



crypto.com

Forensics of Attacks and Exploits in DeFi

January 2022

Research and Insights



Head of Research and Insights
Henry Hon PhD, CFA

Research Intern
Bowen Liu

RESEARCH DISCLAIMER

This report alone must not be taken as the basis for investment decisions. Users shall assume the entire risk of any use made of it. The information provided is merely complementary and does not constitute an offer, solicitation for the purchase or sale of any financial instruments, inducement, promise, guarantee, warranty, or an official confirmation of any transactions or contract of any kind.

The views expressed herein are based solely on information available publicly, internal data or information from other reliable sources believed to be true. This report includes projections, forecasts and other predictive statements which represent [Crypto.com](https://crypto.com)'s assumptions and expectations in the light of currently available information. Such projections and forecasts are made based on industry trends, circumstances and factors involving risks, variables and uncertainties. Opinions expressed herein are our current opinions as of the date appearing on the report only.

No representations or warranties have been made to the recipients as to the accuracy or completeness of the information, statements, opinions or matters (express or implied) arising out of, contained in or derived from this report or any omission from this document. All liability for any loss or damage of whatsoever kind (whether foreseeable or not) which may arise from any person acting on any information and opinions contained in this report or any information which is made available in connection with any further enquiries, notwithstanding any negligence, default or lack of care, is disclaimed.

This report is not meant for public distribution. Reproduction or dissemination, directly or indirectly, of research data and reports of [Crypto.com](https://crypto.com) in any form, is prohibited except with the written permission of [Crypto.com](https://crypto.com). Persons into whose possession the reports may come are required to observe these restrictions.

Contents

Executive Summary	6
1. Introduction	7
2. Reentrancy Attack	9
2.1 Attack Explained	9
2.2 Case Studies	11
2.3 Mitigation & Prevention	12
3. Phishing Attack	14
3.1 Attack Explained	14
3.2 Case Studies	14
3.3 Variants of Phishing Attacks	16
3.4 Mitigation & Prevention	17
4. Flash Loan Attacks & Price Manipulation	18
4.1 Attack Explained	18
4.2 Case Studies	19
4.3 Mitigation & Prevention	22
5. Rug Pulls	23
5.1 Attack Explained	23
5.2 Case Studies	24
5.3 Mitigation & Prevention	25
6. Code Bugs	26
6.1 Attack Explained	26
6.2 Case Studies	26
6.2.1 Lack of Balance/Address Checking	27
6.2.2 Integer Overflow	28

6.2.3 Multi-sig Incompleteness	28
6.2.4 Re-callable Init()	29
6.2.5 Code Workflow Error	30
7. Poor Access Control	32
7.1 Attack Explained	32
7.2 Case Studies	32
7.3 Mitigation & Prevention	35
8. Compromised Private Key	36
8.1 Attack Explained	36
8.2 Case Studies	37
8.3 Mitigation & Prevention	38
9. Conclusion	39
References	40

Executive Summary

Decentralised finance (DeFi) is a category of blockchain-based solutions that aims to solve the problems of traditional finance. **The DeFi space is rapidly expanding and their TVLs [surpassed over USD 100 billion](#) as of December 2021.**

However, greater usability comes with greater risks and challenges. As of December 2021, there were **[over \\$2.5B funds lost](#)** in DeFi due to targeted attacks and underlying system vulnerabilities.

In this report, we **dispel the mist of infamous security exploits and vulnerabilities** targeted on multiple DeFi protocols in the past two years, **shed light on the postmortem of each attack type**, and present the **potential mitigation strategies**.

The vulnerabilities that we discuss are as follows:

- We dissect how an attacker launches a **Reentrancy Attack** by exploiting vulnerable **fallback functions** on [Grim Finance](#), [Visor Finance](#), [Cream Finance](#), [DForce](#), and [Uniswap](#) protocols.
- **Phishing** is a type of social engineering attack often used to steal user credential data. This report discusses the generic form of phishing attack in addition to **a variant at programming language level**.
- **Arbitrageurs in DeFi usually play Flash Loan Attacks to make a profit** by manipulating the price of particular asset types. We will sketch the workflow of flash loan attacks suffered by victim platforms.
- **Rug pulls** are one of the most common DeFi attacks in which an individual misuses their privileges for draining value from the protocol, which took **[more than \\$2.8 billion worth of cryptocurrency](#)** from victims in 2021.
- Like all computer programs, **it is likely that most DeFi protocols will contain bugs**. In this report, we pick up **five typical yet easily-overlooked code flaws** and present possible mitigations.
- Access control is a traditional security mechanism that regulates and limits the access permission to particular system resources. We present **[the biggest DeFi hack event in 2021 on PolyNetwork](#)** due to **poor access control**.
- In the crypto world, **private keys are crucial to an individual's funds**. This report analyses three private key compromise events on [EasyFi](#), [Nexus Mutual](#), and [Levyathan project](#).

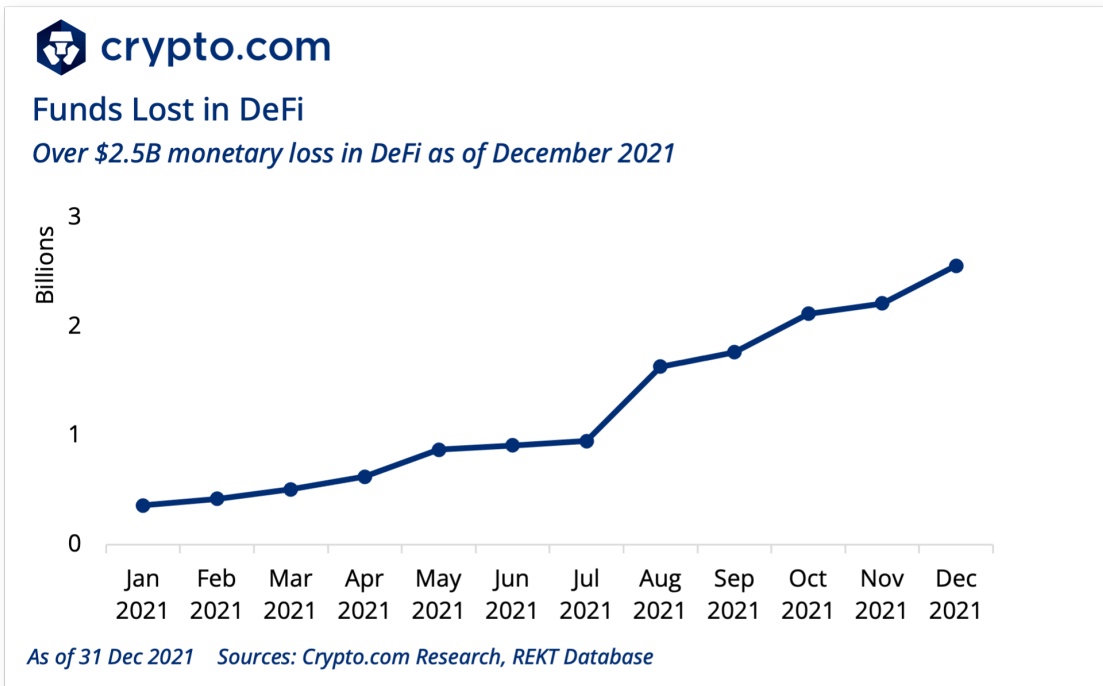
1. Introduction

One promise of open cryptocurrencies is to make payments universally accessible without needing trusted parties. [Decentralised Finance \(DeFi\) aims at extending this promise, proposing novel and traditional financial tools built on the top of a blockchain-based smart contract platform.](#) DeFi offers multiple advantages over traditional finance. First, it inherits the blockchain properties, like decentralisation, openness, accessibility, and censorship-resistance. Second, DeFi is highly flexible, allowing for rapid innovation and experiments by combining, stacking, or connecting different financial instruments. Finally, DeFi provides interoperable services. Generally, new DeFi projects can be built or composed by combining other DeFi platforms. Nowadays, the DeFi space is rapidly expanding including lending & borrowing platforms, DEXs and derivatives, etc. Their TVLs [surpassed over \\$100 billion](#) as of December 2021.

Greater usability and adoption come with greater security concerns. Like many other platforms, the underlying DeFi ecosystems [suffer from a number of security risks.](#) Firstly, most DeFi platforms are deployed over permissionless and transparent smart contract environments. Thus, anyone, including curious attackers and rational arbitrageurs, can inspect and interact with them. Secondly, DeFi projects are always involved in [monetary business logic.](#) Smart contracts determine how units of value convertible to real money move, making them a high-value target with intrinsic economic incentives. Finally, many DeFi platforms [lack fine-grained security protection](#) and regular security audit, bringing into irretrievable financial loss.

