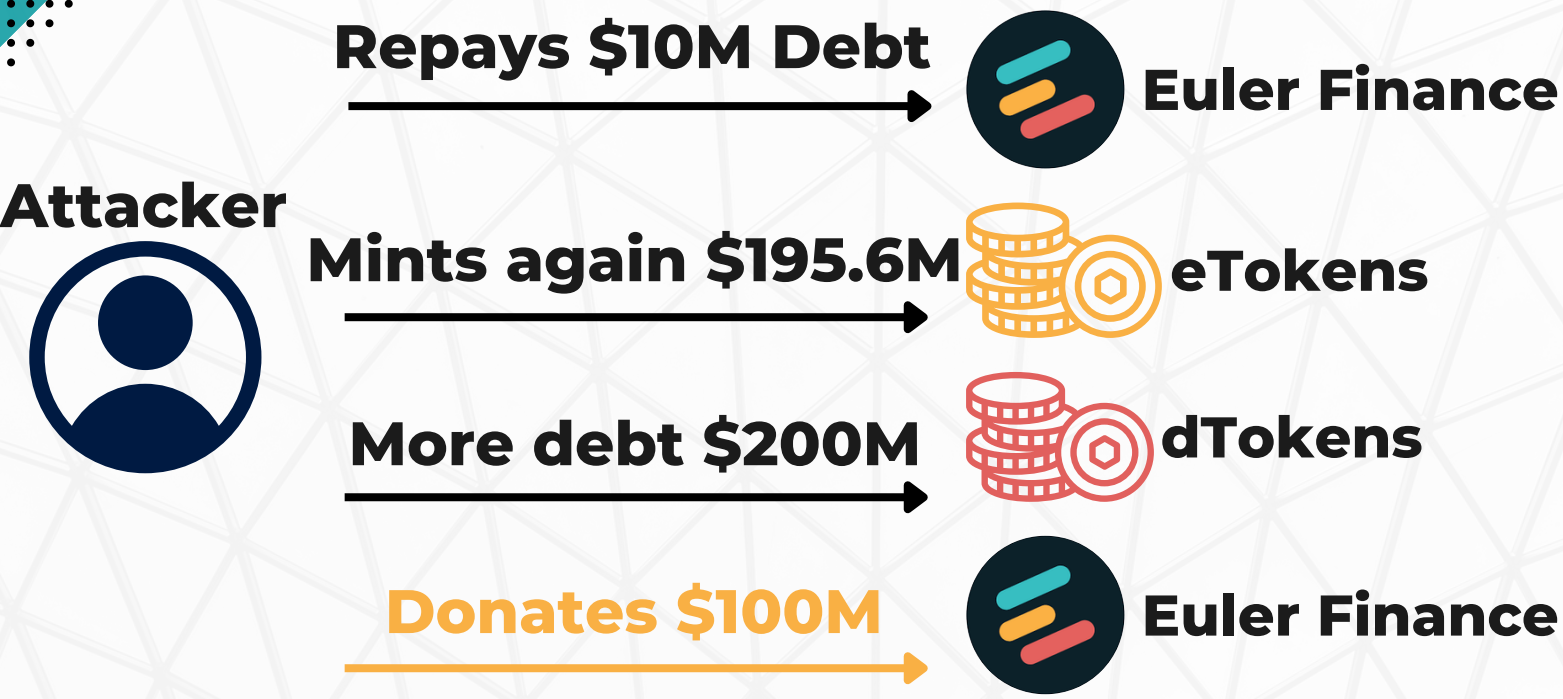
The attacker used 3 accounts to perform the attack.
A smart contract to get a loan from AAVE (audaciously labeled "Euler Exploit Contract"), an account that interacted with Euler and a liquidator bot contract. They got a flash loan of 30 million DAI from Aave, using the exploiter contract.

The hacker deposited 20 million DAI to Euler Protocol and got approximately 19.6 million eDAI in return. Since Euler has some fees, you always get slightly fewer tokens than the ones you deposit on the platform.

Then they leveraged the 19.6 million eDAI to borrow approximately 195.6 million eDAI and 200 million dDAI. (Remember the eTokens and dTokens that are part of the Euler ecosystem.)

At this point, the hacker still has 10 million DAI left out of the 30 million DAI they initially borrowed from AAVE.

