TokenCat: Detect Flaw of Authentication on ERC20 Tokens

Zheyuan He[1], Zhou Liao[1], Feng Luo[1], Dijun Liu[2], Ting Chen[✉1] and Zihao Li[3]

[1]Center Of Cyber Security, University of Electronic Science and Technology of China
[2]Foundational Security, Ant Group
[3]Department of Computing, The Hong Kong Polytechnic University
Email: ecjgvmhc@gmail.com

ICC 2022 - IEEE International Conference on Communications | 978-1-5386-8347-7/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICC45855.2022.9839252

***Abstract*—The development of blockchain has promoted the prosperity of the cryptocurrency ecosystem. The majority of cryptocurrencies are ERC20 tokens implemented based on Ethereum contracts. The major role of ERC20 tokens is to carry out various trades and loans in decentralized applications (DApps). To participate in DApps, users must grant the DApps permission to spend tokens on user behalf. However, if the authorization logic of token contract is flawed implementation, the holder of token will suffer tremendous financial losses. In this work, we detect the authentication implementation of the flaws in ERC20 token, which has not been done before. We find the authentication process of the token is implemented by operating the authentication data structure of the token. Therefore, we capture the operations of the authentication data structure in token contract to infer authentication behaviors and detect authentication defects. However, it's not a simple task as most smart contracts are not open source and the bytecode of token contract lacks type information. To tackle these problems, we utilize symbolic execution on the token bytecode, then identify the authentication data structure and capture the operations by parsing the symbolic expressions, and finally detect authentication defects through the inferred authentication behavior. To best our knowledge, this is the first work to detect the flaws in the implementation of authentication in ERC20 Token. To automate the analysis, we implement our approach in a new tool named `TokenCat` and use it to inspect 245,822 tokens. As a result, the `TokenCat` found 491 ERC20 token authentication implementation flaws with 94% precision.**

***Index Terms*—Ethereum, smart contract, token, ERC20, authentication**

I. Introduction

As a digital asset with market value of hundreds of billions of dollars, cryptocurrencies are gaining great popularity. Most cryptocurrencies named tokens are implemented by the smart contract on a blockchain platform Ethereum [1]. The smart contracts are programs that can be executed on Ethereum. To regulate the interaction between these tokens and users as well as third-party tools (e.g., wallets, exchange markets, etc.), several standards (e.g., ERC20, ERC721, etc.) have been proposed for the implementation of token contracts [2]. Among these standards, the ERC20 standard is broadly applied, and the token contract based on ERC20 standard is one of the most popular smart contracts in Ethereum. The ERC20 standard contains six standard interfaces and two standard events. As of September 1st, 2019, more than 160,000 ERC20 compatible tokens exist on Ethereum blockchain platform [3].

Currently, ERC20 tokens have seen widespread adoption in Decentralised Finance (DeFi). DeFi is a decentralized financial transaction platform based on blockchain and smart contract. Thousands of tokens are staked, loaned, and traded in different DeFi projects. If user wants to participate in the DeFi project, user has to utilize the authentication method in ERC20 to grant the DeFi projects permission to spend tokens on his behalf. Such token's authentication is an integral part of the DeFi protocol.

However, if the implementation of token's authentication is flawed, it will cause devastating consequences. For example, due to flawed authentication implementation, the DeFi project named Primitive Finance [4] put its funds at risk. If the Primitive Finance project team does not find the vulnerable in time, it will cause a loss of 1M USD [5]. Unfortunately, few people focus on the flaws in authentication of ERC20 tokens.

The standard authentication process is that the user's account first authorizes the token of the specified amount by the third party, and then the third party can transfer the token of the specified amount from the user's account. The design flaws of token authentication will cause two consequences. First, the attacker can steal the user's token by modifying the user's authentication record. Second, the user's modification of the authentication record may be inconsistent with the user's expectations changes. This may cause loss of assets.