According to the REKT database, [over \\$2.5B funds loss](#) happened in the DeFi aspect due to targeted attacks, code flaws, rug pulls, and underlying system vulnerabilities as of December 2021.

In the following sections, we will dispel the mist of infamous attacks and vulnerabilities that happened in the DeFi space over the past two years. **We hope to shed light on the underlying mechanisms of those security exploits. We also provide general mitigation strategies to discuss what lessons we can learn from those attacks.**



2. Reentrancy Attack

2.1 Attack Explained

A so-called reentrancy attack was the essence of the [real-world The DAO attack](#) and led to a [loss of over \\$50m worth of Ether](#) at the time the attack occurred. [Fallback function](#) abuse played a very important role in this reentrancy attack. Let's first see what a fallback function is and how it can be used for malicious purposes.

A smart contract can have one anonymous function, known as well as the [fallback function](#). This function does not take any arguments and it is triggered in three cases:

1. If none of the functions of the call to the contract match any of the functions in the called contract
2. When the contract receives ether without extra data
3. If no data was supplied

Let us consider a vulnerable smart contract – a victim contract, in a simplified version of a reentrancy attack. Note that the code is demonstrated by [Solidity programming language](#) as it's the most popular and developed language of Ethereum.



Reentrancy Attack Example

An attacker can recursively call Victim.withdraw() method

Victim Contract

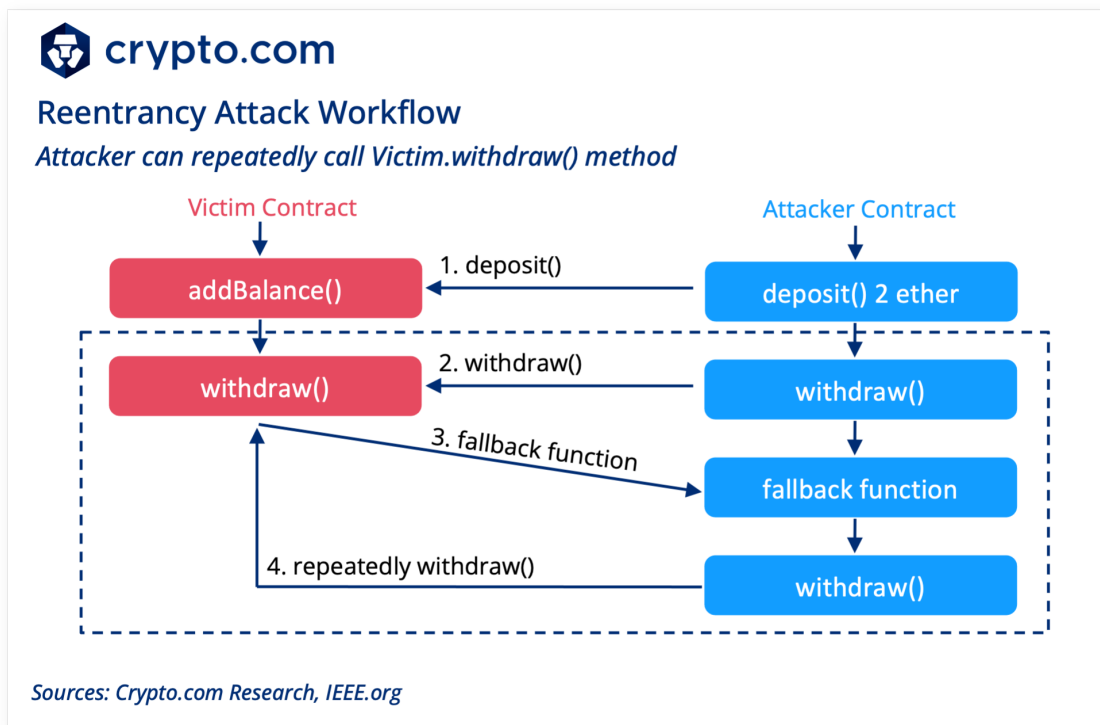
```
contract Victim {
    mapping(address=>uint) balance;
    function addBalance() public{
        balance[msg.sender] += msg.value;
    }
    function withdraw() public{
        uint amount = balance[msg.sender];
        bool res = msg.sender.call.value(amount());
        if (!res) { throw; }
        balance[msg.sender] = 0;
    }
}
```

Attacker Contract

```
contract Attacker {
    address Victim;
    function Attacker(addr _victim) {
        Victim = _victim;
    }
    // fallback function
    function() payable{
        if(address(this).balance < 9999 ether)
            { withdraw(); }
    }
    function deposit() {
        Victim.call.value(2).addBalance();
    }
    function withdraw() { Victim.withdraw();}
}
```

Sources: *Crypto.com Research, IEEE.org*

The above example includes a **Victim contract** and an **Attacker contract**. As shown, anyone can deposit ether into **Victim contract**, the number of ether is recorded in the **mapping balance**. The ether deposited can be withdrawn by calling **withdraw()**, which sends the ether to the **msg.sender address**. This transfer implicitly triggers a **fallback method** (an anonymous method that does not take any arguments) of the **Attacker**. This default behavior can have security consequences as the execution flow can be controlled by a remote fallback method. The reentrancy attack can be launched by an attacker using the **Attacker contract**. A demonstrative workflow is shown below.



1. The Attacker first calls the `deposit()` method to deposit 2 ether into the Victim Contract
2. Now the Attacker is ready to attack the target victim by calling the `Attacker.withdraw()` method
3. Subsequently, `Attacker.withdraw()` calls `Victim.withdraw()` which then triggers a recursive `Victim.withdraw()` call via the Attacker contract's fallback method
4. The above attack strategy effectively moves **more ether from the Victim contract** to the account controlled by the Attacker

2.2 Case Studies

Reentrancy attacks are difficult to avoid since this class of bug can take many forms. Developers should try to avoid coding reentrancy-vulnerable functions into a smart contract. Otherwise, the malicious adversaries are able to carefully analyse the workflow of the code and stealthily launch this attack type.


In the past few years, many DeFi platforms suffered from reentrancy exploits. Apart from the original The DAO attack, over \$112M were stolen, targeting [Grim Finance](#), [Visor Finance](#), [Cream Finance](#), [DForce](#), and [Uniswap](#).

Platform	Date	Blockchain	Value Stolen	Attacker Address
Visor Finance	21 December 2021	Ethereum	\$8.2M	Untraceable
Grim Finance	19 December 2021	Fantom	\$30M	0xdefc...
Cream Finance	30 August 2021	Ethereum	\$29M	0xce1f...
DForce	19 April 2020	Ethereum	\$25M	0xa6a6...
Uniswap & Lendf.me	18 April 2020	Ethereum	\$25M	0xae7d...

2.3 Mitigation & Prevention

The community has proposed several efficient solutions to prevent reentrancy attacks, ranging from programming language and coding practises to the attack detection and prevention tools.

1. At the smart contract programming language level (e.g. Solidity), the methods of [send\(\) and transfer\(\)](#) are recommended rather than using the more vulnerable `call.value()`.
2. For developers' coding policy, [we have recommended finishing all internal work \(ie. state changes\) first, and only then calling the external function.](#) As indicated below, the reentrancy attack can be mitigated if we move `'balance[msg.sender]=0'` before transferring value.



Reentrancy Attack Mitigation

Towards more secure coding practices

Insecure Code	Secure Code
<pre style="margin: 0;"> contract Insecure { mapping(address=>uint) balance; function addBalance() public{ balance[msg.sender] += msg.value; } function withdraw() public{ uint amount = balance[msg.sender]; bool res = msg.sender.call.value(amount()); if (!res) { throw; } balance[msg.sender] = 0; } }</pre>	<pre style="margin: 0;"> contract Secure { mapping(address=>uint) balance; function addBalance() public{ balance[msg.sender] += msg.value; } function withdraw() public{ uint amount = balance[msg.sender]; balance[msg.sender] = 0; bool res = msg.sender.call.value(amount()); if (!res) { throw; } } }</pre>