Hack 1: Euler Finance



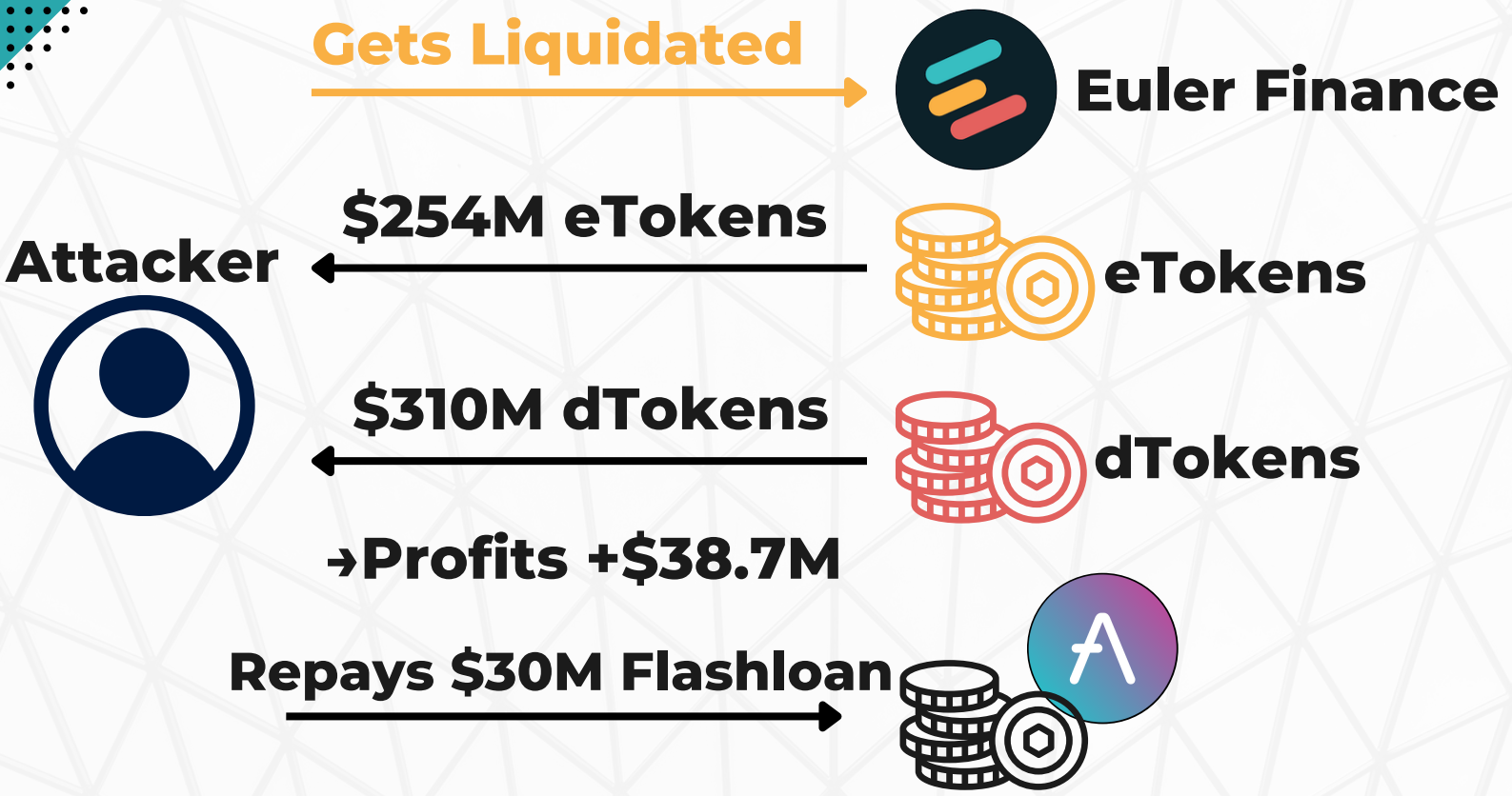
They then used the remaining 10 million DAI to repay some of their debt.

They did this because the Euler Finance smart contract checks how much debt a user can have at any point before they get liquidated.

Balance is now 190 million dDAI. After that, they borrowed another 195.6 million eDAI and 200 million dDAI.

So far, so good. But at this point, the hacker donated 100 million eDAI to the Euler protocol reserve.

Hack 1: Euler Finance



At this moment the liquidation happens (with the attackers liquidator bot).

Since they have donated 100M eDAI, the liquidator got 254 million dDAI and 310 million eDAI.

At this point the liquidator redeems the 38.7 million profit in Euler. The attacker then repaid Aave its 30 million DAI and made about 8.7 million DAI from the exploit.

Hack 1: Euler Finance

Attacker

➜ **First Attack Profit +8.7M**

➜ **Repeats 6 times**

➜ **Total Profit $197M**

The same attack was repeated 6 times on different tokens amounting to the $197 million total losses for Euler.

So, what went wrong here?

Hack 1: Euler Finance

```
function donateToReserves(uint subAccountId, uint amount) external nonReentrant {
    (address underlying, AssetStorage storage assetStorage, address proxyAddr, address msgSender) = CALLER();
    address account = getSubAccount(msgSender, subAccountId);

    updateAverageLiquidity(account);
    emit RequestDonate(account, amount);

    AssetCache memory assetCache = loadAssetCache(underlying, assetStorage);

    uint origBalance = assetStorage.users[account].balance;
    uint newBalance;

    if (amount == type(uint).max) {
        amount = origBalance;
        newBalance = 0;
    } else {
        require(origBalance >= amount, "e/insufficient-balance");
        unchecked { newBalance = origBalance - amount; }
    }

    assetStorage.users[account].balance = encodeAmount(newBalance);
    assetStorage.reserveBalance = assetCache.reserveBalance = encodeSmallAmount(assetCache.reserveBalance + amount);

    emit Withdraw(assetCache.underlying, account, amount);
    emitViaProxy_Transfer(proxyAddr, account, address(0), amount);

    logAssetStatus(assetCache);
}
```

At one point the attacker donated some of their eTokens to Euler. The vulnerability that was exploited by the hacker was the lack of liquidity checks in the donate function.

This allowed the attacker to donate funds with a lot of bad debt and profit massively from the trade.

Hack 1: Euler Finance

```
.test({
    desc: "donate to reserves - basic",
    actions: ctx => [
        { send: 'eTokens.eTST.deposit', args: [0, et.eth(10)] },
        { call: 'eTokens.eTST.totalSupply', equals: [et.eth(10), '0.000000001' ], onResult: r => {
            ctx.stash.ts = r;
        } },

        { send: 'eTokens.eTST.donateToReserves', args: [0, et.eth(1)], onLogs: logs => {
            et.expect(logs.length).to.equal(4);

            et.expect(logs[0].name).to.equal('RequestDonate');
            et.expect(logs[0].args.account).to.equal(ctx.wallet.address);
            et.expect(logs[0].args.amount).to.equal(et.eth(1));

            et.expect(logs[1].name).to.equal('Withdraw');
            et.expect(logs[1].args.underlying).to.equal(ctx.contracts.tokens.TST.address);
            et.expect(logs[1].args.account).to.equal(ctx.wallet.address);
            et.expect(logs[1].args.amount).to.equal(et.eth(1));

            et.expect(logs[2].name).to.equal('Transfer');
            et.expect(logs[2].args.from).to.equal(ctx.wallet.address);
            et.expect(logs[2].args.to).to.equal(et.AddressZero);
            et.expect(logs[2].args.value).to.equal(et.eth(1));
        } },

        { call: 'eTokens.eTST.totalSupply', equals: () => ctx.stash.ts },
        { call: 'eTokens.eTST.reserveBalance', equals: et.eth(1).add(et.DefaultReserve) },
        { call: 'eTokens.eTST.balanceOf', args: [ctx.wallet.address], equals: [et.eth(9), '0.000000001'] },
        { call: 'tokens.TST.balanceOf', args: [ctx.contracts.euler.address], equals: et.eth(10), },
    ],
})
```