In this work, we detect the authentication implementation of the flaws in ERC20 token, which has not been done before. We first found that the token contract implements the authentication behavior by operating an authentication data structure (defined in in §III). Then, we infer the authentication behavior by capturing the operations about authentication data structure. We define the standard and flawed authentication behavior in §III. Finally, we detect authentication defects in authentication by comparing the difference between the current authentication behavior and the standard authentication behavior. It is nontrivial to realize our approach because of two challenges: (1) How to identify the authentication data structure in contract bytecode. Since the token contract is Turing complete and most contracts are not open source. (2) How to infer authentication behavior. The bytecode of the smart contract has no type information.

To address these challenges, we use symbolic execution to analyze token authentication behaviors. Specifically, we first extracted the operating characteristics of authentication data structure during contract execution. Next, we mapped these operating characteristics into symbolic expressions to identify the authentication data structure. Then, we still utilize symbolic expressions to infer the authentication behavior. Finally, we use the difference between the target authentication behaviors and standard authentication behaviors to detect authentication defects.

We implement our approach in a new tool named `TokenCat` and inspect 245,822 token contracts from xblock-eth [6]. `TokenCat` reported 522 token contracts with authentication flaws. After manually analyze, we found that there are 491 token contracts with flaws in the authentication and thus the precision of `TokenCat` is 94% = 491/522. Furthermore, we reveal four types of flawed ERC20 token authentication implementations (e.g., unreasonable approval authorization and unreasonable authorized transfer) in §V-D.

In summary, this work has two major contributions.

- To best our knowledge, this is the first work to detect the flaws in the implementation of authentication in ERC20 token. To automate the analysis, we implement our approach in a new tool named `TokenCat`.
- `TokenCat` detects 245,822 tokens and finds 491 ERC20 token with authentication implementation flaws. The token dataset, flawed token list and the source code of `TokenCat` are available at https://github.com/hzysvilla/TokenCat/.

II. BACKGROUND

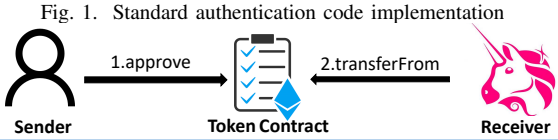
EVM (Ethereum Virtual Machine). EVM is a Turing-complete state machine, including the accounts state model of Ethereum [1], and incrementally executes the transactions on blockchain to morph state. The program executed on EVM with Turing completeness is the so-called smart contract. Smart contracts are usually written in high-level programming language (e.g., solidity [7]), then compiled into bytecode, and finally executed in the EVM. Smart contracts also have a persistent storage a key-value store with 256-bits keys and 256-bits values [8].

Token. A token is a smart contract which records the information of token holders and their shares, and supports token activities, e.g., query the balance of a token holder, transfer tokens to another holder [2].

Token Authorized Transfer. ERC20 authentication is the process by which users invoke the authorization method to grant third parties the authority to use tokens on user behalf. As usual, the authentication information is recorded in a two-dimensional mapping data structure and we will discuss other two types of data structure forms in §V-C. In Fig. 1 line2, the authentication information record represented by the allowance variable. According to the ERC20 API definition, the `approve()` method (in Fig. 1 line 3 to 6) allows a spender (e.g., user, wallet, exchange, etc.) to withdraw up to an allowed amount of tokens from token pool of the approver [9]. The `transferFrom()` method (in Fig. 1 line 7 to 15) allows the spender to actually transfer tokens from the approver to anyone they choose [9]. A complete authorization transfer process is shown in Fig. 2. The specific process in Fig. 2 is as follows:

```
1 mapping(address => uint256) balances;
2 mapping (address => mapping (address => uint256)) allowance;
3 function approve(address _spender, uint256 _value) public{
4     allowance[msg.sender][_spender] = _value;
5     emit Approval(msg.sender, _spender, _value);
6 }
7 function transferFrom(address _from, address _to, uint256 _value) public{
8     require(_value <= balances[_from]);
9     require(_value <= allowance[_from][msg.sender]);
10
11    balances[_from] = balances[_from] - _value;
12    balances[_to] = balances[_to] + _value;
13    allowance[_from][msg.sender] = allowance[_from][msg.sender] - value;
14    emit Transfer(_from, _to, _value);
15 }
```