Source: Crypto.com Research

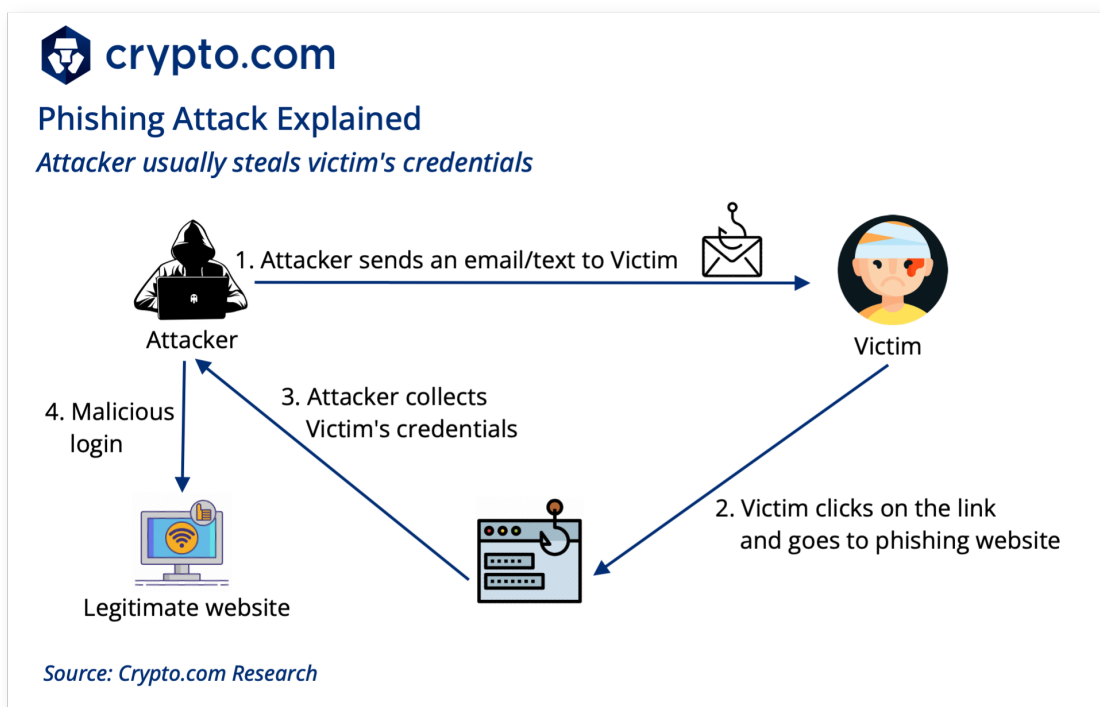
3. Meanwhile, academia has devised several verification tools that are able to detect reentrancy vulnerability at run time. DeFi operators can employ and deploy those tools to detect & prevent reentrancy attacks.

Tool	Date	Blockchain	Team & Affiliations
ECFChecker	January 2018	Ethereum	Shelly Grossman et al, Tel Aviv Univ & VMware
Sereum	February 2019	Ethereum	Michael Rodler et al, Univ of Duisburg-Essen & NEC Lab
SODA	February 2020	Ethereum	Ting Chen et al, UESTC & HK Polytechnic Univ & Texas Univ & Univ of Guelph
SMACS	July 2020	Ethereum	Bowen Liu et al, SUTD & Chinese Academy of Science
TxSpector	August 2020	Ethereum	Mengya Zhang et al, The Ohio State Univ

3. Phishing Attack

3.1 Attack Explained

Phishing is a type of social engineering attack often used to steal user data, including login credentials and credit card numbers. As shown below, the phishing attack occurs when an adversary, masquerading as a trusted entity, dupes a victim into opening an email, instant message, or text message. The victim is then tricked into clicking a malicious link, which can lead to the installation of malware, freezing of the system as part of a ransomware attack or revealing sensitive information.



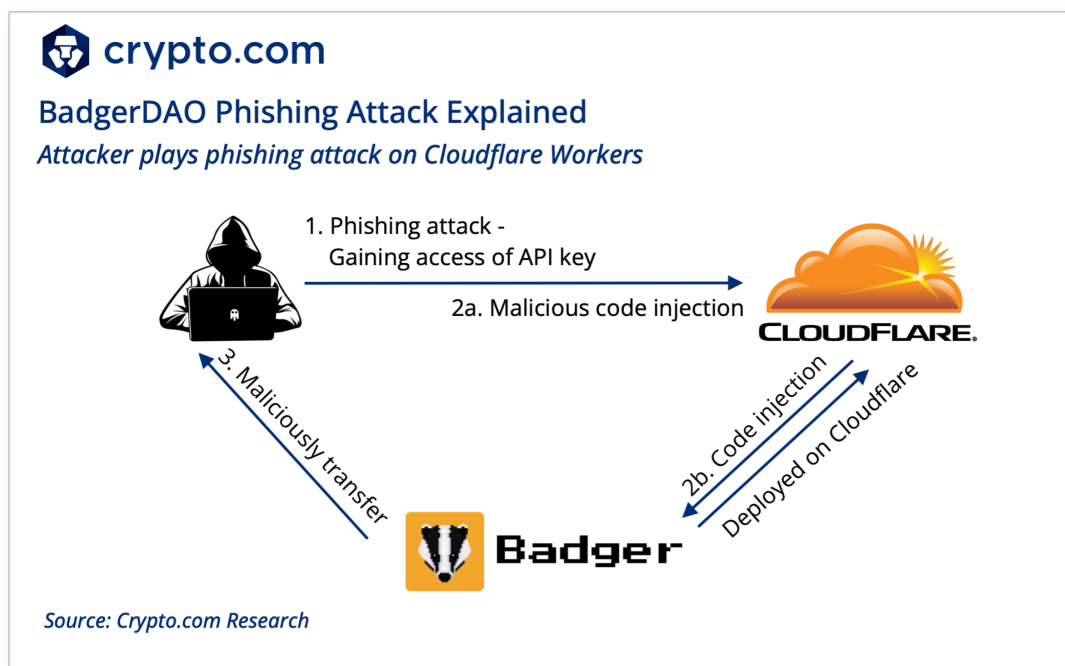
3.2 Case Studies

Phishing attacks are common in DeFi ecosystems. An attacker usually prepares a fake URL link and sends it to DeFi platform users or even developers to extract information. When personal credentials such as login information and API keys are leaked, the attack will compromise the victim's accounts and transfer all their crypto assets.

For instance, on 2 December 2021, a series of unauthorised transactions occurred on BadgerDAO, [resulting in the \\$120M loss of funds from its users](#). It was caused by 'a maliciously injected snippet' from [Cloudflare Workers](#), an application platform that runs on BadgerDAO's cloud network. The hacker used a compromised API key that was created without the knowledge or authorisation of BadgerDAO engineers to [periodically inject the malicious code](#) that affected a subset of its customers and steal funds.

The detailed workflow is as follows:

1. Attacker first launched a phishing attack to [maliciously gain access of creating & viewing API keys](#) of Cloudflare Workers.
2. Attacker deployed the worker script via a compromised API key that was created without the knowledge or authorisation of Badger engineers.
3. Attacker used this API access to [periodically inject malicious code](#) into the BadgerDAO application such that the affected users' funds would be transferred.



Similarly, on 5 November 2021, [a hacker stole \\$55M](#) in various currencies from bZx after one of its developers fell for a [phishing attack](#). The breach began with a phishing email sent to a developer's personal computer. That email had a malicious macro in a Word document that was disguised as a legitimate email attachment, which then ran a script on his personal computer. This led to his [personal mnemonic wallet phrase being compromised](#).

3.3 Variants of Phishing Attacks

At the programming language level, phishing attacks [exist as a variant in smart contract language Solidity](#). In general, a smart contract might perform permissions validation before crypto assets related operations.

For example, let us consider two global objects in a call chain triggered by a transaction **tx** originated from user **u**, where **tx** calls the **contract A**, and **A** calls another **contract B**. From **A**'s perspective the objects **msg.sender** and **tx.origin** [all refer to the address of u](#). However, such two objects are not identical from the view of **B**, as **msg.sender** denotes the address of **A** while **tx.origin** is the address of **u**. Therefore, some smart contracts which check whether **tx.origin** is the expected account are at risk of [phishing attacks](#). The community usually recommended checking the value of **msg.sender** with respect to permission checking.



Variant of Phishing Attack in Smart Contract Programming Language

Beware of msg.sender and tx.origin

Phishing Attack Example in Solidity Language

```
function update (address student, int age) {
    assert(msg.sender == owner || tx.origin == admin || msg.sender == admin);
    if (Age[student] == Null) {
        // initial Age[student]
    }
    Age[student].age += age;
}
```

Source: [Crypto.com Research](#)

As shown above, a phishing attack tampers a vulnerable smart contract by **bypassing the insecure permission check**. The method **update()** updates the age information to a given student address originated from an external address (i.e., **msg.sender**). The **assert()** method guarantees that only authorised roles can perform the actual update logic. To launch a phishing attack, let us assume a transaction whose **tx.origin** is a privileged account, and thus it passes the **assert()** check and updates an age to an address provided by **msg.sender**. [Unfortunately, once such a transaction invokes another smart contract](#), which invokes **update()**

back again, the value of ***msg.sender*** is no longer equal to ***tx.origin*** in this transaction, resulting in an attacker can maliciously bypass the permission checking and update arbitrary values to an arbitrary address.

3.4 Mitigation & Prevention

The mitigation of phishing attacks requires not only technical prevention implementation, but also the compliance of human behaviours.

