

SMART CONTRACT AUDIT REPORT

for

MAKER FOUNDATION

Prepared By: Shuxiao Wang

Hangzhou, China Oct 4, 2019

Document Properties

Client	Maker Foundation
Title	Smart Contract Audit Report
Target	Multi-Collateral Dai (MCD)
Version	1.0
Author	Chiachih Wu, Xuxian Jiang
Auditors	Chiachih Wu, Xuxian Jiang
Reviewed by	Xuxian Jiang
Approved by	Jeff Liu
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0	Oct 4, 2019	Chiachih Wu, Xuxian Jiang	Final Release
0.4	Sep 09, 2019	Chiachih Wu, Xuxian Jiang	More Findings Added
0.3	Sep 03, 2019	Chiachih Wu, Xuxian Jiang	More Findings Added
0.2	Aug 23, 2019	Chiachih Wu, Xuxian Jiang	Findings Added
0.1	Aug 21, 2019	Chiachih Wu	Initial Draft

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1	Intro	oduction	5
	1.1	About Multi-Collateral Dai (MCD)	5
	1.2	About PeckShield	6
	1.3	Methodology	6
	1.4	Disclaimer	7
2	Find	lings	10
	2.1	Summary	10
	2.2	Key Findings	11
3	Deta	ailed Results	12
	3.1	Potential Denial-of-Service in Global Settlement	12
	3.2	Potential Divide-By-Zero in Spotter	15
	3.3	Inconsistent Time Type in Debt Engine	16
	3.4	approve()/transferFrom() Race Condition	17
	3.5	Unhandled Auction Corner Cases	18
	3.6	CDP Fork Restrictiveness	20
	3.7	Drip Efficiency Improvement	21
	3.8	Debt Auction Prevention	23
	3.9	Misadjusted CDP Cancellation	24
	3.10	Auction Kick-Off Authorization	25
	3.11	Auction Tick Validity	28
	3.12	Auction Deal Inconsistency	29
	3.13	Bloated Setter Interface	31
	3.14	Missed Deployment Dependency Checks	32
	3.15	Excessive Authorization in Deployment	37
	3.16	Collateral Check in MCD CDP Manager	39
	3.17	Other Suggestions	41
4	Con	clusion	42

5	Арр	endix		43
	5.1	Basic (Coding Bugs	43
		5.1.1	Constructor Mismatch	43
		5.1.2	Ownership Takeover	43
		5.1.3	Redundant Fallback Function	43
		5.1.4	Overflows & Underflows	43
		5.1.5	Reentrancy	44
		5.1.6	Money-Giving Bug	44
		5.1.7	Blackhole	44
		5.1.8	Unauthorized Self-Destruct	44
		5.1.9	Revert DoS	44
		5.1.10	Unchecked External Call	45
		5.1.11	Gasless Send	45
		5.1.12	Send Instead Of Transfer	45
		5.1.13	Costly Loop	45
		5.1.14	(Unsafe) Use Of Untrusted Libraries	45
		5.1.15	(Unsafe) Use Of Predictable Variables	46
		5.1.16	Transaction Ordering Dependence	46
		5.1.17	Deprecated Uses	46
	5.2	Seman	tic Consistency Checks	46
	5.3	Additio	nal Recommendations	46
		5.3.1	Avoid Use of Variadic Byte Array	46
		5.3.2	Use Fixed Compiler Version	47
		5.3.3	Make Visibility Level Explicit	47
		5.3.4	Make Type Inference Explicit	47
		5.3.5	Adhere To Function Declaration Strictly	47

References

1 Introduction

Given the opportunity to review the Multi-Collateral Dai (MCD) design document and related smart contract source code, we in the report outline our systematic method to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistency between smart contract code and the white paper, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Multi-Collateral Dai (MCD)

Multi-Collateral Dai (MCD) is a modular system of inter-dependent smart contracts developed for the Ethereum blockchain. An off-chain system of oracles is used to supply price data on which the system relies. The core system of permissioned modules is maintained by MKR governance, with updates being executed via approval voting. Non-permissioned front-ends such as the CDP Manager and SCD-MCD Migrator provide convenience for CDP operators and Dai holders.

The basic information of Multi-Collateral Dai (MCD) is as follows:

ltem	Description
lssuer	Maker Foundation
Website	https://makerdao.com
Туре	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	Oct 4, 2019

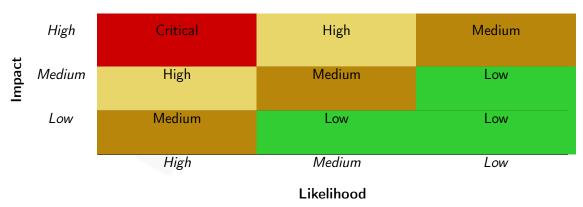
Table 1.1:	Basic Information of Multi-Collateral Dai (MCD)
------------	---	------

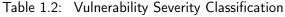
In the following, we show the Git repository of reviewed files and the commit hash value used in this audit:

- https://github.com/makerdao/dss.git (526fa6a)
- https://github.com/makerdao/dss-deploy.git (ec9a414)
- https://github.com/makerdao/median.git (d95bbc1)
- https://github.com/makerdao/oracles-v2.git (a216cd0)
- https://github.com/makerdao/dss-cdp-manager.git (c11ec39)
- https://github.com/makerdao/scd-mcd-migration.git (4f7030c)

1.2 About PeckShield

PeckShield Inc. [28] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystem by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).





1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [23]:

- <u>Likelihood</u> represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;

• Severity demonstrates the overall criticality of the risk;

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- <u>Basic Coding Bugs</u>: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- <u>Semantic Consistency Checks</u>: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- <u>Advanced DeFi Scrutiny</u>: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- <u>Additional Recommendations</u>: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [22], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some catogories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this audit does not give any warranties on finding all possible security issues of the given smart contract(s), i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as an investment advice.

Category	Check Item	
	Constructor Mismatch	
	Ownership Takeover	
	Redundant Fallback Function	
	Overflows & Underflows	
	Reentrancy	
	Money-Giving Bug	
	Blackhole	
	Unauthorized Self-Destruct	
Basic Coding Bugs	Revert DoS	
Dasic Couling Dugs	Unchecked External Call	
	Gasless Send	
	Send Instead Of Transfer	
	Costly Loop	
	(Unsafe) Use Of Untrusted Libraries	
	(Unsafe) Use Of Predictable Variables	
	Transaction Ordering Dependence	
	Deprecated Uses	
Semantic Consistency Checks	Semantic Consistency Checks	
	Business Logics Review	
	Functionality Checks	
	Authentication Management	
	Access Control & Authorization	
	Oracle Security	
Advanced DeFi Scrutiny	Digital Asset Escrow	
	Kill-Switch Mechanism	
	Operation Trails & Event Generation	
	ERC20 Idiosyncrasies Handling	
	Frontend-Contract Integration	
	Deployment Consistency	
	Holistic Risk Management	
	Avoiding Use of Variadic Byte Array	
	Using Fixed Compiler Version	
Additional Recommendations	Making Visibility Level Explicit	
	Making Type Inference Explicit	
	Adhering To Function Declaration Strictly	
	Following Other Best Practices	