How could the attack have been prevented?

Test, test and test again.

Looking closely at the test scenarios on Euler's GitHub, there weren't any tests for a donating after borrowing scenario.

The team could have mitigated this attack if they had tested the vulnerable function against every possible scenario.

It's worth noting that Euler Finance had worked with 6 different audit firms, although not all of them had reviewed all the contacts.

It's better to work closely and in-depth with cybersecurity companies and not leave any functions or contracts out of the audit scope.

The good news of this story is that hacker eventually returned most of the stolen funds.

Hack 2: Curve Finance

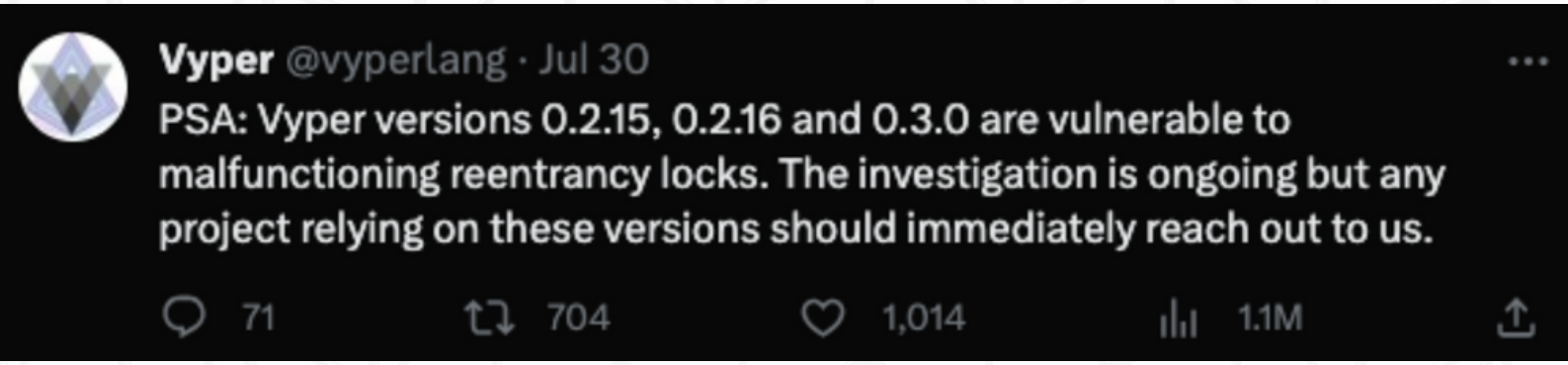
$70,000,000+
Total Assets Lost

Moving on, to the second biggest attack of the year. The $70 million attack on Curve Finance.

On July 30, 2023, several liquidity pools on Curve Finance and other DeFI protocols were exploited due to a vulnerability on Vyper.

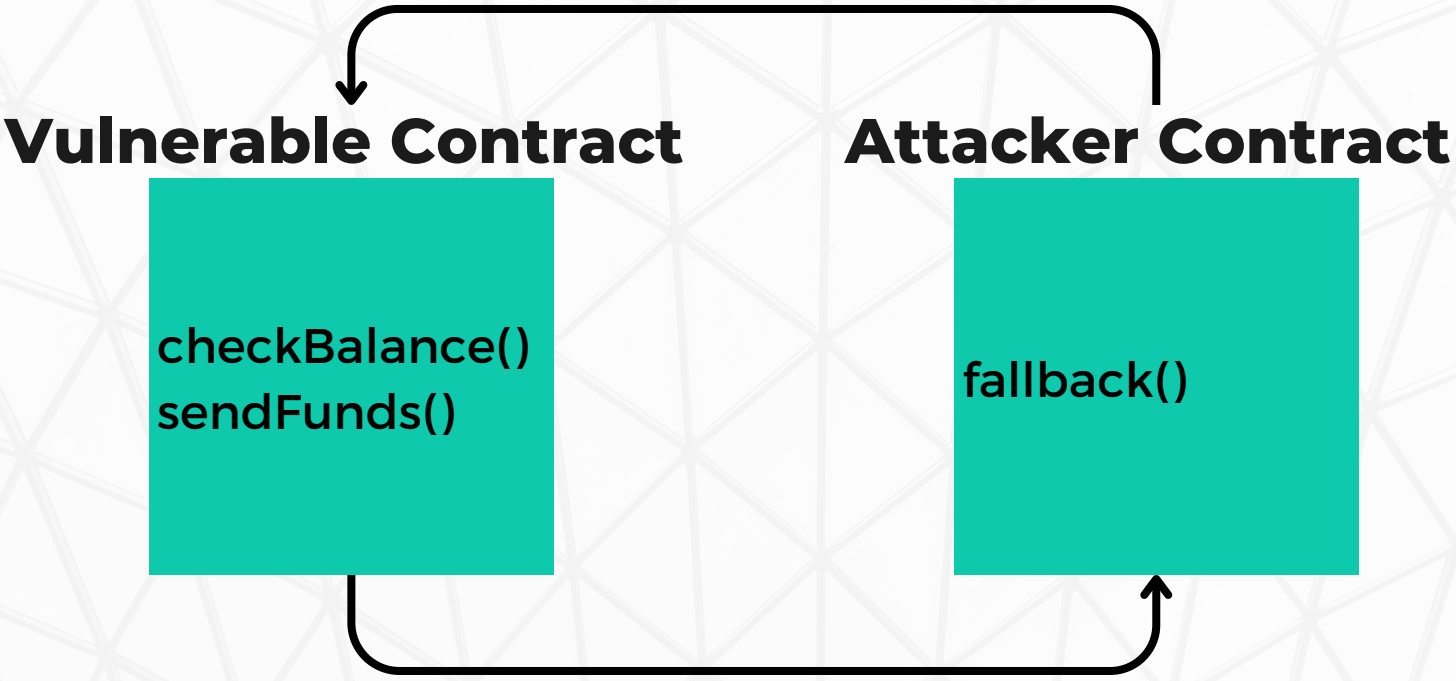
Vyper is a third-party Pythonic programming language for Ethereum smart contracts used by Curve and other decentralized protocols.