In general, the prevention strategies can be divided into three classifications:

1. **Good phishing attack training and routine self-checking.** For instance, every employee should carefully check the correctness of email addresses, URLs, and text messages before clicking any external link.
2. Companies may use SSL certificates to secure all traffic to and from their website. This protects the information being sent between the web server and user browsers from eavesdropping.
3. Regularly training technical developers with secure coding habits so that variants of phishing attacks (which occurs at low-level programming language) can be prevented.

4. Flash Loan Attacks & Price Manipulation

4.1 Attack Explained

[Decentralised lending platforms](#) provide loans to businesses or the public with no intermediaries present. On the other hand, DeFi lending protocols enable everyone to earn interest on supplied stable coins and cryptocurrencies. **Before the invention of flash loans, DeFi lending required users to over-collateralise a loan upfront in order to borrow funds. Flash loans introduced a new form of borrowing whereby a user can borrow a large amount of funds without putting up collateral**, allowing them to leverage themselves without risking their own funds. By definition, flash loans are only valid within **one transaction** and must be **repaid by the end of that transaction**.

Generally, many DeFi platforms require real-time information about the market price of the assets used as borrowed loans or redemption. To implement this functionality, [DeFi protocols introduce oracles, third parties reporting the price of assets from real-world \(off-chain\) sources](#). Before diving into the details of the flash loan attack, we should understand what exactly an oracle is.

Unlike the straightforward interactions between two on-chain entities, transferring information from external off-chain sources to a smart contract creates new challenges. Bridging the connection between the blockchain and the outside world requires an additional infrastructure known as an **oracle**. The popular oracle solutions include on-chain DEXs ([Uniswap](#), [dYdX](#)) and decentralised oracle networks ([Chainlink](#)), etc.

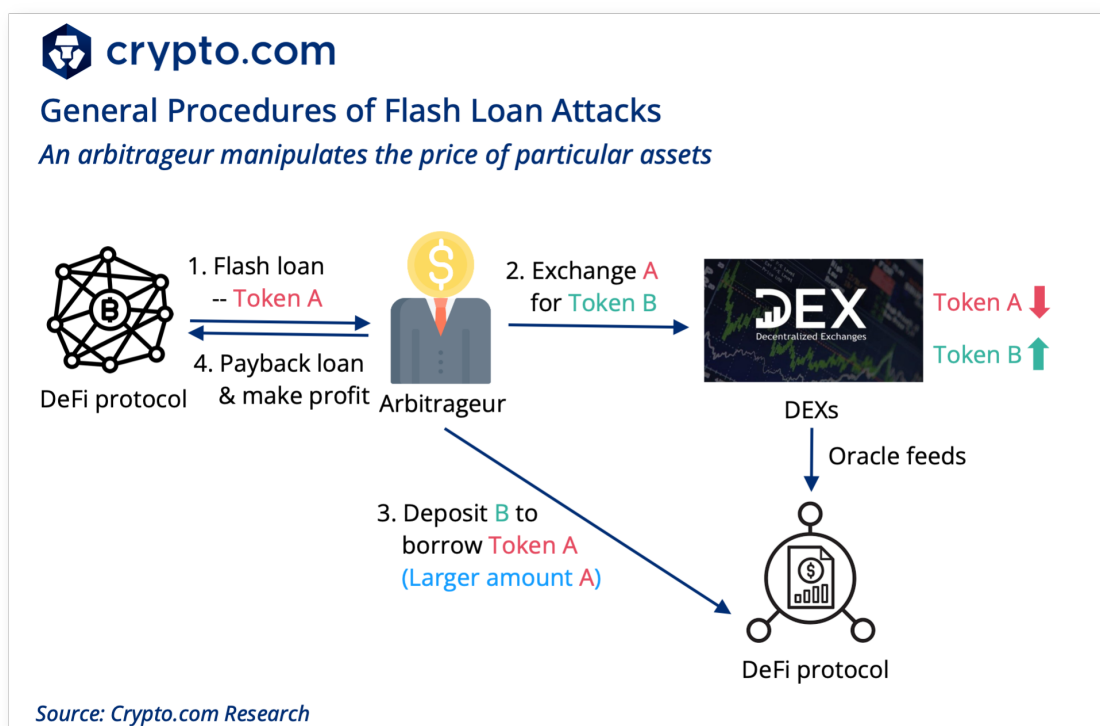
In practice, flash loan attacks usually couple with price (or oracle) manipulation. **The arbitrageur plays the strategies by lowering or increasing the particular assets and makes profit.** A successful flash loan attack involves the following steps:

1. An arbitrageur looks for a DeFi protocol that supports flash loans, and borrows a large number of token A as a flash loan.
2. Arbitrageur exchanges token A for token B. Due to large amount exchange, this results in lowering the price of token A and raising the price of token B.
3. Subsequently, the arbitrageur selects another DeFi lending protocol as the target, and deposits token B as collateral to borrow out a larger

number of token A due to the increased price of token B. Note that the criteria of protocol selection is that this DeFi protocol employs the above DEX as its sole oracle data feed.

4. Eventually, the arbitrageur uses the borrowed token A to payback the previous flash loan. As a larger number of token A was borrowed out, the arbitrageur is able to make profit with the remaining token A.

What are the consequences? Apart from the unearned profit from the perspective of arbitrageur, [as the prices of token A and B on the DEX get arbitrated back to the true market-wide price](#), the DeFi protocol is left with an undercollateralised position (i.e. the debt worth more than collateral), directly harming users such as liquidity providers of the pool.

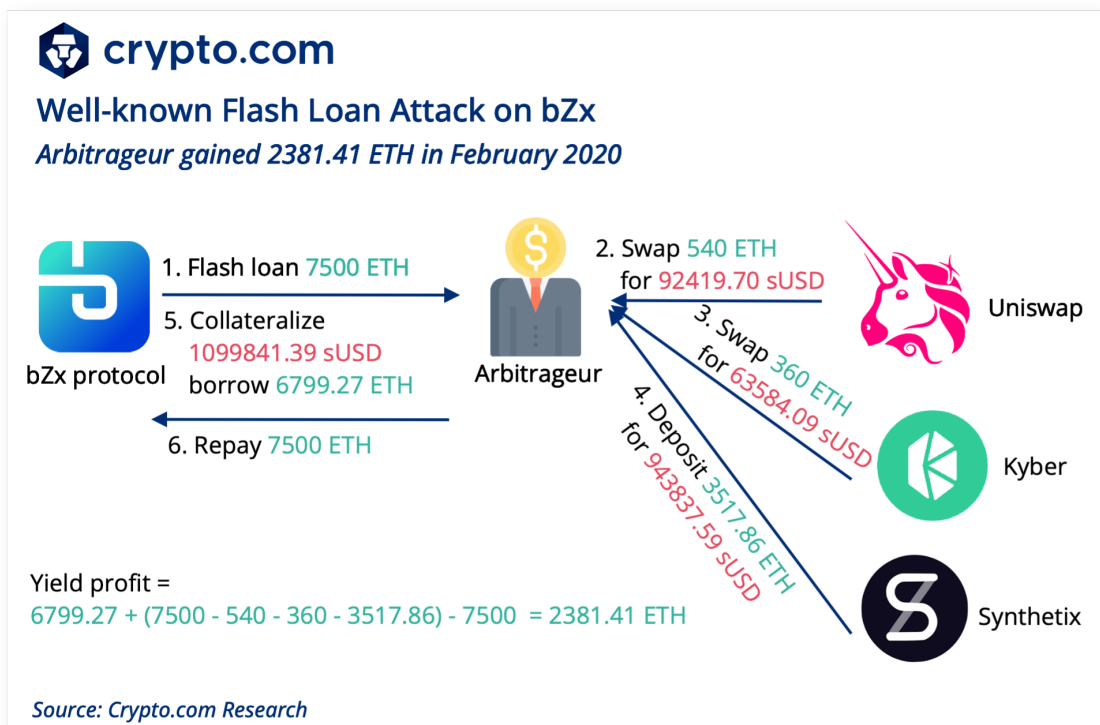


4.2 Case Studies

The example above is a formal (or simplified) version of a flash loan attack. **In real world scenarios, the attackers generally play on more sophisticated strategies to launch a flash loan attack.** [Below, we select a pioneer, well-known, but more complex case that attacked bZx platforms and yielded a profit of 2381.41 ETH \(\\$634.9k\), happened on 18 February 2020.](#)

The core of this attack is an oracle manipulation using a flash loan, which lowers the price of sUSD/ETH. In the following, the arbitrageur benefits from this decreased sUSD/ETH price by borrowing ETH with sUSD as collateral.