Table 1.3:	The Full	List of	Check	Items
------------	----------	---------	-------	-------

Category	Summary		
Configuration	Weaknesses in this category are typically introduced during		
	the configuration of the software.		
Data Processing Issues	Weaknesses in this category are typically found in functional-		
	ity that processes data.		
Numeric Errors	Weaknesses in this category are related to improper calcula-		
	tion or conversion of numbers.		
Security Features	Weaknesses in this category are concerned with topics like		
	authentication, access control, confidentiality, cryptography,		
	and privilege management. (Software security is not security		
	software.)		
Time and State	Weaknesses in this category are related to the improper man-		
	agement of time and state in an environment that supports		
	simultaneous or near-simultaneous computation by multiple		
	systems, processes, or threads.		
Error Conditions,	Weaknesses in this category include weaknesses that occur if		
Return Values,	a function does not generate the correct return/status code,		
Status Codes	or if the application does not handle all possible return/status		
	codes that could be generated by a function.		
Resource Management	Weaknesses in this category are related to improper manage-		
	ment of system resources.		
Behavioral Issues	Weaknesses in this category are related to unexpected behav-		
	iors from code that an application uses.		
Business Logics	Weaknesses in this category identify some of the underlying		
	problems that commonly allow attackers to manipulate the		
	business logic of an application. Errors in business logic can		
Initialization and Cleanup	be devastating to an entire application.		
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.		
Arguments and Parameters	Weaknesses in this category are related to improper use of		
Arguments and Parameters	arguments or parameters within function calls.		
Expression Issues	Weaknesses in this category are related to incorrectly written		
Coding Practices	expressions within code. Weaknesses in this category are related to coding practices		
	that are deemed unsafe and increase the chances that an ex-		
	ploitable vulnerability will be present in the application. They		
	may not directly introduce a vulnerability, but indicate the		
	product has not been carefully developed or maintained.		
	product has not been calefully developed of maintained.		

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used In This Audit

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing Multi-Collateral Dai (MCD) implementation. During the first phase of our audit, we studied the MCD source code and ran our in-house static code analyzer through the codebase, including areas such as ERC20 tokens, CDPs, Dai Saving Rate (DSR), auctions, permissions, price oracle, and inter-contract actions. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity		# of Findings	
Critical	0		
High	1		
Medium	1		
Low	3		
Informational	11		
Total	16		

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined that 16 issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved with the identified issues, including 1 high-severity vulnerability, 1 medium-severity vulnerability, 3 low-severity vulnerabilities and 11 informational recommendations, as shown in Table 2.1.

ID	Severity	Title	Category	Status
PVE-001	Informational	Potential DoS in Global Settlement	Business Logics	Resolved
PVE-002	Informational	Potential Divide-By-Zero in Spotter	Numeric Errors	Confirmed
PVE-003	Informational	Inconsistent Time Type in Debt Engine	Coding Practices	Resolved
PVE-004	Low	approve()/transferFrom() Race Condition	Time and State	Confirmed
PVE-005	Informational	Unhandled Auction Corner Cases	Business Logics	Confirmed
PVE-006	Informational	CDP Fork Restrictiveness	Coding Practices	Confirmed
PVE-007	Informational	Drip Efficiency Improvement	Coding Practices	Confirmed
PVE-008	Informational	Debt Auction Prevention	Business Logics	Resolved
PVE-009	Medium	Misadjusted CDP Cancellation	Behavioral Issues	Resolved
PVE-010	High	Auction Kick-Off Authorization	Business Logics	Resolved
PVE-011	Informational	Auction Tick Validity	Time and State	Confirmed
PVE-012	Low	Auction Deal Inconsistency	Business Logics	Resolved
PVE-013	Informational	Bloated Setter Interface	Coding Practices	Resolved
PVE-014	Informational	Missed Deployment Dependency Checks	Coding Practices	Confirmed
PVE-015	Low	Excessive Authorization in Deployment	Security Features	Resolved
PVE-016	Informational	Collateral Check in MCD CDP Manager	Coding Practices	Confirmed

Table 2.1: Key Audit Findings

Please refer to Section 3 for details.

3 Detailed Results

3.1 Potential Denial-of-Service in Global Settlement

- ID: PVE-001
- Severity: Informational
- Likelihood: High
- Impact: None

- Target: src/end.sol
- Category: Business Logics [20]
- CWE subcategory: CWE-754 [14]

Description

The sixth step of 'End', thaw(), is vulnerable to a potential denial-of-service attack when vat.dai(address(vow))> 0. Specifically, throughout the five steps before thaw(), only the cage() step subtracts vat.dai(address(vow)) by calling vow.cage() which calls vat.heal(). However, in the condition of vat.dai(address(vow))> vat.sin(address(vow)), vat.dai(address(vow)) would be greater than 0 after vow.cage(). This results in the termination of thaw() due to the vat.dai(address(vow))== 0 check. Another easier way to exploit this vulnerability is simply calling vat.move() to add some dai balance to vat.dai(address(vow)) right before thaw() is executed.

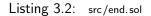
Specifically, while auditing the thaw() of end.sol, we find line 313 is vulnerable to a potential denial-of-service attack when the attacker intentionally makes vow have some dai balance in vat.

```
310 function thaw() external note {
311     require(live == 0);
312     require(debt == 0);
313     require(vat.dai(address(vow)) == 0);
314     require(now >= add(when, wait));
315     debt = vat.debt();
316 }
```

Listing 3.1: src/end.sol

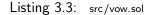
In particular, if we examine the first five steps before thaw() in end.sol to find out where vat.dai (address(vow)) is modified. In the first step, cage(), vow.cage() is invoked in line 259.

```
253 function cage() external note auth {
254    require(live == 1);
255    live = 0;
256    when = now;
257    vat.cage();
258    cat.cage();
259    vow.cage();
260 }
```



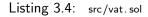
Inside vow.cage(), vat.heal() is called with the smaller value between vat.dai(address(vow)) and vat.sin(address(vow)) as the parameter (line 138).

```
132 function cage() external note auth {
133     live = 0;
134     Sin = 0;
135     Ash = 0;
136     flapper.cage(vat.dai(address(flapper)));
137     flopper.cage();
138     vat.heal(min(vat.dai(address(this)), vat.sin(address(this))));
139 }
```



Here comes the interesting part. In vat.heal(), if vat.dai(address(vow))> vat.sin(address(vow)), vat.dai(address(vow)) would be greater than 0 (line 241).

```
238
    function heal(uint rad) external note {
239
        address u = msg.sender;
240
        sin[u] = sub(sin[u], rad);
241
        dai[u] = sub(dai[u], rad);
242
               = sub(vice,
        vice
                              rad);
243
        debt
               = sub(debt,
                              rad);
244 }
```



In the following steps, cage(ilk), skip(), skim(), free(), vat.dai(address(vow)) is NOT subtracted but only added (e.g., skip() calls vat.suck()). As a result, the vat.dai(address(vow))> 0 condition would not be changed until reaching thaw(), which leads to the termination of 'End' process.