Fig. 1. Standard authentication code implementation



Statement Change Situation		
Steps	Balance	Allowance
0.initial	(Sender:100, Reciever:0)	allowance[Sender][Receiver]=0
1.approve(Receiver, 100)	(Sender:100, Receiver:0)	allowance[Sender][Receiver]=100
2.transferFrom(Sender, Receiver, 70)	(Sender:30, Receiver:70)	allowance[Sender][Receiver]=30

Fig. 2. The process of token authorized transfer

1) The sender allows receiver to transfer 100 tokens on sender behalf by invoking `approve(Receiver, 100)`. The allowance variable is assigned a value of 100.
2) The receiver uses the authentication to transfer the sender's 70 tokens by invoking `transfer(Sender, Receiver,70)` The allowance variable is deducted by 70, leaving only 30.

Symbolic Expression. The symbolic execution is a method of program analysis. The symbolic expression refers to that when an instruction depends on at least one external runtime input during symbolic execution, a symbolic expression will be constructed to describe the result of an instruction.

III. DEFINITION

Authentication Data Structure. The role of the authentication data structure is to record the user's authentication information. We use $\Im$ to represent the authentication data structure. The $\Im$ is defined in the following form:

$$< addr_1, < addr_2, amount >> \quad (1)$$

where $addr_1$ represents the address of authorizer and the token holder, $addr_2$ represents the address of authorized third party, and $amount$ represents the authorized amount. $\Im$ is authentication record which $addr_1$ authorizes $addr_2$ to transfer a token with a value (i.e., $amount$) from the account of $addr_1$.

Standard Authentication Behavior. We define the standard authentication behaviors into two types. The first type is approval authorization and the second type is using authorization transfer. In Fig. 3, we explain the definition of approval

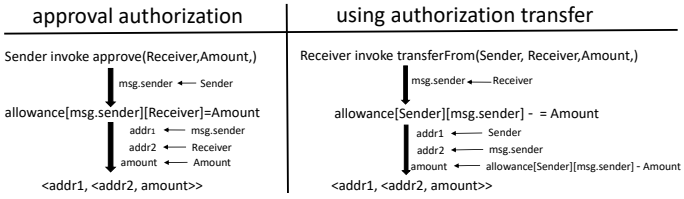


Fig. 3. The definition of standard authentication behavior.

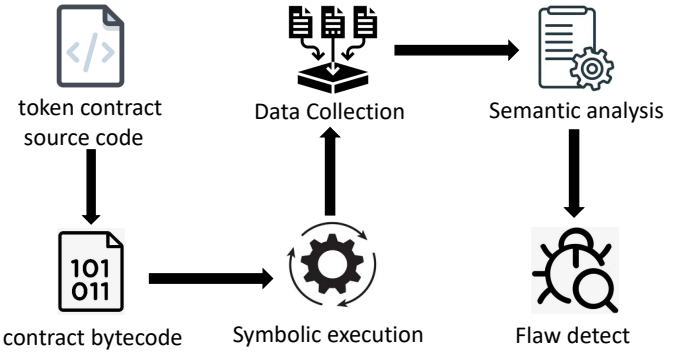


Fig. 4. The archtecture of TokenCat

authorization. In this type, the $addr_1$ in Eq. 1 is limited to `msg.sender`, so that the caller can only access his own authentication information (i.e., $\Im$). The `msg.sender` is a built-in global variable used to indicate the initiator of the transaction, `msg.sender` represent the token holder and authorizer. The $addr_2$ and $amount$ are given by the caller as method parameters. We also give the specific implementation of approval authorization in `approve()` method (Fig. 1 line 3 to 6).