1. In Step 1, the arbitrageur borrows a flash loan of 7500.00 ETH from bZx.
2. In the next three steps (Step 2 - 4), the arbitrageur converts a total of 4417.86 ETH to 1099841.39 sUSD (at an average of 248.95 sUSD/ETH).
3. The exchange rates in Step 2 and 3 are 171.15 and 176.62 sUSD/ETH, respectively. These two steps decrease the sUSD/ETH price to 106.05 sUSD/ETH on Uniswap and 108.44 sUSD/ETH on Kyber Reserve, which are collectively used as a price oracle of the lending platform bZx. The trade on the Synthetix in Step 4 is yet unaffected by the previous trades. The adversarial trader then collateralises all purchased sUSD (1099 841.39) to borrow 6799.27 ETH.
4. Now the arbitrageur possesses 6799.27+3082.14 ETH and in the last step repays the flash loan amounting to 7500.00 ETH. The arbitrageur, therefore, generates a revenue of 2381.41 ETH while only paying 0.42 ETH (118.79 USD) transaction fees



Besides the above case on bZx, in the past two years, **flash loan attacks have compromised the multiple DeFi platforms with over \$200M in cryptocurrencies stolen.**

Platform	Date	Blockchain	Value Stolen
Cream Finance	28 October 2021	Ethereum	\$130M
PancakeHunny	20 October 2021	BSC	\$2M
Indexed Finance	15 October 2021	Ethereum	\$16M
Vee Finance	20 September 2021	Avalanche	\$35M
ApeRocket	14 July 2021	BSC & Polygon	\$1.3M
Belt Finance	30 May 2021	BSC	\$50M
BurgerSwap	28 May 2021	BSC	\$7.2M
PancakeBunny	20 May 2021	BSC	\$3M
Rari Capital	8 May 2021	Ethereum	\$11M
Alpha Homora	13 February 2021	BSC	\$37M
Yearn Finance	5 February 2021	Ethereum	\$11M

Balancer	28 June 2021	Ethereum & Polygon & Arbitrum	\$500k
--------------------------	--------------	-------------------------------	--------

4.3 Mitigation & Prevention

Flash loan attacks can be mitigated in two ways.

On the one hand, DeFi operators should avoid using centralised/single oracles for external data feeds. Instead, some best practises include the [Time Weighted Average Price \(TWAP\) oracle adopted by Uniswap](#) and decentralised oracle networks like Chainlink.

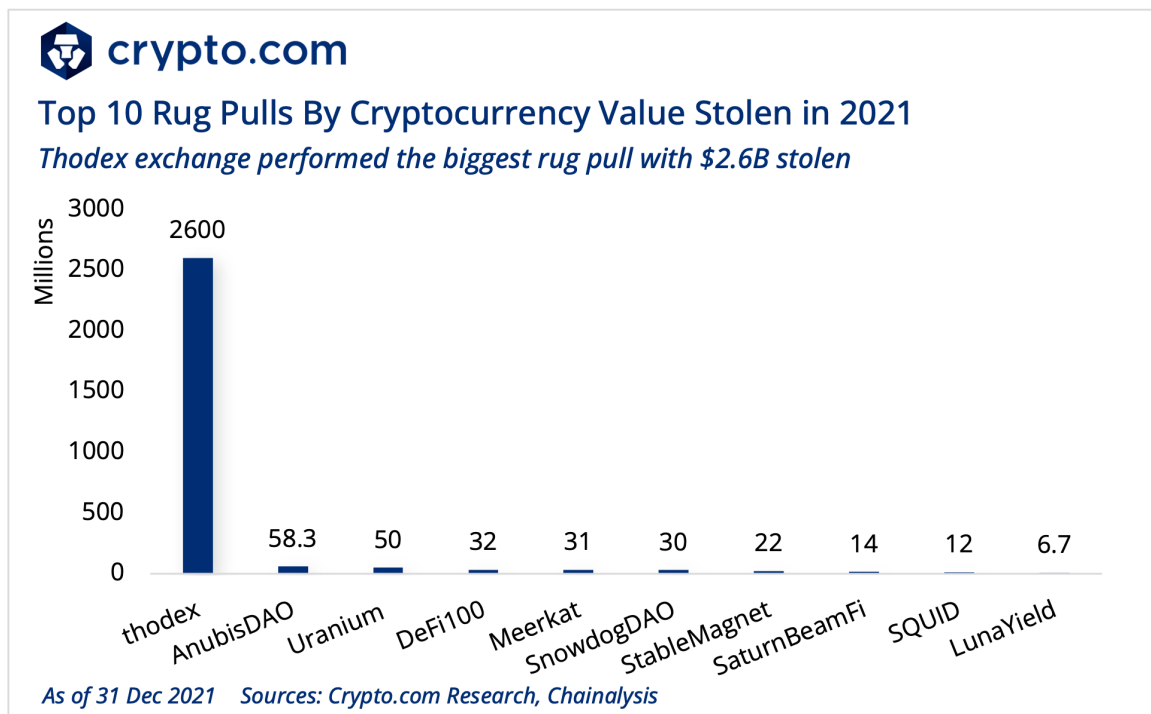
On the other hand, it is recommended to leverage tools which can identify Flash loan attack possibility, [such as OpenZeppelin](#). It has been adopted by Synthetix, Yearn Finance, and [Opyn](#).

5. Rug Pulls

5.1 Attack Explained

Rug pulls are one common DeFi attacks in which an individual in the company with access to the company’s contracts misuses their privileges for draining value from the protocol. [In all cases, the project and the team disappear into oblivion with little left to solve the issue.](#) Generally, we use it to refer to cases in which developers build out what appear to be legitimate cryptocurrency projects — meaning they do more than simply set up wallets to receive cryptocurrency for, say, fraudulent investing opportunities — before taking investors’ money and disappearing.

According to a report by Chainalysis, rug pulls took [more than \\$2.8B worth of cryptocurrency](#) from victims in 2021.



5.2 Case Studies

Most rug pulls entail developers creating new tokens and promoting them to investors, who trade for the new token in the hopes the token will rise in value, which also provides liquidity to the project — just how most DeFi projects start. [In rug pulls, however, the developers eventually drain the funds from the liquidity pool, sending the token's value to zero, and disappear.](#) **Rug pulls are prevalent in DeFi** because with the right technical know-how, it's cheap and easy to create new tokens on the Ethereum blockchain or others and get them listed on decentralised exchanges (DEXes) without a code audit. That last point is crucial — decentralised tokens are meant to be designed in such a way that investors holding governance tokens can vote on things like how assets in the liquidity pool are used, which would make it impossible for the developers to drain the pool's funds. While code audits that would catch these vulnerabilities are common in the space, they're not required in order to list on most DEXes, hence why we see so many rug pulls.

Platform	Date	Blockchain	Value Stolen
AnubisDAO	1 November 2021	Ethereum	\$60M
StableMagnet	23 June 2021	Ethereum & BSC	\$27M
Alchemix	16 June 2021	Ethereum	\$6.5M
Uranium Finance	28 April 2021	BSC	\$50M
TurtleDex	19 March 2021	BSC	\$2.4M
Meerkat Finance	4 March 2021	BSC	\$31M

In the table above, several rug pulls events that took place in 2021 on six DeFi protocols are listed. For instance, on 4 March 2021, Meerkat Finance, a DeFi yield project using a forked cryptocurrency of Yearn.finance, perpetrated a rug pull worth \$31M in crypto assets. The platform's official Telegram channel claimed that its smart contract vault had been compromised. Investigations suggested that either the private key of the Meerkat deployer was compromised or the incident was self-directed by the owners of the project.

5.3 Mitigation & Prevention

Avoiding scams and potential rug pulls in crypto projects ultimately boils down to awareness before investing and doing your own research. Below, we list a few ways to check the legitimacy of any project.

1. **Check liquidity** – Legitimate projects usually lock up a significant number of tokens for a long time, which cannot be withdrawn from the liquidity pool during that time frame. Check the project's staking period, and verify the amount of liquidity belonging to project owners.
2. **Review the project's Github, whitepaper, and social media channels** – Github usually contains their development activities. It's well worth keeping an eye on these, as well as a project's social media channels including Telegram, Twitter, etc.
3. **Confirm team credibility** – Any project that's potentially a rug pull is defined by its owners and developers. Their relevance to the cryptocurrency space, as well as their previous involvements, track records, social media, industry history, and connections must add up if they are to gain credibility.
4. **Look at holders and listings on DEX Platforms** – If a token has only a few token holders and isn't actively traded on multiple platforms, it's possible that it might be a rug pull waiting. Tools like Etherscan and CoinGecko can reveal more information about a token.

6. Code Bugs

6.1 Attack Explained

Like all computer programs, it is likely that most DeFi protocols contain errors. These kinds of errors should be addressed even more seriously than ordinary program bugs as they are involve monetary-related.


6.2 Case Studies

Below, we explain five exploitation events in DeFi space due to the code flaws.