Alternatively, another easier way to exploit this vulnerability is calling vat.move(msg.sender, vow, 1) in the case of vat.dai[msg.sender] >= 1 before thaw() is triggered. Specifically, when the attacker has vat.dai[msg.sender] >= 1, she can easily move 1 dai balance to vow in vat (line 146-147). The only thing prevents this is the require(wish(src, msg.sender)) check in line 145.

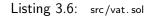
```
144 function move(address src, address dst, uint256 rad) external note {
145 require(wish(src, msg.sender));
146 dai[src] = sub(dai[src], rad);
147 dai[dst] = add(dai[dst], rad);
```

148 }

Listing 3.5: src/vat.sol

However, the check can be easily bypassed when src = msg.sender (i.e., the bit == usr case in line 31).

```
30 function wish(address bit, address usr) internal view returns (bool) {
31 return either(bit == usr, can[bit][usr] == 1);
32 }
```



To sum up, either the system is in the condition of vat.dai(address(vow))> vat.sin(address(vow)) or the attacker intentionally calls vat.move(msg.sender, vow, 1) before thaw() is executed, the 'End' process might not be as smooth as expected because of this issue.

Fortunately, such attack can be effectively alleviated by calling vow.heal() right before the thaw() within the same transaction. In particular, the keeper responsible for the 'End' process needs to call vow.heal() right before the thaw() and needs to call it in the same transaction. This effectively becomes an operation issue that should be kept in mind when initiating the 'End' process. While we consider the severity lowered to be informational, it is strongly recommended to add this issue into related memos for proper 'End' operations.

Recommendation Alleviate the issue by combining a vow.heal() call in the same transaction as the thaw().

3.2 Potential Divide-By-Zero in Spotter

- ID: PVE-002
- Severity: Informational
- Likelihood: Low
- Impact: None

- Target: src/spot.sol
- Category: Numeric Errors [21]
- CWE subcategory: CWE-369 [8]

Description

The poke() function in spot.sol does not validate par and ilks[ilk].mat before dividing something by them respectively. Since the existence of file() functions for setting par and ilks[ilk].mat, this could lead to divide by zero exceptions.

In particular, while auditing poke(), we notice that there are two rdiv operations in line 84. However, neither of them check the divide by zero cases, leaving the EVM reverts without useful information.

```
82
   function poke(bytes32 ilk) external {
83
        (bytes32 val, bool zzz) = ilks[ilk].pip.peek();
84
        if (zzz) {
85
            uint256 spot = rdiv(rdiv(mul(uint(val), 10 ** 9), par), ilks[ilk].mat);
86
            vat.file(ilk, "spot", spot);
            emit Poke(ilk , val , spot);
87
88
       }
89
   }
```



Recommendation Alter the poke() function in Spotter to explicitly check the validity of rdiv perands as follows:

operands as follows: function poke(bytes32 ilk) external { 82 83 (bytes32 val, bool zzz) = ilks[ilk].pip.peek(); 84 if (zzz) { 85 require(par > 0, "spot/invalid-par"); 86 require(ilks[ilk.mat] > 0, "spot/invalid-ilk"); 87 uint256 spot = rdiv(rdiv(mul(uint(val), 10 ** 9), par), ilks[ilk].mat); vat.file(ilk , "spot", spot); 88 emit Poke(ilk , val , spot); 89 90 }

```
90 }
91 }
```



3.3 Inconsistent Time Type in Debt Engine

- ID: PVE-003
- Severity: Informational
- Likelihood: Low
- Impact: None

Target: src/vow.solCategory: Coding Practices [18]

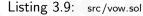
• CWE subcategory: CWE-474 [10]

Description

In vow.sol, the uint256 variable, wait, is used to represent a specific time. However, other similar variables such as ttl and tau are declared with the type uint48 in flop.sol.

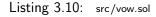
The wait variable declared in vow.sol is an uint256 (line 57).

```
48 // --- Data ---
49 VatLike public vat;
50 Flapper public flapper;
51 Flopper public flopper;
53 mapping (uint256 => uint256) public sin; // debt queue
54 uint256 public Sin; // queued debt
                                              [rad]
55 uint256 public Ash;
                       // on-auction debt
                                                [rad]
57 uint256 public wait; // flop delay
                                                 [rad]
58 uint256 public sump; // flop fixed lot size
                                                 [rad]
59 uint256 public bump; // flap fixed lot size
                                                 [rad]
60 uint256 public hump; // surplus buffer
                                                 [rad]
```



As shown in line 100, wait is used to represent a specific time.

```
99 function flog(uint era) external note {
100     require(add(era, wait) <= now);
101     Sin = sub(Sin, sin[era]);
102     sin[era] = 0;
103 }</pre>
```



However, in flop.sol, ttl and tau are declared as uint48 variables but also used to represent time such as line 115.

```
112 function tick(uint id) external note {
113    require(bids[id].end < now);
114    require(bids[id].tic == 0);
115    bids[id].end = add(uint48(now), tau);
116 }</pre>
```

Listing 3.11: src/flop.sol

Recommendation Reflect wait with proper type casting such as line 87 in the following:

```
86 function file(bytes32 what, uint data) external note auth {
87     if (what == "wait") wait = uint(uint48(data));
88     if (what == "bump") bump = data;
89     if (what == "sump") sump = data;
90     if (what == "hump") hump = data;
91 }
```



3.4 approve()/transferFrom() Race Condition

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Medium

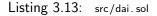
- Target: src/dai.sol
- Category: Time and State [17]
- CWE subcategory: CWE-362 [7]

Description

There is a known race condition issue regarding approve() / transferFrom() [3]. Specifically, when a user intends to reduce the allowed spending amount previously approved from, say, 10 DAI to 1 DAI. The previously approved spender might race to transfer the amount you initially approved (the 10 DAI) and then additionally spend the new amount you just approved (1 DAI). This breaks the user's intention of restricting the spender to the new amount, **not** the sum of old amount and new amount.

With the introduction of new approve()-style permit() and transferFrom()-style push()/pull() /move(), it apparently raises the concern of possible race conditions. To alleviate such concern, it is recommended to apply a known workaround (e.g., increaseApproval()/decreaseApproval()) and further add necessary sanity checks when entering the approve() function.

```
98 function approve(address usr, uint wad) external returns (bool) {
99 allowance[msg.sender][usr] = wad;
100 emit Approval(msg.sender, usr, wad);
101 return true;
102 }
```



Recommendation Add additional sanity checks in approve() and workaround functions increaseApproval ()/decreaseApproval().

```
98
99
```

```
function approve(address usr, uint wad) external returns (bool) {
    require(wad == 0 || allowance[msg.sender][usr] == 0);
```

100	allowance[msg.sender][usr] = wad;
101	emit Approval(msg.sender, usr, wad);
102	return true;
103	}
104	<pre>function increaseApproval(address usr, uint wad) external returns (bool) {</pre>
105	allowance[msg.sender][usr] = add(allowance[msg.sender][usr], (wad));
106	emit Approval(msg.sender, usr, allowance[msg.sender][usr]);
107	return true;
108	}
109	<pre>function decreaseApproval(address usr, uint wad) external returns (bool) {</pre>
110	allowance[msg.sender][usr] = sub(allowance[msg.sender][usr], (wad));
111	emit Approval(msg.sender, usr, allowance[msg.sender][usr]);
112	return true;
113	}

Listing 3.14: Revised src/dai.sol

3.5 Unhandled Auction Corner Cases

- ID: PVE-005
- Severity: Informational
- Likelihood: Medium
- Impact: None

- Target: src/{flip, flop, flap}.sol
- Category: Business Logics [20]
- CWE subcategory: CWE-754 [14]

Description

The MCD introduces three types of auctions: flip, flop, and flap. The flip auction, known as the collateral auction, is used to sell off collateral from risky CDP positions in exchange for borrowed DAIs. The flop auction, known as the deficit auction, is used to cover underwater debt by selling off a fixed amount (sump) of DAI deficit at a time, resulting in diluted MKR valuation. The flap auction, known as the surplus auction, is used to auction off a fixed amount (bump) of DAI surplus at a time, resulting in increased MKR valuation.