Hack 2: Curve Finance



According to Vyper, its 0.2.15, 0.2.16, and 0.3.0 versions contained issues making some smart contracts vulnerable to re-entrancy attacks, in which attackers can trick the contracts into incorrectly calculating balances, allowing them to steal funds held by the contracts' protocols.

Hack 2: Curve Finance



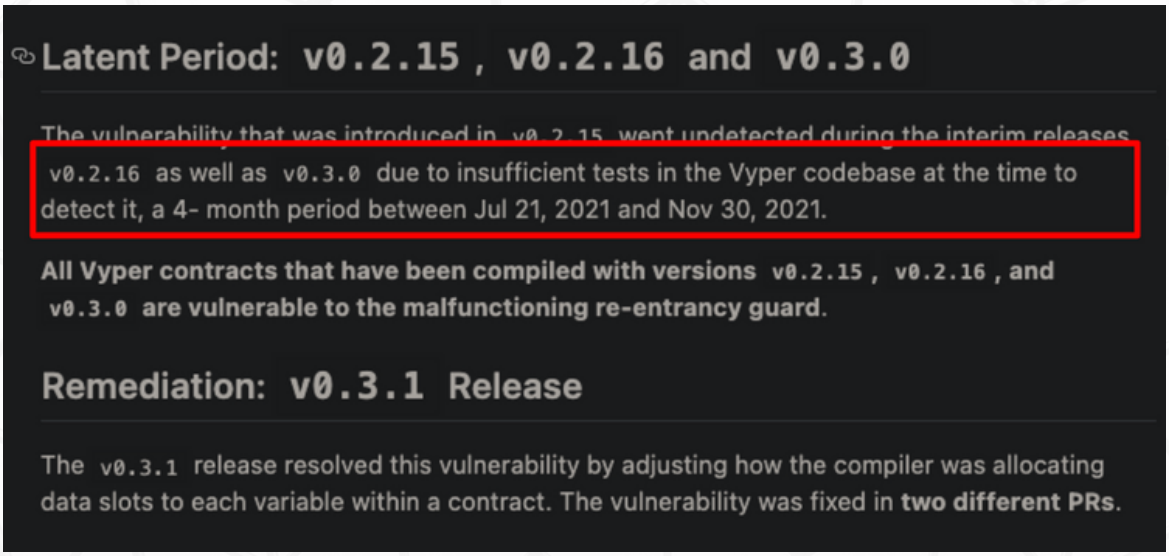
Re-entrancy attacks are some of the most common in Solidity contracts.

A reentrancy attack occurs when a function makes an external call to another untrusted contract.

Then the untrusted contract makes a recursive call back to the original function in an attempt to drain funds.

When the contract fails to update its state before sending funds, the attacker can continuously call the withdraw function to drain the contract's funds.

Hack 2: Curve Finance



What went wrong in this case?

Vyper released a detailed report on how they missed this bug in production and what they are planning to do moving forward.

According to the report:

"due to insufficient tests in the Vyper codebase at the time to detect it, a 4-month period between Jul 21, 2021 and Nov 30, 2021."

Again, we are seeing the same patterns where more tests could have prevented the vulnerability.

Keep in mind that, all contracts compiled with the affected versions of Vyper are still vulnerable to the malfunctioning re-entrancy guard. Vyper has also promised to collaborate with multiple audit firms moving forward to review past and future versions of the language

How could the attack have been prevented?

When working with third-party software, always make sure you use the latest version of the software.

Vyper introduced this bug a year ago and has since released multiple more stable versions of its language.

Re-Entrancy Attacks

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.9.0;
/// @title A contract for demonstrate payable functions and addresses
/// @author Jitendra Kumar
/// @notice For now, this contract just show how payable functions and
contract Reentrancy {
    mapping(address => uint) public balance;

    function deposit() public payable {
        balance[msg.sender] += msg.value;
    }