In Fig. 3, we also described the definition of using authorization transfer. In this type, $addr_2$ in Eq. 1 is limited to `msg.sender`, $addr_1$ is given by the caller as a method parameter. At this time, the $amount$ is updated by subtracting the parameter `Amount` provided by the caller from the previous $amount_{old}$ value. The `transferFrom()` method (in Fig. 1 line 7 to 15) is used to implement current type.

Defect authentication behavior. We define defect authentication behavior in two aspects. The first is that the third party illegally tampered with the user's authentication record. The second is that the changed value of the authentication data structure is inconsistent with the user's expectations.

IV. TokenCat

In this section, we'll provide the detailed design of the `TokenCat`.

A. Overview

`TokenCat` contains three stages (described in Fig. 4). At the first stage, `TokenCat` utilizes the symbolic execution tool Mythril [10] to collect data. In the second stage, `TokenCat` analyzes the data to identify authentication data structure (i.e., $\Im$) from bytecode. In the third stage, when $\Im$ is identified in the previous stage, `TokenCat` infer authentication behaviors and detect authentication flaws.

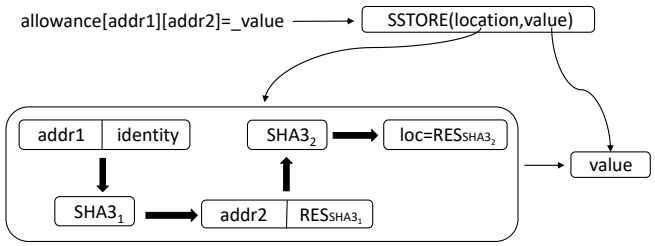


Fig. 5. The process of writintg to authentication data structure

B. Stage1: Data Collect

In first phase, `TokenCat` needs to collect the symbolic expressions of location and value of `SSTORE` instruction to preliminary capture the authentication behavior. Because $\Im$ in persistent storage (described in §II) is modified by `SSTORE` instruction, `TokenCat` collected data on `SSTORE` instruction. We let $location_{se}$ and $value_{se}$ to denote the location and value symbolic expression of the `SSTORE` instruction.

C. Stage2: Identify Authentication Data Structure

In this section, we recognize $\Im$ through addressing mode from $location_{se}$. The addressing mode is a series of EVM instruction operations to calculate the address index in persistent storage to locate $\Im$.

In Fig. 5, we explain the specific process of addressing mode. First, `addr1` and `identity` are spliced together, and the `SHA3` instruction is used to perform hash operations to get the `RES`, where `identity` is a number assigned by the compiler to identify variable corresponding to $\Im$. Next, `addr2` and `RES` are spliced together, and the second `SHA3` instruction is performed to obtain the address index of the $\Im$. Finally, the `SSTORE` instruction will write the value with address index in persistent storage. From the above process, We can get $location_{pattern}$ which represent address index composed of the EVM operations on the following form:

$$location_{pattern} = SHA3_2(addr_2, SHA3_1(addr_1, identity)) \tag{2}$$

We next explain why `TokenCat` can identify $\Im$ by the symbolic expression of the `SSTORE` instruction. $location_{se}$ has the following form:

$$\begin{aligned} location_{se} =& keccak256\_512(Concat(\\ & addr_2, keccak256\_512(Concat(addr_1, identity)) \\ &)) \end{aligned} \tag{3}$$

In Mythril's symbolic expressions [10], the two symbolic functions `keccak256_512` and `Concat` are designed to simulate `SHA3` instructions. Therefore, in Fig. 6, we find that the $location_{pattern}$ of the authentication structure can be directly abstracted into a symbolic expression $location_{se}$. Because the `SSTORE` instruction accessing $\Im$ relies on external input, these external inputs are used as symbols to construct symbolic expressions (i.e., $location_{se}$). Therefor,the $location_{se}$ already contain clear operation information. Only by parsing $location_{se}$, `TokenCat` can get the addressing mode to identify the $\Im$.