In the current implementation, these three types of auctions share certain timing-related corner cases that have not be covered yet. Using the flip auction as an example, the auction has two termination conditions: either when tau seconds (initially 2 days) have passed from the moment the auction was initiated, or when ttl seconds (initially 3 hours) have passed from the moment the last bid was placed. Once each termination condition is met, flip will not accept any more bids, effectively considering the last bidder winning and allowing it to claim the auctioned collateral. Both tau and ttl are risk parameters, initially set to be 2 days and 3 hours respectively, but reconfigurable through governance (e.g., via the file interface).

Note that flip does not cover two particular auction moments, either when bids[id].tic = now or bids[id].end = now. The first moment refers to the exact moment when the last bid expires and the second moment refers to the exact moment when the auction expires (e.g., reaching the allowed time period). flip can be explicitly designed to either accept or deny any bids arrived in these two particular moments. Current flip seems to implicitly deny such last-minute bids. With that, the deal() should be allowed to claim the auctioned collateral starting from the very moment inclusively, not exclusively (current case).

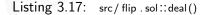
```
127 function tend(uint id, uint lot, uint bid) external note {
128 require(bids[id].guy != address(0));
129 require(bids[id].tic > now || bids[id].tic == 0);
130 require(bids[id].end > now);
131 ...
132 }
```

Listing 3.15: src/flip.sol::tend()

```
144 function dent(uint id, uint lot, uint bid) external note {
145 require(bids[id].guy != address(0));
146 require(bids[id].tic > now || bids[id].tic == 0);
147 require(bids[id].end > now);
148 ...
149 }
```

Listing 3.16: src/flip.sol::dent()

```
161 function deal(uint id) external note {
162 require(bids[id].tic != 0 && (bids[id].tic < now || bids[id].end < now));
163 ...
164 }</pre>
```



Recommendation Cover these corner cases by either allowing last-minute bids on these particular moments in flip.tend(), flip.dent(), flop.dent(), and flap.tend() or allowing the immediate deal settlement in flip.deal(), flop.deal(), and flap.deal(). The latter requires minimal coding changes and has small tangible implication, and thus is strongly preferred. The required changes are shown in the following. Note this applies to flip.deal(), flop.deal(), and flap.deal().

```
161 function deal(uint id) external note {
162 require(bids[id].tic != 0 && (bids[id].tic <= now || bids[id].end <= now));
163 ...
164 }</pre>
```

Listing 3.18: Revised src/flip.sol::deal()

3.6 CDP Fork Restrictiveness

- ID: PVE-006
- Severity: Informational
- Likelihood: High
- Impact: None

Description

- Target: src/vat.sol
- Category: Coding Practices [18]
- CWE subcategory: CWE-474 [10]

vat supports CDP fungibility (via fork()) by allowing the movement of related collateral and debt between two CDPs. The movement is granted on the condition that both src and dst consent the initiator, i.e., msg.sender. This condition can be relaxed such that either the affected CDP becomes more safe, or the owner consents. The relaxed condition shares the same spirit with the sanity check conditions (i.e., require(either(both(dart <= 0, dink >= 0), wish(u, msg.sender)))) applied in CDP-manipulating frob() in line 181.

```
function fork(bytes32 ilk, address src, address dst, int dink, int dart) external
197
            note {
198
             Urn storage u = urns[ilk][src];
199
             Urn storage v = urns[ilk][dst];
200
             llk storage i = ilks[ilk];
202
             u.ink = sub(u.ink, dink);
203
             u.art = sub(u.art, dart);
204
             v.ink = add(v.ink, dink);
205
             v.art = add(v.art, dart);
207
             uint utab = mul(u.art, i.rate);
208
             uint vtab = mul(v.art, i.rate);
210
             // both sides consent
211
             require(wish(src, msg.sender) && wish(dst, msg.sender));
213
             // both sides safe
214
             require(utab <= mul(u.ink, i.spot));</pre>
215
             require(vtab <= mul(v.ink, i.spot));</pre>
217
             // both sides non-dusty
218
             require(utab >= i.dust || u.art == 0);
             require(vtab >= i.dust || v.art == 0);
219
220
```

Listing 3.19: src/vat.sol

Recommendation Relax the restrictive src-and-dst-consenting condition to either the affected CDP becomes more safe or the owner consents (in fork()).

```
197
        function fork(bytes32 ilk, address src, address dst, int dink, int dart) external
            note {
198
             Urn storage u = urns[ilk][src];
199
             Urn storage v = urns[ilk][dst];
200
             llk storage i = ilks[ilk];
202
             u.ink = sub(u.ink, dink);
203
             u.art = sub(u.art, dart);
204
             v.ink = add(v.ink, dink);
205
             v.art = add(v.art, dart);
207
             uint utab = mul(u.art, i.rate);
208
             uint vtab = mul(v.art, i.rate);
210
             // src urn is either more safe, or src owner consents
211
             require (either (both (dart \leq 0, dink \geq 0), wish (src, msg.sender)));
212
             //\ {\rm dst} urn is either more safe, or dst owner consents
213
             require (either (both (dart >= 0, dink <= 0), wish (dst, msg.sender)));
215
             // both sides safe
             require(utab <= mul(u.ink, i.spot));</pre>
216
217
             require(vtab <= mul(v.ink, i.spot));</pre>
219
             // both sides non-dusty
220
             require(utab >= i.dust || u.art == 0);
221
             require(vtab >= i.dust || v.art == 0);
222
```

Listing 3.20: Revised src/vat.sol

3.7 Drip Efficiency Improvement

- ID: PVE-007
- Severity: Informational
- Likelihood: High
- Impact: None

Description

Both jug and pot contracts implement the drip() op to collect either stability fees or saving interests. Note that the operation records the last collection time in rho. And the operation will proceed when require(now >= ilks[ilk].rho) (in jug.drip()) or require(now >= rho) (in pot.drip()). The drip() can be further improved by changing the require() to if condition: if (ilks[ilk].rho >= now) return ; (in jug.drip()) or if (rho >= now) return; (in pot.drip()).