    function withdraw() public payable {
        require(balance[msg.sender] >= msg.value);
        payable(msg.sender).transfer(msg.value);
        balance[msg.sender] -= msg.value;
    }
}
```

How can you prevent re-entrancy attacks?

Since re-entrancy attacks have been used in multiple hacks in the past it's worth mentioning how you can protect yourself against them.

Though many smart contract platforms and development frameworks have built-in protections against reentrancy, it still remains an ongoing threat to smart contract security.

Above is an example of a simple smart contract that is vulnerable to reentrancy attacks:

- In this contract, the deposit function allows users to deposit funds into their account and the withdraw function allows users to withdraw their deposited funds.
- However, the contract does not properly check for reentrancy, so an attacker could create a malicious contract that repeatedly calls the deposit function before calling the withdraw function, effectively stealing funds from the contract.

Re-Entrancy Attacks

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.9.0;
/// @title A contract for demonstrate Reentrancy Attack
/// @author Jitendra Kumar
/// @notice For now, this contract just show how to protect Smart Cont
contract Reentrancy {

    mapping(address => uint) public balance;
    bool public reentrancyLock;

    function deposit() public payable {
        require(!reentrancyLock);
        reentrancyLock = true;
        balance[msg.sender] += msg.value;
        reentrancyLock = false;
    }

    function withdraw() public payable{
        require(balance[msg.sender] >= msg.value);
        payable(msg.sender).transfer(msg.value);
        balance[msg.sender] -= msg.value;
    }
}
```

Here is an example of how the contract can be modified to prevent reentrancy attacks:

- In this example, a reentrancyLock variable is added to the contract.
- The deposit function now checks if the reentrancyLock variable is true before allowing the deposit to occur.
- If the variable is true, the deposit will not occur and the attacker will not be able to steal funds.
- The reentrancyLock variable is set to true before the deposit and set back to false after, this way the contract can only be called once at a time.

This is a very simple example of how to think and protect your functions against re-entrancy attacks.

Re-Entrancy Attacks - How to Prevent

- Check Effects Interactions Pattern (CEI)

- Use a Mutex or Mutual Exclusion Lock

There are several other ways you can make sure your contracts are safe from re-entrancy attacks:

1. Checks effects interactions pattern (CEI)
The first step in using this pattern is to perform some checks and verifications in the contract flow.
Effects/changes in the state variables of the current contract should be carried out before any interactions with another contract.

2. Use a Mutex or Mutual Exclusion Lock
Like in our example earlier, a mutex lock is used to prevent multiple calls to the same function from occurring at the same time.
When a function is called, the mutex lock is set, and other calls to the same function will be blocked until the lock is released.

It's important to note that these are just some examples of the ways to protect smart contracts against a reentrancy attack and that smart contracts must be audited by experts and tested extensively to ensure they are secure.

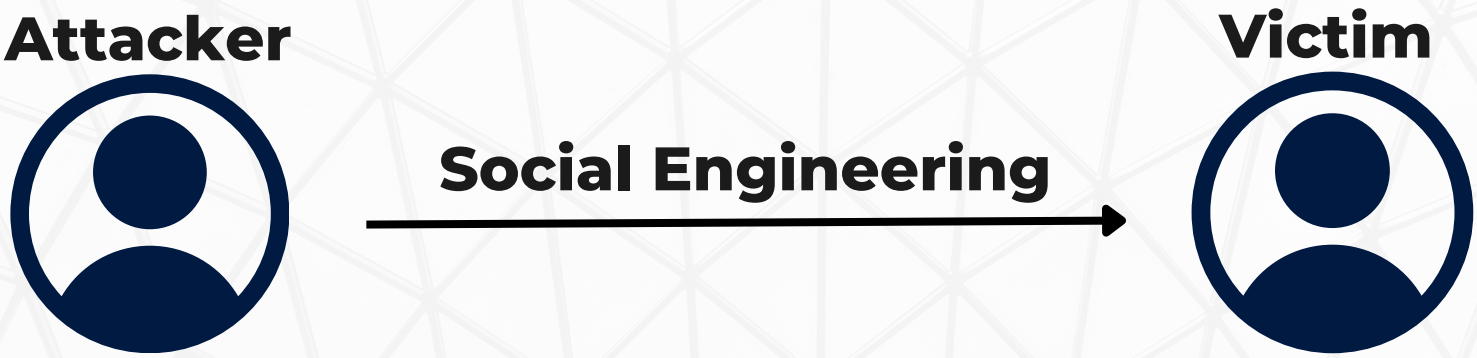
Hack 3: CoinsPaid

$37,000,000+
Total Assets Lost

One of the most noteworthy hack of this year happened on July 22nd on one of the world's biggest crypto payments providers.

The hack is attributed to the notorious Lazarus group which is believed to have been targeting CoinsPaid for about 6 months before the successful attack.