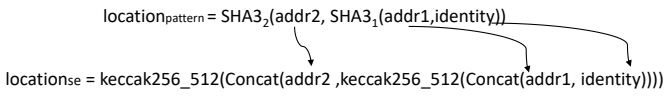


Fig. 6. Symbolic expression abstract process

Algorithm 1 symbolic expression parse algorithm

Input: Symbolic Expression of location: $location_{se}$
Terminal_Symbols:{addr1,addr2,identity}
NonTerminal_Symbol:{keccak256_512,Concat}
Output: Tree of authentication data structure

```
Tree_se = newTree()
for each lexeme in scan(location_se) do
switch (lexeme)
case Nonterminal_Symbol:
   Tree_se.add_NoneLeaf_Node()
case Terminal_Symbol:
   Tree_se.add_Leaf_Node()
case ( :
   Tree_se.next_Layer()
case ) :
   Tree_se.last_Layer()
```

After we get $location_{se}$ of the `SSTORE` instruction, we'll parse $location_{se}$ into a tree structure $Tree_{se}$ for matching authentication data structure (i.e., $\Im$). The reason why `TokenCat` converts the $location_{se}$ to $tree_{se}$ is the tree structure can express code logic clearly and universally. Based on $tree_{se}$, we can analyze more data structure types (e.g., balance) in future work. We represent the parsing process by Algorithm 1 and give a standard $Tree_{se}$ of $\Im$ in Fig. 7. The `scan()` method divides $location_{se}$ into lexical units based on commas and brackets and traverses these lexical units. `add_NoneLeaf_Node()` method is to add non-leaf nodes to $Tree_{se}$. Non-leaf nodes include two types: `kecak256_512` and `Concat`. The method of `add_Leaf_Node()` is to add a leaf node, usually a leaf node contains $addr_1$, add_2, and $identity$. `next_Layer()` and `last_Layer()` are used to control the layer of $Tree_{se}$ where the currently added node is located. After obtaining $Tree_{se}$ of $location_{se}$, we'll compare the target $Tree_{se}$ with the tree of the standard authentication structure (described in Fig. 7) by traversing every symbol. If it is consistent, `TokenCat` has already identified $\Im$.

D. Stage3: Authorized Transfer Behavior Inference

After located $\Im$, `TokenCat` will infer the behavior of authentication and inspect the authentication defects by $value_{se}$. `TokenCat` accomplish the above goals through Algorithm 2. In Algorithm 2, `judgment_Type()` method first distinguishes the two authentication behaviors (described in §III) by the

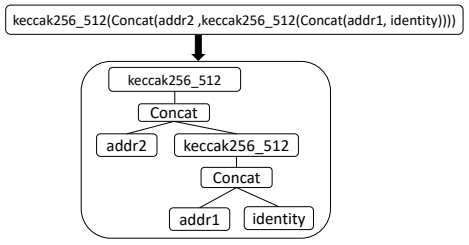


Fig. 7. The $Tree_{se}$ of standard authentication data structure

Algorithm 2 Analysis of authentication behavior

Input: The Tree of location: $Tree_{se}$
Symbolic expression of value: $value_{se}$
Output: Authentication behavior detect results

```
type=judgment_Type(Tree_se)
if type = approval authorization then
   if approve_behavior_detect(value_se) then
    return //correct authentication
   else
    flaw_Detect()
   end if
else if type == use authorization then
   if usingAuthorization_behavior_detect(value_se) then
    return //correct authentication
   else
    flaw_Detect()
   end if
else
    flaw_Detect()
```

access method to $\Im$. When the authorization is approved, $addr_1$ of $\Im$ should be $msg.sender$ and $addr_2$ of $\Im$ is a method parameter set by the caller. As for using authorization, when a third party uses authorized transfers, $addr_1$ of $\Im$ should be a method parameter set by the caller and $addr_2$ of $\Im$ should be $msg.sender$. If it does not belong to these rules, this means that the attacker may be able to access the user's authentication record and `TokenCat` will report flaw by `flaw_Detect()` method.