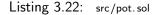
- Target: src/{jug.sol, pot.sol}
- Category: Coding Practices [18]
- CWE subcategory: CWE-1164 [5]

One benefit is to avoid unnecessary friction potentially caused by require() (that might revert ongoing transaction). The second benefit is the improved gas efficiency by avoiding unnecessary computations and inter-contract calls.

```
95 // --- Stability Fee Collection ---
96 function drip(bytes32 ilk) external note {
97 require(now >= ilks[ilk].rho);
98 VatLike.llk memory i = vat.ilks(ilk);
99 vat.fold(ilk, vow, diff(rmul(rpow(add(base, ilks[ilk].duty), now - ilks[ilk].rho
, ONE), i.rate), i.rate));
100 ilks[ilk].rho = now;
101 }
```

Listing 3.21: src/jug.sol

```
124 // --- Savings Rate Accumulation ---
125 function drip() external note {
126    require(now >= rho);
127    uint chi_ = sub(rmul(rpow(dsr, now - rho, ONE), chi), chi);
128    chi = add(chi, chi_);
129    rho = now;
130    vat.suck(address(vow), address(this), mul(Pie, chi_));
131 }
```



Recommendation Improve the drip() efficiency by removing the = in the required condition. Moreover, replace require accordingly with if to avoid introducing unnecessary frictions.

```
95 // --- Stability Fee Collection ---
96 function drip(bytes32 ilk) external note {
97 if (ilks[ilk].rho >= now) return;
98 VatLike.Ilk memory i = vat.ilks(ilk);
99 vat.fold(ilk, vow, diff(rmul(rpow(add(base, ilks[ilk].duty), now - ilks[ilk].rho
, ONE), i.rate), i.rate));
100 ilks[ilk].rho = now;
101 }
```

Listing 3.23: Revised src/jug.sol

```
124 // --- Savings Rate Accumulation ---
125 function drip() external note {
126     if (rho >= now) return;
127     uint chi_ = sub(rmul(rpow(dsr, now - rho, ONE), chi), chi);
128     chi = add(chi, chi_);
129     rho = now;
130     vat.suck(address(vow), address(this), mul(Pie, chi_));
131  }
```

Listing 3.24: Revised src/pot.sol

3.8 Debt Auction Prevention

- ID: PVE-008
- Severity: Informational
- Likelihood: High
- Impact: None

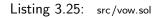
Description

- Target: src/vow.sol
- Category: Business Logics [20]
- CWE subcategory: CWE-754 [14]

The debt engine, vow, is vulnerable to a potential denial-of-service attack when vat.dai(address(vow)) > 0. This is a similar issue with PVE-001, but in a much more realistic context. Specifically, flop() is the function that kicks off a debt deficit auction. However, the kick-off requires vat.dai(address(this)) = 0 where this is vow itself. Similar to PVE-001, if someone attempts to deposit some tiny dai amount to make vat.dai(address(vow)) > 0, which effectively prevents the debt auction from proceeding.

Fortunately, such issue can be effectively alleviated by calling vow.heal() right before the flop() within the same transaction, hence the severity is similarly lowered to informational. However, it is strongly recommended to add this issue into related memos for proper auction flop() operations.

```
118 // Debt auction
119 function flop() external note returns (uint id) {
120 require(sump <= sub(sub(vat.sin(address(this)), Sin), Ash));
121 require(vat.dai(address(this)) == 0);
122 Ash = add(Ash, sump);
123 id = flopper.kick(address(this), uint(-1), sump);
124 }
```



Recommendation Alleviate the issue by combining with a vow.heal() call in the same transaction as the flop().

3.9 Misadjusted CDP Cancellation

- ID: PVE-009
- Severity: Medium
- Likelihood: Low
- Impact: High

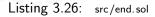
• Target: src/end.sol

- Category: Behavioral Issues [19]
- CWE subcategory: CWE-440 [9]

Description

The fourth step of 'End', skim(), is designed to cancel current CDPs. It basically grab()s away current CDP debt with the goal of converting all art to sin. Note the owed debt is calculated and saved in the local owe variable, i.e., owe = rmul(rmul(u.art, i.rate), tag[ilk]) (line 294 in end.sol). However, it misses the par factor, the reference price feed per dai, leading to debt misadjustment in CDP Cancellation.

```
289
        function skim(bytes32 ilk , address urn) external note {
290
             require(tag[ilk] != 0);
291
             VatLike.IIk memory i = vat.ilks(ilk);
292
             VatLike.Urn memory u = vat.urns(ilk, urn);
294
             uint owe = rmul(rmul(u.art, i.rate), tag[ilk]);
295
             uint wad = min(u.ink, owe);
296
             gap[ilk] = add(gap[ilk], sub(owe, wad));
298
             require (wad \leq 2**255 & u.art \leq 2**255);
299
             vat.grab(ilk, urn, address(this), address(vow), -int(wad), -int(u.art));
300
```



Recommendation Reconsider the associated , i.e., par, in the calculation of owed debt when canceling CDPs.

3.10 Auction Kick-Off Authorization

- ID: PVE-010
- Severity: High
- Likelihood: Medium
- Impact: High

Description

- Target: src/{flip, flap}.sol
- Category: Business Logics [20]
- CWE subcategory: CWE-862 [15]

As mentioned in Section 3.5, MCD has three types of auctions: flip, flop, and flap. The flip auction, known as the collateral auction, is used to sell off collateral from risky CDP positions in exchange for borrowed DAIs. The kick-off is initiated when a keeper calls cat.bite(). The flop auction, known as the deficit auction, is used to cover underwater debt by selling off a fixed amount (sump) of DAI deficit at a time, resulting in diluted MKR valuation. The kick-off is initiated when a keeper calls vow.flop(). The flap auction, known as the surplus auction, is used to auction off a fixed amount (bump) of DAI surplus at a time, resulting in increased MKR valuation. The kick-off should be initiated when a keeper calls vow.flap().

We emphasize that the kick-offs of these three auctions should only be initiated from cat.bite(), vow.flop(), and vow.flap() respectively. In other words, there is a need to add necessary authorization to ensure that related kick()s are only called from these trusted contracts. Note that the current code base only has the required authorization in place for flop.kick(), but not flip.kick() and flap .kick(). If there is no such authorization, it is possible to inject crafted auctions that will be later finalized in deal to cause asset loss or manipulation.

To elaborate, the crafted auctions via flap.kick() injection, once finalized in flap.deal(), could lead to gem.burn(address(this), bids[id].bid) -- line 135 in flap.sol where the burned amount bids[id].bid is directly controlled by the attacker. Note this attack might be readily launched without much cost (basically transaction gas fee) and could seriously affect MKR valuation and even the stability of entire DAI surplus auction subsystem.

Specifically, an attacker could simply kick off (via flap.kick()) a honeypot auction (e.g., initialized with an extremely high 1,000MRK bid for a very low 0.01DAI lot). This honeypot auction will not likely attract any external bidders (including bots) as it does not appear to be profitable. Until the auction expires, say, 2 days later, the attacker could simply call flap.deal() to finalize the auction. Due to the issue identified in PVE-013 (Section 3.12), the honeypot auction will be successfully closed, but the closed deal() will cause the auction subsystem to burn (via gem.burn()) the amount of MKR specified in the honeypot auction (as far as it is less than the amount held by the flap auction subsystem at the moment when deal() is being closed). The burnt MKR amount leads to a direct loss of flap auction subsystem and could cascading impact other ongoing surplus auctions. We consider this a

high-severity issue since the difficulty of launching this attack is very low, but the impact of causing MKR loss and disrupting normal surplus auctions is quite serious!