Hack 3: CoinsPaid



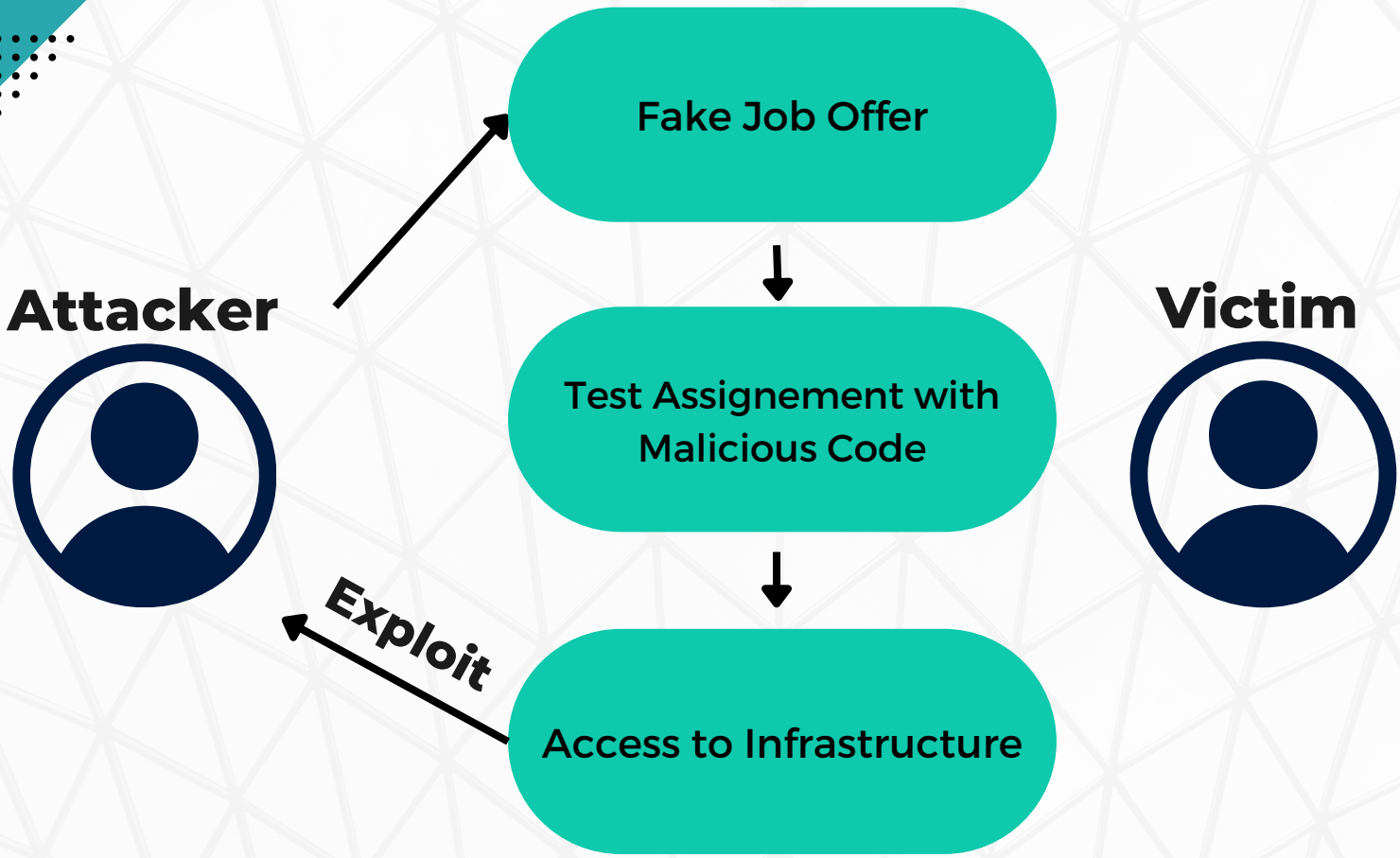
The hack was a result of a successful social engineering attack.

The perpetrators' main goal was to trick a critical employee into installing software, to gain remote control of a computer for the purpose of infiltrating and accessing CoinsPaid's internal systems.

Recruiters from crypto companies contacted CoinsPaid employees via LinkedIn and various Messengers, offering very high salaries.

For instance, some of our team members received job offers with compensation ranging from 16,000-24,000 USD a month.

Hack 3: CoinsPaid



The hack was a result of a successful social engineering attack.

The perpetrators' main goal was to trick a critical employee into installing software, to gain remote control of a computer for the purpose of infiltrating and accessing CoinsPaid's internal systems.

Recruiters from crypto companies contacted CoinsPaid employees via LinkedIn and various Messengers, offering very high salaries.

For instance, some of our team members received job offers with compensation ranging from 16,000-24,000 USD a month.

Hack 3: CoinsPaid

How could the attack have been prevented?

- Employee Training

- Access Privileges

- Monitoring Systems

How could the attack have been prevented?

While CoinsPaid had invested significantly in its cybersecurity infrastructure, all it took was an employee to make a small mistake

- It is imperative for all companies to explain to their employees how perpetrators can use fake job offers, bribing, and even ask for harmless tech advice to access the company's infrastructure.
- Also, it's very important that you should implement strict security practices for privileged users.
- The hack could have also been caught early if the right monitoring systems were in place.
- Keeping track of the operating balances of the company and monitoring their unusual movement and behaviour, could have triggered alerts before the whole balance was drained

Working closely with a cybersecurity partner can help you set up all the preventative systems to avoid such incidents in the future.

Hack 4: Jimbo’s Protocol

$7,500,000+
Total Assets Lost

Another headliner this year, was the Jimbos Protocol exploit for $7.5M.

On 28 May 2023, the Arbitrum-based protocol project fell victim to a Flash Loan Attack.

The Jimbo's protocol hack was made possible by a lack of slippage control in the project’s smart contract