Next, the `approve_behavior_detect()` method infers the behavior of approval authorization. In Fig. 3, we can see that the approval authentication behavior directly assigns the parameter set by the caller to $\Im$. In other words, $value_{se}$ should be the parameter given by the caller. Because it is a single simple symbol, we can directly analyze it by pattern matching. If it fails to match, it means that the authorized amount is inconsistent with the amount given by the user and `TokenCat` will report it.

Finally, the `usingAuthorization_behavior_detect()` method is used to analyze the behavior the using authorization transfer (defined in Fig. 3). There are two difficulties to analyze behavior in such type. (1) The read operation (to get $amount_{old}$) causes the $value_{se}$ to contain complex persistent storage symbolic expression, which makes $value_{se}$ difficult to parse. (2) Operator optimization problem. Because the compiler may optimize the subtraction into complex bit operations and it'll mislead pattern matching to produce false positives.

`TokenCat` handles the first challenge by simplifying $value_{se}$. Specifically, we abstract the symbolic expression of persistent storage as $Storage_{se}$ and we can get the following expression of $value_{se}$:

$$Storage_{se}[location_{se}] - \_Amount_{se}$$

`TokenCat` performs pattern matching on $value_{se}$. If the match fails, `TokenCat` needs to perform further analysis and would face the second difficulty. Under current circumstances, `TokenCat` uses z3 SMT (Satisfiability modulo theories) solver [11] by constructing conditional constraints to tackle this

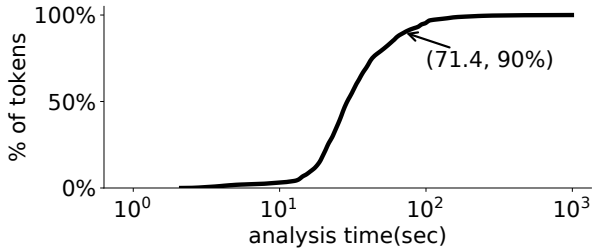


Fig. 8. CDF of analysis time per contracts

problem. Specifically, TokenCat constructs the symbolic expression according to the semantics of standard authentication behavior, and then compares the standard authentication expression with $value_{se}$ to obtain the following conditional constraints:

$$(value_{se}) \neq (OriginalStorage_{se}[location_{se}] - \_Amount_{se})$$

$OriginalStorage_{se}$ represents the original expression of Storage without abstraction. If the above condition has a solution by z3 solver, It means that the actual value of the authentication data structure is inconsistent with the standard authentication behavior. Therefore, the authentication implementation of the token is flawed.

V. EXPERIMENTS AND ANALYSIS

To evaluate the efficiency and precision of TokenCat, we run TokenCat on a large-scale dataset about 245,822 tokens. Then, we analyze the results and summarize four types of ERC20 token contracts with authentication flaws. Our results include both closed source contracts and open source contracts. For closed-source contracts, we first check with third-party reverse tools (i.e., Gigahorse [12]), and then analyze them through manual debugging. For open-source contracts, we audit source code manually.

A. Dataset

The dataset we termed by *D*1, including 245,822 token contract bytecodes collected from 0 to 12M block height by xblock-eth [6]. After removing the duplicate bytecode from *D*1, we get the *D*2 dataset which contains 190,495 token contracts.

B. Efficiency of TokenCat

We conduct time efficiency experiments on *D*2, using Intel W-3275M CPU and 32GB memory. The contract analysis time ranges from 2.1s to 992.7s, with an average value of 39.2s. Fig. 8 is a CDF (cumulative distributed method) diagram, which shows that 90% of the smart contract analysis time will not exceed 71.4s.

C. Precision of TokenCat

To verify the precision of TokenCat, we manually audit 522 token contracts detected by TokenCat, and found 31 tokens of false positives, so the precision is 94% = $(522-31)/522$. To our knowledge, no ground truth dataset exists that represents tokens with authentication defect, so we can't get the false positive results of TokenCat.