For the crafted auctions via flip.kick() injection, it may not immediately lead to materialized cost as the flip.deal() simply "returns" back the collateralized asset. But if such auction is injected right before 'End', such auction will be trusted by 'End' and the claimed bids[id].bid (controlled by injector) will be "credited", effectively stealing assets through 'End'.

During our investigation, we also notice that both flip and flop auctions have the tick mechanism that can be used to re-start (or extend) the auction if there is no bid received yet. If tick is designed to be callable by any entity (likely the first bidder), there is no need to add authorization. However, if it is intended for specific auction entity, there is a need to add similar authorization. It is our understanding that the former is the case, i.e., tick is designed to be callable by any entity.

```
103
         // --- Auction ---
         function kick (address usr, address gal, uint tab, uint lot, uint bid)
104
105
             public returns (uint id)
106
         {
107
             require (kicks < uint(-1));
108
             id = ++kicks;
110
             bids[id].bid = bid;
111
             bids[id].lot = lot;
112
             bids[id].guy = msg.sender; // configurable??
113
             bids[id].end = add(uint48(now), tau);
114
             bids[id].usr = usr;
115
             bids[id].gal = gal;
116
             bids[id].tab = tab;
118
             vat.flux(ilk, msg.sender, address(this), lot);
120
             emit Kick(id, lot, bid, tab, usr, gal);
121
         }
122
         function tick(uint id) external note {
123
             require(bids[id].end < now);</pre>
124
             require(bids[id].tic == 0);
125
             bids[id].end = add(uint48(now), tau);
126
```

Listing 3.27: src/flip.sol

```
99
         // --- Auction ---
100
         function kick(address gal, uint lot, uint bid) external auth returns (uint id) {
101
             require(live == 1);
102
             require (kicks < uint(-1));
103
             id = ++kicks;
105
             bids[id].bid = bid;
106
             bids[id].lot = lot;
107
             bids[id].guy = gal;
108
             bids[id].end = add(uint48(now), tau);
```

```
110 emit Kick(id, lot, bid, gal);
111 }
112 function tick(uint id) external note {
113 require(bids[id].end < now);
114 require(bids[id].tic == 0);
115 bids[id].end = add(uint48(now), tau);
116 }</pre>
```

Listing 3.28: src/flop.sol

```
98
         // --- Auction ---
99
         function kick(uint lot, uint bid) external returns (uint id) {
100
             require(live == 1);
101
             require (kicks < uint(-1));
102
             id = ++kicks;
104
             bids[id].bid = bid;
105
             bids[id].lot = lot;
106
             bids[id].guy = msg.sender; // configurable??
107
             bids[id].end = add(uint48(now), tau);
109
             vat.move(msg.sender, address(this), lot);
111
             emit Kick(id, lot, bid);
112
         ł
```

Listing 3.29: src/flap.sol

Recommendation Add necessary authorization to ensure auctions are kick()'ed-off from trusted contracts, i.e., cat and vow respectively.

```
103
         // --- Auction ---
104
         function kick (address usr, address gal, uint tab, uint lot, uint bid)
105
             public auth returns (uint id)
106
         {
107
             require (kicks < uint(-1));
108
             id = ++kicks;
110
             bids[id].bid = bid;
             bids[id].lot = lot;
111
             bids[id].guy = msg.sender; // configurable??
112
113
             bids[id].end = add(uint48(now), tau);
114
             bids[id].usr = usr;
115
             bids[id].gal = gal;
116
             bids[id].tab = tab;
118
             vat.flux(ilk, msg.sender, address(this), lot);
120
             emit Kick(id, lot, bid, tab, usr, gal);
121
```

Listing 3.30: Revised src/flip.sol

```
98
         // --- Auction ---
99
         function kick(uint lot, uint bid) external auth returns (uint id) {
100
             require(live == 1);
101
             require (kicks < uint (-1));
102
             id = ++kicks;
104
             bids[id].bid = bid;
             bids[id].lot = lot;
105
             bids[id].guy = msg.sender; // configurable??
106
107
             bids [id].end = add(uint48(now), tau);
109
             vat.move(msg.sender, address(this), lot);
             emit Kick(id, lot, bid);
111
112
         }
```

Listing 3.31: Revised src/flap.sol

3.11 Auction Tick Validity

- ID: PVE-011
- Severity: Informational
- Likelihood: High
- Impact: None

- Target: src/{flip, flop}.sol
- Category: Time and State [17]
- CWE subcategory: CVE-668 [13]

Description

As mentioned in PVE-011 (Section 3.10), both flip and flop auctions have the tick mechanism that can be used to re-start (or extend) the auction if there is no bid received yet. And tick is designed to be callable by any entity and thus does not require auth protection.

However, current tick mechanism does not validate whether the given auction ID exists yet and, once called, will simply set the given auction's total auction time. Note the given auction must not receive any bids yet, and the current implementation allows those upcoming, but not present yet, auctions as well.

```
122 function tick(uint id) external note {
123 require(bids[id].end < now);
124 require(bids[id].tic == 0);
125 bids[id].end = add(uint48(now), tau);
126 }</pre>
```

Listing 3.32: src/flip.sol

Specifically, if we take a look at the tick implementation in flip.sol, current sanity checks verify the require(bids[id].end < now) and require(bids[id].tic == 0). Both succeed as for a non-present

auction, the end field and the tic field hold the default 0. Therefore, the non-present auction's end time is now pre-set. Fortunately, when the upcoming auction is kicked-off, kick will reset the auction end time, neutralizing previous effect.

Our assessment indicates this might not be an intended behavior, and though no harmful impact or exploitation have been identified yet, it is still recommended to add necessary validity check inside tick() function.