Platform	Date	Blockchain	Value Stolen	Bug
Polygon	29 December 2021	Polygon	\$24B (at risk)	Lack of balance/address checking
Pizza	9 December 2021	BSC	\$5M	Integer overflow
Compound	30 September 2021	Ethereum	\$80M (at risk)	Multi-sig wallet missing
MakerDAO	4 September 2021	Ethereum	\$4M	Re-callable Init()
bZx	15 September 2020	Ethereum	\$8M	Parameter update bug

6.2.1 Lack of Balance/Address Checking


On [29 December 2021](#), a critical vulnerability in the Polygon's genesis contract was highlighted, putting \$24B of MATIC at risk. The vulnerability consisted of a lack of balance/address check in the **transferWithSig()** and **_transferFrom()** functions of [Polygon's BaseERC20.sol contract](#) and allowed an attacker to maliciously steal all balances from the genesis contract.



Lack of Balance/Address Checking Bug in Polygon

Do check the validity of balance & address before transfer operation

1. invalid signature
to --> attacker's addr
amount --> victim's fund



Attacker

3. _transferFrom()
victim's balance

```

contract PolygonERC20 {
    function transferWithSig(bytes sig, address to, uint amount) {
        .....
        // sig is invalid, recovered from address is genesis contract (victim)
        from = ecrecovery(sig);
        // balance & address of from is missing
        _transferFrom(from, to, amount);
    }
        
```

2. ecrecovery()
-- victim's address

Sources: Crypto.com Research, Medium

The adversarial strategies are as follows:

1. An attacker prepares an invalid ECDSA signature (a byte string with a length outside the 65-byte norm), an [amount](#) (the full balance of the victim's contract), a [to address](#) (attacker's address), and calls **transferWithSig()** function.
2. Due to the check in [ecrecovery\(\) function](#), if a passed signature does not follow the ECDSA scheme, it will return the zero address, referring to the [genesis contract](#). After the [from address](#) is recovered from the invalid signature, the **_transferFrom()** function is called.
3. In **_transferFrom()** function, the validity checking of the **from** address is missing. Moreover, as the balances are not checked for the **from** and **to**, it will directly transfer the whole **amount** to the attacker from the victim's contract.


As a remedy, Polygon finally removed the `transferWithSig()` function [from its ERC20 code repository](#).

6.2.2 Integer Overflow

Integer overflow vulnerabilities are possible because software uses variables of a fixed size to store values. This fixed size means that any variable can only store a certain range of values. [If a value goes outside of this range, then it rolls over to be interpreted as a lower value.](#)

On 9 December 2021, the [BSC DeFi platform PIZZA](#) was the victim of an attack. The attacker took advantage of vulnerabilities in eCurve to steal \$5M in tokens from the protocol due to integer overflow vulnerability.

A direct explanation is depicted below. As shown in the code snippet, if a **balance** reaches the maximum uint value (2^{256}) it will circle back to zero which is so-called integer overflow. Therefore, programmers are advised to **check the state of changes before updating parameter values.**


crypto.com

Integer Overflow Bug in Pizza

Check your code before updating parameter values

Insecure Code	Secure Code
<pre> mapping (address=>uint256) public balanceOf; function transfer(address _to, uint256 _value) { // check if sender has balance require(balanceOf[msg.sender] >= _value); // add and subtract new balance balanceOf[msg.sender] -= _value; balanceOf[_to] += _value; } </pre>	<pre> function transfer(address _to, uint256 _value) { // check if sender has balance and for overflow require(balanceOf[msg.sender] >= _value); require(balanceOf[_to] + _value >= balanceOf[_to]); // add and subtract new balance balanceOf[msg.sender] -= _value; balanceOf[_to] += _value; } </pre>

Source: [Crypto.com Research](#)

6.2.3 Multi-sig Incompleteness

[Multisignature wallets are cryptocurrency wallets that require two or more private keys to sign and send a transaction](#), which can potentially keep your funds safer. A demonstrative example can be found in this [Crypto.com report](#).


On 30 September 2021, observers have noted that Compound's Comptroller contract [is not managed by a multi-sig](#) controlled by Compound Labs, and any fix to the exploit may require a governance vote among COMP holders, which staked \$80M value at risk.

Compound acknowledged the exploit on [its official Twitter handle](#) and quickly fixed it, saying no user funds are at risk.

6.2.4 Re-callable Init()

On [4 September 2021](#), MakerDAO was exploited for \$4M. They left the [init\(\) function](#) unprotected. **The attacker re-initialised the contract with malicious data** and then [called emergencyExit\(\)](#) to get away with the funds.

As shown below, the vulnerability arose from the public **init()** function missing a check for initialisation, allowing the attacker to reinitialise 4 token contracts with malicious data. Then, the [emergencyExit\(\)](#) function was used to withdraw the funds from each. A secure practice is shown in the right code snippet. The reinitialisation actions can be avoided with blue lines that guarantees the **init()** function can be initialised **once**.



Reinitialization of init() Bug in MakerDAO

Set the initialisation state for security-critical function

Insecure Code

```

function init(args...){
    // missing @initialized parameter
}
                    
```

Secure Code

```

function init(args...){
    assert(initialized == false);
    initialized = True;
    .....// code flow
}
                    
```

Sources: Crypto.com Research, Github

6.2.5 Code Workflow Error

Interestingly, **bZx** suffered from not only flash loan attacks but also code bugs due to wrong code workflow within its smart contract code. It was attacked [on 15 September 2021](#) and lost \$8M due to a faulty code.

The flawed code allowed an attacker to duplicate assets, and increase their balance of iTokens on bZx, allowing the hacker to mint \$8M in total.

They executed the following steps in the legacy code (before exploitation):

1. The `transfer()` function was called with the same `_from` and `_to` address
2. At this point, having `_from` and `_to` as the same address will result in `_balancesFrom` and `_balancesTo` being equal.
3. Then, line #3 - #4 decrease the balance of `_balancesFrom` and line #5 - #6 increase balance of `_balancesTo`.
4. Lastly the most important part is storing `_balancesFromNew` and `_balancesToNew`. The user was effectively able to increase his balance artificially.



Code Workflow Bug in bZx

Do review the code flow of updating key parameters

Insecure Transfer() Code

```
#1 uint256 _balancesFrom = balances[_from];
#2 uint256 _balancesTo = balances[_to];
#3 uint256 _balancesFromNew = _balancesFrom
    .sub(_value, '16');
#4 balances[_from] = _balancesFromNew;
#5 uint256 _balancesToNew = _balancesTo
    .add(_value);
#6 balances[_to] = _balancesToNew;
```

Secure Transfer() Code

```
#1 uint256 _balancesFrom = balances[_from];
#2 uint256 _balancesFromNew = _balancesFrom
    .sub(_value, '16');
#3 balances[_from] = _balancesFromNew;
#4 uint256 _balancesTo = balances[_to];
#5 uint256 _balancesToNew = _balancesTo
    .add(_value);
#6 balances[_to] = _balancesToNew;
```

Sources: *Crypto.com Research, bZx*

In the patched code (after fixing), the fix saw the move of `balancesTo` being set after the deduction from `balances[_from]`. This prevents a user from inflating their balance.

What can we learn from this attack? When it comes to updating the values of key parameters, it is important to carefully review the code flow.