Among the 31 false positive tokens, 20 are mainly because the token is composed of multiple contracts and the symbolic execution tool cannot obtain the bytecode of all contracts. Therefore TokenCat cannot generate the symbolic expression to identify authentication data structure. The reason for the remaining 11 false positives is that there are variables in the contract that define the same type as the authentication data structure. These variables are incorrectly identified as authentication data structures. TokenCat takes all the variables successfully identified in the standard ERC20 approve() method as candidate authentication data structures. Although this strategy caused 2% (11/522) false positives, it is necessary to ensure completeness.

TABLE I
The number of different authentication defect types

Type	Name	# of tokens
I	Unreasonable Approval Authorization	60
II	Unreasonable Authorized Transfer	50
III	Inconsistent Authentication	234
IV	Unlimited Authorization	147

We also found two other types of authentication implementations. In the implementation of some token contracts, one-dimensional mapping is used as $\Im$, but one-dimensional mapping (e.g., $< addr, value >$) cannot associate the information of the authorizer and the authorized person. This kind of implementation is inherently flawed. There is also an $\Im$ that uses two-dimensional mapping to point to structure type. TokenCat can handle these two types of authentication data structures well.

D. Analysis of Defect Authentication Types

We divide the results of TokenCat detection into the following four categories. Table I lists the numbers of different types.

Type-I: Unreasonable Approval Authorization (60 tokens). Because of unreasonable approval authentication implementation in Fig. 9, the $addr_1$ in authentication data structure can be set by the user. The attacker can directly manipulate authentication record to steal the user's token. Specifically, the attacker first invokes the flawed approval authorization method to access the authentication record, allowing the victim to authorize the attacker to transfer money. Next, the attacker can transfer the balance from victim's account. There are two reasons for this authentication defect. For the implementation in Fig. 9(a), it may be caused by a bug, because anyone can access it. But for the type in Fig. 9(b) and Fig. 9(c), it might belong to the backdoor, because only the specific user can exploit such defective authentication.

Type-II: Unreasonable Authorized Transfer (50 tokens). When using authentication to transfer token, both $addr_1$ and $addr_2$ of the authentication structure (described as Eq. 1) can be controlled by the caller, as we show in Fig. 10. As a result, the attacker can initiate a token transfer that can only be initiated by an authorized third party. The DeFi project usually requires unlimited authorization from the user [13]. After obtaining unlimited authorization, the attacker can transfer the victim's token to the DeFi project with extremely fluctuating

```
function approve1(address _owner, address _spender,
 uint256 _value) public{
    allowance[_owner][_spender] = _value;
    emit Approval( _owner, _spender, _value);
}
```

(a) Source code1

```
address creator3=0x12345678912345678900
function approve3(address _spender, uint256 _value)
public{
    require(msg.sender==creator3)
    allowance[_spender][msg.sender] = _value;
    emit Approval(_spender, msg.sender, _value);
}
```

(b) Source code2

```
function transferFrom(address _from, address _to,
    uint256 _value) public{
    if (msg.sender == creator){
        allowance[_from][msg.sender] = _value
    }
    require(_value <= balances[_from]);
    require(_value <= allowance[_from][msg.sender]);
    balances[_from] = balances[_from] - _value;
    balances[_to] = balances[_to] + _value;
    allowance[_from][msg.sender] = allowance[_from][msg.sender] - value;
    emit Transfer(_from, _to, _value);
}
```

(c) Source code3

Fig. 9. A contract code snippet of type-I

```
function transferFrom( address from, address to,
uint256 _value) public{
   require(_value <= balances[_from]);
   require(_value <= allowance[_from][to]);

   balances[from] = balances[from] - _value;
   allowance[from][to] = allowance[from][to] - _value;
   balances[to] = balances[to] + tokens;
   emit Transfer( from, to, tokens);
}
```

Fig. 10. A contract code snippet of type-II

prices at any time. The attackers can carry out DeFi arbitrage attacks [14] and user will suffer huge losses.

Type-III: Inconsistent Authentication (234 tokens). Inconsistent authentication means inconsistency between the method parameters set by the user and the actually changed value of authentication data structure. On Fig. 11, the user wants to authorize $\_value$, but the actual value of the authentication data structure changed is $\_value.mul(multiplier)$. Such authentication defect will make user confusion and cause token loss.