Recommendation Validate the auction ID when entering tick(). This applies to flip.tick() and flop.tick()

```
122 function tick(uint id) external note {
123     require(bids[id].guy != address(0));
124     require(bids[id].end < now);
125     require(bids[id].tic == 0);
126     bids[id].end = add(uint48(now), tau);
127 }</pre>
```

Listing 3.33: Revised src/flip.sol

```
112 function tick(uint id) external note {
113 require(bids[id].guy != address(0));
114 require(bids[id].end < now);
115 require(bids[id].tic == 0);
116 bids[id].end = add(uint48(now), tau);
117 }</pre>
```

Listing 3.34: Revised src/flop.sol

3.12 Auction Deal Inconsistency

- ID: PVE-012
- Severity: Low
- Likelihood: Medium

- Target: src/{flop, flap}.sol
- Category: Business Logics [20]
- CWE subcategory: CWE-754 [14]

Impact: Low

Description

To finalize an auction, the deal mechanism is implemented in flip, flap, and flop. However, in flip.deal(), the sanity checks are subtly inconsistent with those in flap.deal() and flop.deal().

```
161 function deal(uint id) external note {
162 require(bids[id].tic != 0 && (bids[id].tic < now || bids[id].end < now));
163 vat.flux(ilk, address(this), bids[id].guy, bids[id].lot);
164 delete bids[id];</pre>
```

Listing 3.35: src/flip.sol

In particular, in flip.deal():162, bids[id].tic != 0 is a necessary condition to enter flip.deal().
Then, either bids[id].tic < now or bids[id].end < now can pass the sanity checks.</pre>

```
130 function deal(uint id) external note {
131 require(live == 1);
132 require(bids[id].tic < now && bids[id].tic != 0 ||
133 bids[id].end < now);
134 vat.move(address(this), bids[id].guy, bids[id].lot);
135 gem.burn(address(this), bids[id].bid);
136 delete bids[id];
137 }</pre>
```

Listing 3.36: src/flap.sol

Listing 3.37: src/flop.sol

However, in flap.deal():132-133 and flop.deal():135-136, the sanity checks allows any bids[id] with bids[id].end < now to pass the checks, which is inconsistent with the implementation of flip. deal(). As discussed in Section 3.10, such inconsistency can be exploited together with PVE-011 to cause damages to the surplus auction subsystem.

Furthermore, an attacker can use a non-present id to trigger flap.deal() and flop.deal() due to the fact that bids[id].end < now is always true for non-present id (i.e., bids[id].end is 0). This also has the noisy side-effect of generating non-meaningful events (e.g., Transfer events generated by gem.burn() with zero amount).

Recommendation Make the sanity checks of flap.deal() and flop.deal() consistent with flip.deal() as follows:

```
130 function deal(uint id) external note {
131 require(live = 1);
132 require(bids[id].tic != 0 && (bids[id].tic < now || bids[id].end < now));
133 vat.move(address(this), bids[id].guy, bids[id].lot);
134 gem.burn(address(this), bids[id].bid);
135 delete bids[id];
136 }</pre>
```

Listing 3.38: Revised src/flap.sol

```
133 function deal(uint id) external note {
134 require(live = 1);
135 require(bids[id].tic != 0 && (bids[id].tic < now || bids[id].end < now));
136 gem.mint(bids[id].guy, bids[id].lot);
137 delete bids[id];
138 }</pre>
```



3.13 Bloated Setter Interface

- ID: PVE-013
- Severity: Informational
- Likelihood: Medium
- Impact: None

Description

- Target: dss-deploy/src/govActions.sol
- Category: Coding Practices [18]
- CWE subcategory: CWE-561 [11]

The Setter interface defines a common way to set up various risk parameters agreed upon through governance. Currently, there are 7 variants of file ops defined in Setter. However, among the 7 variants, the following 3 are not used in MCD: function file(address) public, function file(uint) public, and function file(bytes32, bytes32) public. These unused interfaces unnecessarily complicate the abstraction and understanding, and therefore are suggested for removal.

```
3 contract Setter {
4
       function file(address) public;
5
       function file (uint) public;
6
       function file(bytes32, address) public;
7
       function file(bytes32, uint) public;
8
       function file(bytes32, bytes32) public;
9
       function file(bytes32, bytes32, uint) public;
       function file(bytes32, bytes32, address) public;
10
11
       function rely(address) public;
12
       function deny(address) public;
13
       function init(bytes32) public;
14 }
```

Listing 3.40: dss-deploy/src/govActions.sol

Accordingly, on top of the Setter.file() interface, GovActions defines 7 related file high-level abstraction ops. And 3 of them are similarly unused in this MCD.

```
26 contract GovActions {
27 function file(address who, address data) public {
28 Setter(who).file(data);
29 }
```

```
31
        function file(address who, uint data) public {
32
            Setter(who).file(data);
33
        }
35
        function file (address who, bytes32 what, address data) public {
36
            Setter(who).file(what, data);
37
       }
39
        function file(address who, bytes32 what, uint data) public {
40
            Setter(who).file(what, data);
41
        }
43
        function file(address who, bytes32 what, bytes32 data) public {
44
            Setter(who).file(what, data);
45
        }
47
        function file (address who, bytes32 ilk, bytes32 what, uint data) public {
48
            Setter(who).file(ilk, what, data);
49
       }
51
        function file(address who, bytes32 ilk, bytes32 what, address data) public {
52
            Setter(who).file(ilk, what, data);
53
       }
54
55
   }
```

Listing 3.41: dss-deploy/src/govActions.sol

Recommendation Assuming Setter and govActions are designed only for MCD, we suggest the previously-mentioned unused 3 file interfaces in both Setter and govActions can be removed.

3.14 Missed Deployment Dependency Checks

- ID: PVE-014
- Severity: Informational
- Likelihood: Low
- Impact: None

Description

- Target: dss-deploy/src/DssDeploy.sol
- Category: Coding Practices [18]
- CWE subcategory: CWE-1120 [4]
- During the deployment of various MCD modules, there are multiple required dependency checks to ensure that those modules used in this deployment are ready for use. However, we identified a few occasions where some of these dependency checks are missing:

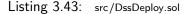
252 **function** deployLiquidator() **public** auth {

```
253
             require(address(vow) != address(0), "Missing previous step");
255
             // Deploy
256
             cat = catFab.newCat(address(vat));
258
             // Internal references set up
259
             cat.file("vow", address(vow));
261
             // Internal auth
262
             vat.rely(address(cat));
263
             vow.rely(address(cat));
264
```

Listing 3.42: src/DssDeploy.sol

In deployLiquidator(), vat is used in line 256 to deploy cat but the dependency check for address (vat) is missing.

```
266
        function deployShutdown(address gov, address pit, uint256 min) public auth {
267
             require(address(cat) != address(0), "Missing previous step");
269
             // Deploy
270
             end = endFab.newEnd();
272
             // Internal references set up
             end.file("vat", address(vat));
273
274
             end.file("cat", address(cat));
275
             end.file("vow", address(vow));
276
             end.file("pot", address(pot));
277
             end.file("spot", address(spotter));
279
             // Internal auth
280
             vat.rely(address(end));
             cat.rely(address(end));
281
282
             vow.rely(address(end));
283
             pot.rely(address(end));
285
             // Deploy ESM
286
             esm = new ESM(gov, address(end), address(pit), min);
287
             end.rely(address(esm));
288
```



In deployShutdown(), vat, vow, pot, and spotter, are used in line 273-277, but the dependency checks for them are missing as well.