7. Poor Access Control

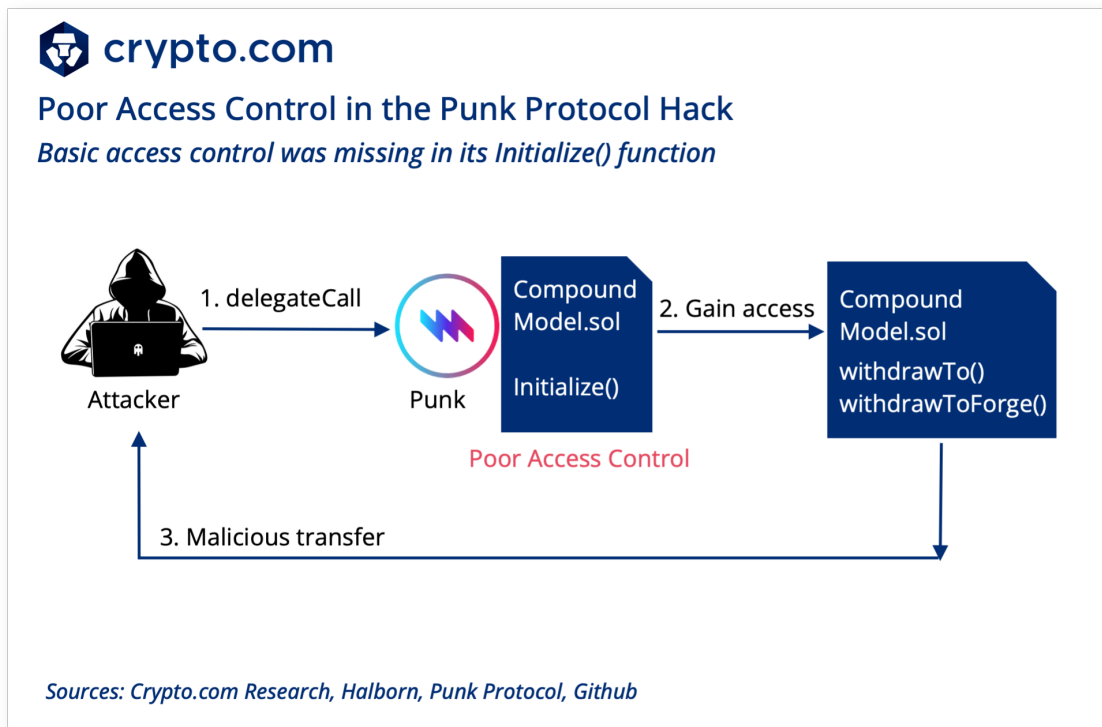
7.1 Attack Explained

Access control is a traditional security mechanism that regulates and limits the access permission to particular system resources. For instance, on Ethereum, by applying well-designed and suitable criteria the access of contract functionalities can be restricted. A few build-in methods (i.e., [assert\(\)](#) and [require\(\)](#)) have been already proposed in the contract programming languages to facilitate a fundamental level of access control, which are limited and far from being enough. Due to [high on-chain storage cost and limitations of the smart contract programming language](#), no fine-grained access control mechanisms could be enforced on the contract side.

Usually, many DeFi protocols leverage [OpenZeppelin](#) to provide templates for e.g. role-based access control. Unfortunately, this solution does not allow for flexible access control management, and can't block all malicious access from suspicious actors. As a result, we can see numerous security exploit events due to poor access control enforced.

7.2 Case Studies

In August 2021, the [Punk protocol](#) was the victim of a hack. A vulnerability in the project's smart contracts led to [over \\$8.9M in tokens extracted from the project](#).

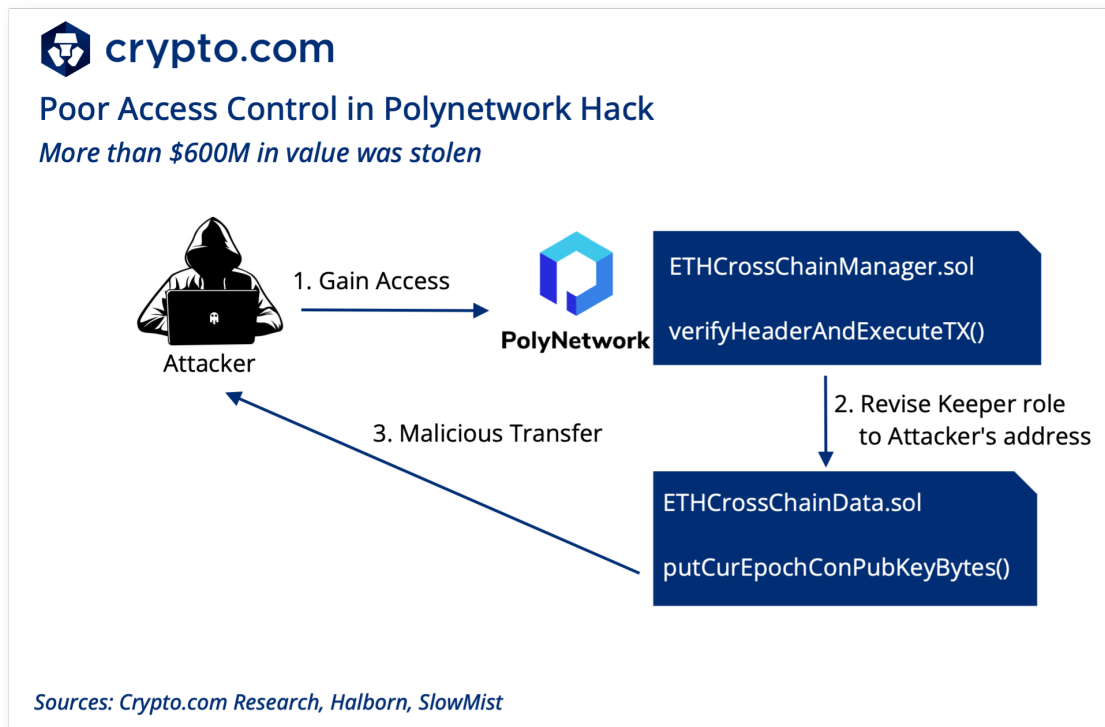


As indicated above, the root cause is a missing [modifier function](#) (access control) in the `initialize()` function within the [CompoundModel.sol code](#). After gaining access to the `initialize()` function, the attacker could call the `withdrawTo()` and `withdrawToForge()` functions to [send the tokens stored in the contract to the attacker-control address](#). More specifically:

1. The attacker used `delegateCall()` to replace what should have been the protocol's `forgeAddress` with their own [malicious contract](#), as a parameter of the `CompoundModel`'s `initialize()` function. The lack of an 'initializer()' Modifier meant that such the function omitted good access control logic. [Any external account can call it in addition to revising any core parameter](#).
2. With the malicious contract address updated, the attacker was then able to call `withdrawTo()` and `withdrawToForge()`, sending the assets controlled by the `CompoundModel` directly to his malicious contract, and into their [wallet](#).

Based on [the official remedy by Punk protocol on its github](#), the logic of access control has been enforced on `initialize()` function. Currently, only the platform admin is able to bypass the access control and alter the particular parameter values.

Similarly, in August 2021, you are likely familiar with the [hack on the Polynetwork](#) protocol, which was the victim of the biggest DeFi hack to date. The attacker stole an estimated \$600.3M in various tokens by exploiting a vulnerability (i.e. poor access control) in the protocol's smart contracts.



As shown above, the hack was made possible by mismanagement of the access rights between two important Polynetwork smart contracts. The first one is [EthCrossChainManager](#) and the second one is [EthCrossChainData](#).

The **EthCrossChainManager** contract holds high privilege that has the right to trigger messages from another chain to the Poly chain. Any external account can call a cross-chain event by issuing a transaction on the source chain that invokes the [verifyHeaderAndExecuteTx\(\)](#) function within EthCrossChainManager, and specifying a target Poly contract to execute. However, poor access control was found in this function.

The malicious adversary first breached the access of **verifyHeaderAndExecuteTx()** within EthCrossChainManager contract, which subsequently called the [putCurEpochConPubKeyBytes\(\)](#) function of the **EthCrossChainData** contract. This contract is [responsible for setting and managing a list of public keys of 'authenticator nodes' \(i.e., keepers\)](#) that manage the wallets in the underlying liquidity chains. In other words, EthCrossChainData can decide who has the privilege of moving the large amount of funds. [Eventually, the adversaries simply set their own public keys to replace that of a keeper](#), and

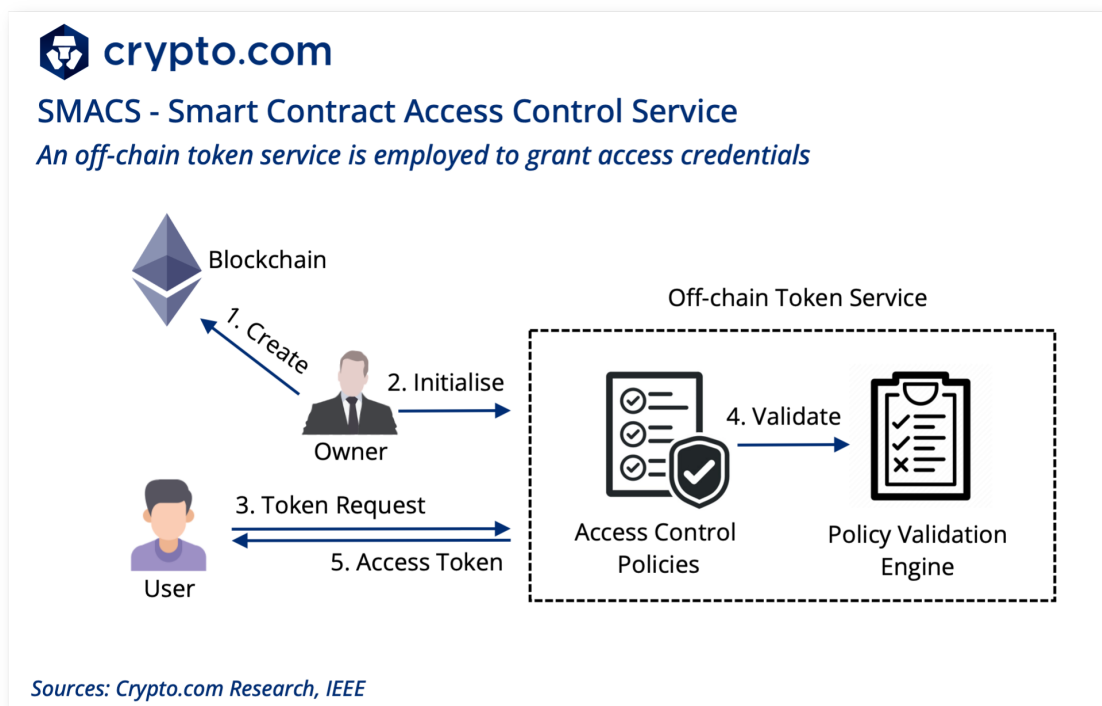
then have the right to execute a high volume transaction within the Poly network to exfiltrate a large amount of funds to other wallets.