Hack 4: Jimbo's Protocol

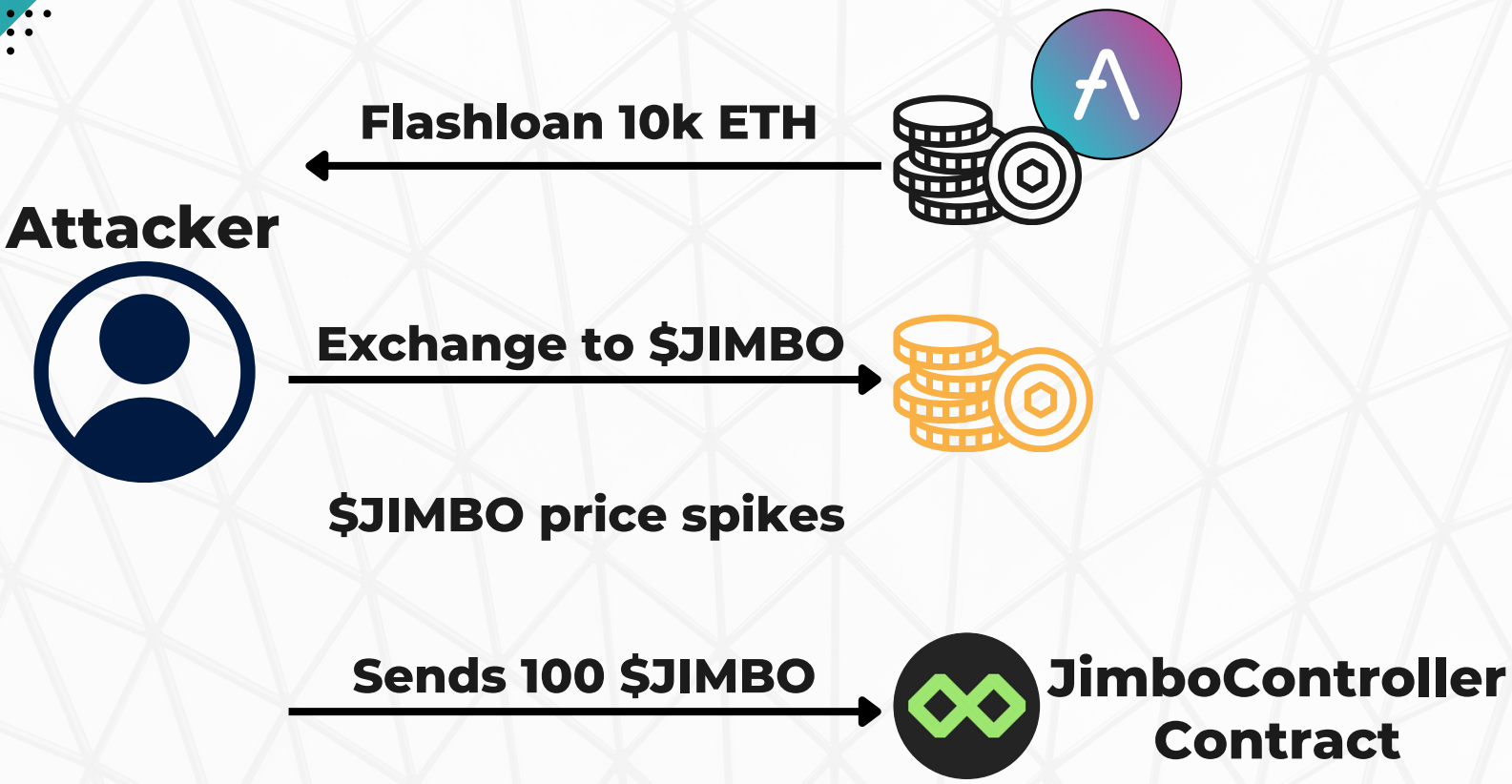
```
function shift() public returns (bool) {
    if (canShift()) {
        // Let the token know the protocol is rebalancing
        jimbo.setIsRebalancing(true);

        // Get the active bin
        uint24 activeBin = pair.getActiveId();

        // Remove all non-floor bin liquidity (max bin -> anchor bin)
        _removeNonFloorLiquidity();

        // Remove all floor bin liquidity
        _removeFloorLiquidity();

        // Count the total JIMBO and ETH in the contract after liquidity removal
        uint256 totalJimboInPool = jimbo.balanceOf(address(this));
        uint256 totalEthInContract = weth.balanceOf(address(this));

        // Floor is based on total eth / circulating supply
        uint256 totalCirculatingJimbo = jimbo.totalSupply() -
        // Calculate the new target floor bin
        uint24 newFloorBin = _calculateNewFloorBin(
        );

        // Calculate new anchor bin id
        // Make sure you use the new floor bin and not the stale one
        uint24 newAnchorBin = activeBin - newFloorBin > NUM_ANCHOR_BINS
        // Set internal bin state
        _setBinState({
        });

        // Deploy all the JIMBO liquidity first
        _deployJimboLiquidity();

        // Deploy floor bin liquidity with 90% of all ETH in the contract
        _deployFloorLiquidity((weth.balanceOf(address(this)) * 90) / 100);

        // Use entire remaining weth balance in the contract to deploy anchors
        _deployAnchorLiquidity(weth.balanceOf(address(this)));

        // Let the token know we are done rebalancing to apply taxes
        jimbo.setIsRebalancing(false);
        return true;
    } else return false;
```

For those not familiar with the term slippage, it's what happens when there is a difference between the price the investor is expecting to pay in a trade, and the actual price when the trade is executed.

Hence, slippage control is a mechanism in smart contracts that prevents large trades from causing significant price fluctuations.

In the case of the Jimbo protocol, the lack of slippage control in the shift() function of the JimboController contract allowed the hacker to exploit the vulnerability.

Hack 4: Jimbo's Protocol



Here's how the attack happened:

- The attacker borrowed around 10,000 ETH from AAVE
- Then they exchanged the borrowed ETH for a substantial amount of Jimbo funds through the [ETH-Jimbo] trading pair
- This caused a surge in the current price of Jimbo.

After that, the attacker transferred 100 JIMBO tokens to the JimboController contract.

Hack 4: Jimbo's Protocol

By invoking the JimboController's shift() function (the one without the slippage control) manipulated the token balance in the liquidity pool.

Following the manipulation, the attacker converted the acquired Jimbo tokens back into ETH and repaid the flash loan, thereby exiting the exploit with substantial profits.

Hack 4: Jimbo’s Protocol

The attack, of course, caused the price of $JIMBO to crash.

What went wrong?

- Jimbos Protocol was a fairly new DeFi to the space, launched only 20 days before the attack.
- It aimed to address liquidity and volatile token prices through a new testing approach.
- However, the protocol’s mechanism was not adequately developed and tested leading to a logical vulnerability creating favorable conditions for attackers.

Hack 4: Jimbo’s Protocol

How could the attack have been prevented?

- Test, Test & Test Again

- Smart Contract Audits

How could the attack have been prevented?