Type-IV: Unlimited Authorization (147 tokens). The implementation of this kind of authentication will induce users to perform unlimited authentication. In this type, the user cannot choose the amount of authorization, and the token contract directly allows the user to grant unlimited authorization to a third party through hard coding, as we show in Fig. 12. As mentioned in the research of blocksec [13], when there is a problem with the authorized party, it will bring great risks to

```
function approve(address _spender, uint256 _value) {
    allowed[msg.sender][_spender] = _value.mul(multiplier);
    Approval(msg.sender, _spender, _value.mul(multiplier));
    return true;
}
```

Fig. 11. A contract code snippet of type-III

```
function approve(address _spender) public{
    allowance[msg.sender][_spender] = -1;
    //-1 indicates the largest integer in uint type
    emit Approval(msg.sender, _spender, -1);
}
```

Fig. 12. A contract code snippet of type-IV

the user's asset.

VI. CONCLUSION AND FUTURE WORK

In this work, we detected the authentication implementation of defective ERC20 tokens, which has not been done before. We implemented a new tool called `TokenCat` to automatically detect such flaws. We inspected 245,822 tokens and detected 491 flawed tokens with precision of 94% all proved the effectiveness of `TokenCat`. Moreover, we reveal four types of flawed ERC20 token authentication implementations.

In future work, we will combine the transaction information on Ethereum to study the defect authentication behavior that has been triggered.

VII. ACKNOWLEDGMENT

This research is partially supported by National Natural Science Foundation of China (61872057), National Key R&D Program of China (2018YFB0804100), National Natural Science Foundation of China (No.U19A2066), and Sichuan Science and Technology Program under Grants 2020JDTD0007.

REFERENCES

[1] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, 2018.

[2] T. Chen, Y. Zhang, Z. Li, X. Luo, T. Wang, R. Cao, X. Xiao, and X. Zhang, "Tokenscope: Automatically detecting inconsistent behaviors of cryptocurrency tokens in ethereum," in *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, 2019, pp. 1503–1520.

[3] W. Chen, T. Zhang, Z. Chen, Z. Zheng, and Y. Lu, "Traveling the token world: A graph analysis of ethereum erc20 token ecosystem," in *Proceedings of The Web Conference 2020*, 2020, pp. 1411–1421.

[4] bancor.network, "primitive," https://docs.primitive.finance/, 2021.

[5] A. Group, "Exploiting Primitive Finance's Approval Flaws," https://medium.com/amber-group/exploiting-primitive-finances-approval-flaws-b86db031b4, 2020.

[6] P. Zheng, Z. Zheng, J. Wu, and H.-N. Dai, "Xblock-eth: Extracting and exploring blockchain data from ethereum," *IEEE Open Journal of the Computer Society*, vol. 1, pp. 95–106, 2020.

[7] Ethereum, "Solidity documentation," https://docs.soliditylang.org/, 2020.

[8] J. Krupp and C. Rossow, "teether: Gnawing at ethereum to automatically exploit smart contracts," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 1317–1333.

[9] R. Rahimian, S. Eskandari, and J. Clark, "Resolving the multiple withdrawal attack on erc20 tokens," in *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, 2019, pp. 320–329.

[10] B. M. [n. d.]., "Mythril." https://github.com/ConsenSys/mythril, 2021.

[11] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[12] DEDAUB, "Contract library," https://contract-library.com/, 2019.

[13] BlockSec, "Tradeoff between convenience and security: Unlimited approval in erc20," in *Def Con*, 2021. [Online]. Available: https://blocksecteam.medium.com/unlimited-approval-in-erc20-convenience-or-security-1c8dce421ed7

[14] L. Zhou, K. Qin, A. Cully, B. Livshits, and A. Gervais, "On the just-in-time discovery of profit-generating transactions in defi protocols," in *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 2021, pp. 919–936.