7.3 Mitigation & Prevention

As discussed above, basic access control can be implemented at programming language level. **When it comes to fine-grained access control, DeFi protocols are expected to devise crafted access control mechanisms and integrate these into more advanced frameworks.**

One line of research lies in blacklists and whitelists to limit the access from external suspicious addresses. These lists are supposed to be dynamically updated and easily managed by protocol governors.

Another aspect is to employ third-party access control public services. According to the [SMACS research project](#), an off-chain public access service can be employed, in which all sophisticated access control policies are put into such a service that can validate the malicious transaction calls and issue the access credentials to protocol users.



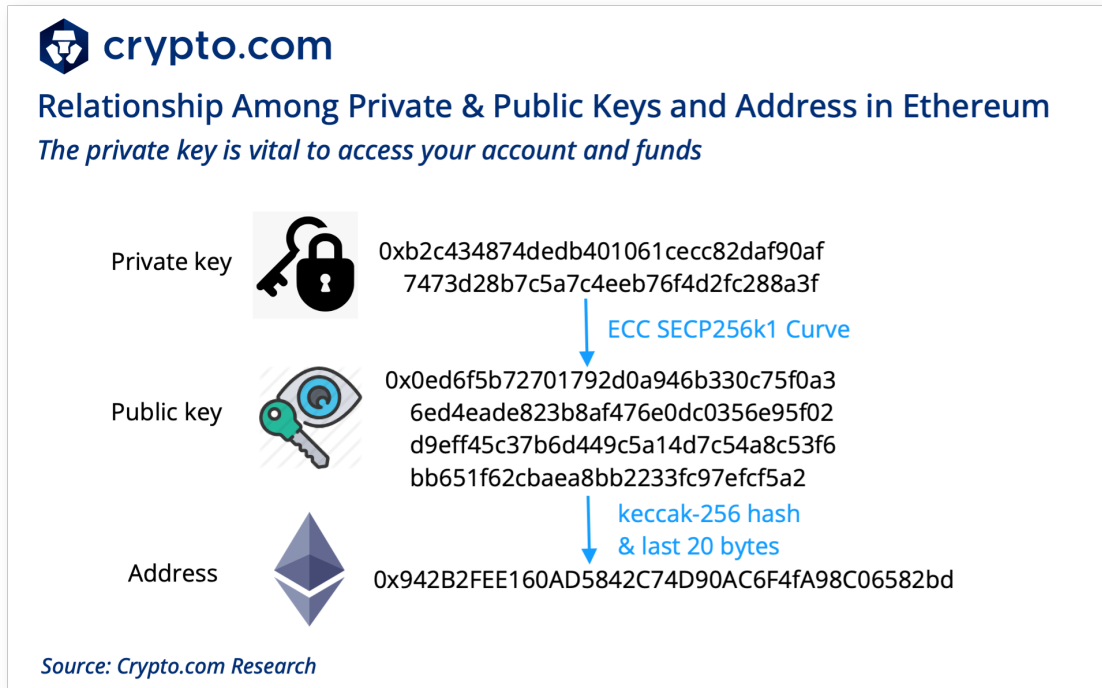
8. Compromised Private Key

8.1 Attack Explained

When you are interacting with a blockchain ecosystem, e.g. to buy cryptocurrencies or deposit crypto assets into lending platforms, you are issued two keys: a **public key**, which works like an email address (meaning you can safely share it with others, allowing you to send or receive funds), and a **private key**, which is typically a string of letters and numbers (and which is not to be shared with anyone). [You can think of the private key as a password that unlocks the virtual vault that holds your money.](#) As long as you (and only you) have access to your private key, your funds are safe and can be managed anywhere in the world with an internet connection.

Here we briefly demonstrate the relationship between private key, public key, and blockchain valid address in Ethereum. As indicated, a valid Ethereum address is generated by three necessary steps:

1. Create a random private key. [A private key is 64 hexadecimal characters](#) (or 256 bits/32 bytes) that will access your account.
2. Derive a public key from the private key. A public key has 64 bytes, which is generated with [Elliptic Curve Digital Signature Algorithm \(ECDSA\) and SECP256k1 curve](#). Note that public key and private key are a matched key pair.
3. Take the 64-byte public key as input and compute its [keccak-256 hash value](#). The output is 32 bytes and the last 20 bytes would be a valid Ethereum address paired with the above key pair.



Why are private keys so important? Firstly, cryptocurrencies like Bitcoin and Ethereum are decentralised, meaning there is no bank or any other institution in the middle holding your digital money. Keeping private keys helps you to protect your funds. Secondly, the public key is generated by your private key by dedicated cryptography constructions, which makes them a matched pair. When you make a transaction using your public key, you verify that it's really you by using your private key. Finally, taking Bitcoin as an example, even though any curious observer can see when Bitcoin is bought or sold or used, only the holder of a private key can make those transactions.

8.2 Case Studies

There are a number of private key leakage exploitations in several DeFi platforms. **The consequences of private key compromising are always devastating since the internal private keys are usually linked with individual or protocol funds.**

For instance, on 30 July 2021, the [private key that enabled interaction with the timelock was publicly available](#) on the [Levyathan project](#) github repository, due to a serious oversight of the developer. This oversight allowed a malicious hacker to gain control of the LEV contract and mint an infinite number of tokens before dumping them on the market. **The catastrophic consequence** was that Levyathan disappeared from the market.

Furthermore, another two similar attacks compromising wallet's private keys of [EasyFi](#) and [Nexus Mutual](#) stole around [\\$81M](#) (on 19 April 2021) and [\\$8M](#) (on 14 December 2020), respectively.

8.3 Mitigation & Prevention

It is probable that this initial compromise was made possible by human error (clicking on a phishing link, use of a weak password, etc.). An effective approach to prevent private keys from being compromised is good internal training. In addition, keeping keys in a safe place and employing tools to resist & detect potential threats are helpful to avoid private key to be leaked.

9. Conclusion

In this report, we studied historical, infamous security exploits in the DeFi space, including reentrancy, phishing, and flash loan attacks, rug pulls, common code flaws, poor access control and compromised private keys.

By exploring the adversarial strategies of each attack type, we dispelled the mist of the full workflows for particular attacks. Furthermore, we provided real-world case studies that we can learn from the history and presented corresponding high-level prevention solutions.

To be sure, no system is secure forever. Thus, countering the attack vectors never ends. The severity of attacks has fueled interest in the community from both industry and academia to enhance the security of the underlying DeFi environment, and we believe that the ecosystem will become increasingly robust and secure.

References

- Liu, Bowen, et al. "A First Look into DeFi Oracles." 2021 IEEE International Conference on Decentralized Applications and Infrastructures, vol. 10.1109/DAPPS52256.2021.00010, no. 2021, 2021, pp. 39-48. IEEE, <https://www.computer.org/csdl/proceedings-article/dapps/2021/348500a039/1xR7kYeK4De>. Accessed 01 01 2022.
- Liu, Bowen, et al. "Smacs: smart contract access control service." 2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, vol. 978-1-7281-7260-6, no. 2020, 2020, pp. 221-232. IEEE, <https://ieeexplore.ieee.org/document/9153398/references#references>. Accessed 01 01 2022.
- Qin, Kaihua, et al. "Attacking the DeFi Ecosystem with Flash Loans for Fun and Profit." International Conference on Financial Cryptography and Data Security, vol. 12674, no. 2021, 2021, pp. 3-32. Lecture Notes in Computer Science, https://link.springer.com/chapter/10.1007%2F978-3-662-64322-8_1. Accessed 01 01 2022.
- Zhou, Liyi, et al. "High-Frequency Trading on Decentralized On-Chain Exchanges." 2021 IEEE Symposium on Security and Privacy, vol. 978-1-7281-8934-5, no. 2021, 2021, pp. 428-445. IEEE, <https://ieeexplore.ieee.org/document/9519421>. Accessed 01 01 2022.



crypto.com

e. contact@crypto.com

© Copyright 2022. For information, please visit crypto.com