- As a liquidity protocol, the Jimbo's Protocol should have incorporated slippage protection and tested various scenarios on how different trades could have affected the prices.
- It's worth also noting that this smart contract was not audited by any cybersecurity firms.

If Jimbos Protocol had invested in a thorough smart contract audit, the hack could have possibly been prevented.

Hack 5: Deus Finance

$6,000,000+
Total Assets Lost

Let's analyze another DeFi attack, this time on Deus Finance.
This attack resulted in a $6M loss across the BSC, ARB, and ETH networks.

In this instance, the attack was made possible due to a faulty smart contract implementation.

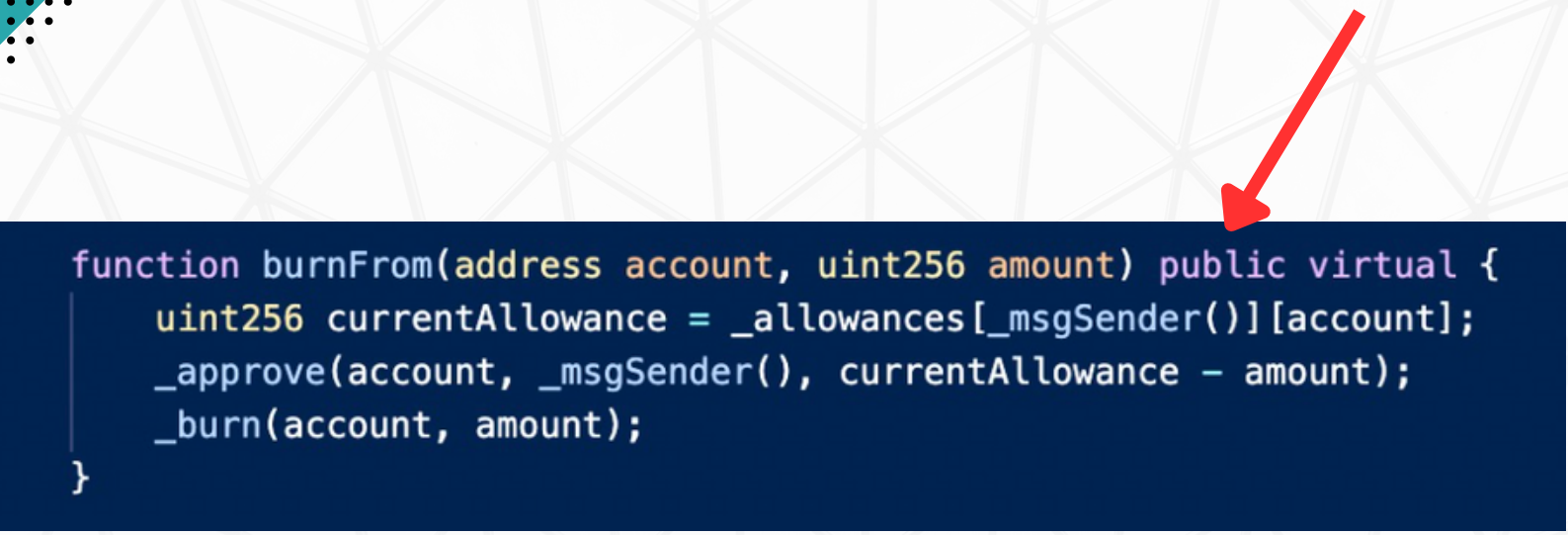
Hack 5: Deus Finance

```
function burnFrom(address account, uint256 amount) public virtual {
    uint256 currentAllowance = _allowances[_msgSender()][account];
    _approve(account, _msgSender(), currentAllowance - amount);
    _burn(account, amount);
}
```

Deus Finance's smart contract had a public burn that allowed the hacker to exploit it.

Here's how the attack happened:
The attacker did some research and identified an address with a huge amount of DEI. DEI was the stablecoin of the Deus Finance protocol

Hack 5: Deus Finance



```
function burnFrom(address account, uint256 amount) public virtual {
    uint256 currentAllowance = _allowances[_msgSender()][account];
    _approve(account, _msgSender(), currentAllowance - amount);
    _burn(account, amount);
}
```

The burn function that was public, also allowed the user to approve a considerable token allowance for that address.

Then the attacker manipulated the amount to 0 and specified the targeted address.

This allowed them to change the approval to their address and drain the funds from the DEI holder's account.

In the end, they utilized the "transferFrom" function to successfully transfer the stolen funds to their address, effectively completing the attack.

This attack was performed on 3 different networks amounting to total losses of $6M.

Hack 5: Deus Finance

What went wrong?

- New Contracts Not Audited

- Public Function

What went wrong?

Deus Finance had recently upgraded their smart contracts prior to the attack and they didn't get them audited before deploying them.

Public functions need a lot of caution in Solidity as they can be used by anyone on the blockchain.

Hack 5: Deus Finance

How could the attack have been prevented?

- Test, Test & Test Again

- Smart Contract Audits

How could the attack have been prevented?

This is not the first time a hack has happened due to a public function being exploited.

We've seen this many times. When a burn function is public an attacker can exploit it in various ways.

It's one of the many things auditors have on their checklist to review when auditing.

A proper smart contract audit would have caught this vulnerability and could have prevented the hack.

Conclusion

Our whitepaper, "How The Crypto Industry Could Have Saved $1B From Hacks This Year," unveils crucial insights. Based on our experience, 80% of previously audited smart contracts reveal vulnerabilities.

Collaboration with auditors is key. Multiple perspectives ensure comprehensive security, fortifying your project against potential threats.

Our advice: rigorously test your smart contract for all scenarios.

Ready to safeguard your project? Reach out to us for a comprehensive smart contract audit. Your success is our priority.

Contact Us

@raf_cyberscope

contact@cyberscope.io