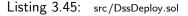
```
290 function deployPause(uint delay, DSAuthority authority) public auth {
291 require(address(dai) != address(0), "Missing previous step");
292 require(address(end) != address(0), "Missing previous step");
294 pause = pauseFab.newPause(delay, address(0), authority);
```

296	vat.rely(address (pause.proxy()));
297	cat . rely (address (pause . proxy ())) ;
298	vow.rely(address (pause.proxy()));
299	jug . rely (address (pause . proxy ())) ;
300	pot . rely (address (pause . proxy ())) ;
301	<pre>spotter . rely (address (pause . proxy ()));</pre>
302	flap . rely (address (pause . proxy ())) ;
303	flop . rely (address (pause . proxy ())) ;
304	end . rely (address (pause . proxy ())) ;
306	<pre>this.setAuthority(authority);</pre>
307	<pre>this.setOwner(address(0));</pre>
308	}

Listing 3.44: src/DssDeploy.sol

In deployPause(), vat, cat, vow, jug, pot, spotter, flap, flop are used in line 296-303, but not checked for their presences at the beginning of the function.

```
310
         function deployCollateral(bytes32 ilk, address adapter, address pip) public auth {
311
             require(ilk != bytes32(""), "Missing ilk name");
312
             require(adapter != address(0), "Missing adapter address");
             require(pip != address(0), "Missing PIP address");
313
314
             require(address(pause) != address(0), "Missing previous step");
316
             // Deploy
317
             ilks[ilk].flip = flipFab.newFlip(address(vat), ilk);
318
             ilks[ilk].adapter = adapter;
319
             Spotter(spotter).file(ilk, address(pip)); // Set pip
321
             // Internal references set up
322
             cat.file(ilk, "flip", address(ilks[ilk].flip));
323
             vat.init(ilk);
324
            jug.init(ilk);
326
             // Internal auth
327
             vat.rely(adapter);
328
             ilks[ilk].flip.rely(address(end));
329
             ilks[ilk].flip.rely(address(pause.proxy()));
330
```



In deployCollateral(), cat, vat, jug are used in line 322-324, but not checked before the usage.

Recommendation Add necessary dependency checks as follows:

```
252 function deployLiquidator() public auth {
253 require(address(vat) != address(0), "Missing previous step");
254 require(address(vow) != address(0), "Missing previous step");
256 // Deploy
257 cat = catFab.newCat(address(vat));
```

```
259 // Internal references set up
260 cat.file("vow", address(vow));
262 // Internal auth
263 vat.rely(address(cat));
264 vow.rely(address(cat));
```

```
265
```

Listing 3.46: Revised deployLiquidator ()

```
266
         function deployShutdown(address gov, address pit, uint256 min) public auth {
267
             require(address(vat) != address(0), "Missing previous step");
268
             require(address(cat) != address(0), "Missing previous step");
269
             require(address(vow) != address(0), "Missing previous step");
270
             require(address(pot) != address(0), "Missing previous step");
271
             require(address(spotter) != address(0), "Missing previous step");
273
             // Deploy
274
             end = endFab.newEnd();
276
             // Internal references set up
277
             end.file("vat", address(vat));
278
             end.file("cat", address(cat));
279
             end.file("vow", address(vow));
             end.file("pot", address(pot));
280
281
             end.file("spot", address(spotter));
283
             // Internal auth
284
             vat.rely(address(end));
285
             cat.rely(address(end));
286
             vow.rely(address(end));
287
             pot.rely(address(end));
289
             // Deploy ESM
290
             esm = new ESM(gov, address(end), address(pit), min);
291
             end.rely(address(esm));
292
```

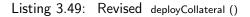
Listing 3.47: Revised deployShutdown()

```
290
        function deployPause(uint delay, DSAuthority authority) public auth {
291
             require(address(dai) != address(0), "Missing previous step");
292
             require(address(end) != address(0), "Missing previous step");
293
            require(address(vat) != address(0), "Missing previous step");
294
            require(address(cat) != address(0), "Missing previous step");
295
            require(address(vow) != address(0), "Missing previous step");
296
            require(address(pot) != address(0), "Missing previous step");
297
            require(address(spotter) != address(0), "Missing previous step");
298
             require(address(flap) != address(0), "Missing previous step");
299
             require(address(flop) != address(0), "Missing previous step");
```

```
301
              pause = pauseFab.newPause(delay, address(0), authority);
303
             vat.rely(address(pause.proxy()));
304
             cat.rely(address(pause.proxy()));
305
             vow.rely(address(pause.proxy()));
306
             jug.rely(address(pause.proxy()));
307
             pot.rely(address(pause.proxy()));
308
             spotter.rely(address(pause.proxy()));
309
              flap . rely ( address ( pause . proxy ( ) ) );
310
             flop . rely ( address ( pause . proxy ( ) ) );
             end.rely(address(pause.proxy()));
311
313
             this.setAuthority(authority);
314
              this.setOwner(address(0));
315
```

Listing 3.48: Revised deployPause()

```
310
         function deployCollateral(bytes32 ilk, address adapter, address pip) public auth {
311
             require(ilk != bytes32(""), "Missing ilk name");
312
             require(adapter != address(0), "Missing adapter address");
313
             require(pip != address(0), "Missing PIP address");
314
             require(address(pause) != address(0), "Missing previous step");
315
             require(address(cat) != address(0), "Missing previous step");
316
             require(address(vat) != address(0), "Missing previous step");
317
             require(address(jug) != address(0), "Missing previous step");
319
             // Deploy
320
             ilks[ilk].flip = flipFab.newFlip(address(vat), ilk);
321
             ilks[ilk].adapter = adapter;
322
             Spotter(spotter).file(ilk, address(pip)); // Set pip
324
             // Internal references set up
325
             cat.file(ilk, "flip", address(ilks[ilk].flip));
326
             vat.init(ilk);
327
             jug.init(ilk);
329
             // Internal auth
330
             vat.rely(adapter);
331
             ilks[ilk].flip.rely(address(end));
332
             ilks[ilk].flip.rely(address(pause.proxy()));
333
```



After the discussion with Maker Foundation, this issue has no impact because of the existence of variable dependencies which guarantee the order of deployment functions. For example, the require (address(vow) != address(0)) check in deployLiquidator() ensures deployTaxationAndAuctions() was successfully executed with vow set before entering deployLiquidator(). Since deployTaxationAndAuctions () checks require(address(vat) != address(0)), deployLiquidator() does not need to check vat. However, this implementation makes it difficult to understand and/or maintain the software.

3.15 Excessive Authorization in Deployment

- ID: PVE-015
- Severity: Low
- Likelihood: Low
- Impact: Low

- Target: dss-deploy/src/DssDeploy.sol
- Category: Security Features [16]
- CWE subcategory: CWE-287 [6]

Description

During the deployment of various MCD modules, the rely() mechanism is used to configure the trusted entities of a given contract. However, we identified some improper authorization settings:

```
229
         function deployTaxationAndAuctions(address gov) public auth {
230
             require(gov != address(0), "Missing GOV address");
231
             require(address(vat) != address(0), "Missing previous step");
233
             // Deploy
234
             jug = jugFab.newJug(address(vat));
235
             pot = potFab.newPot(address(vat));
236
             flap = flapFab.newFlap(address(vat), gov);
237
             flop = flopFab.newFlop(address(vat), gov);
238
             vow = vowFab.newVow(address(vat), address(flap), address(flop));
240
             // Internal references set up
241
             jug.file("vow", address(vow));
242
             pot.file("vow", address(vow));
244
             // Internal auth
245
             vat.rely(address(vow));
246
             vat.rely(address(jug));
247
             vat.rely(address(pot));
248
             flap.rely(address(vow));
249
             flop . rely ( address (vow ) );
250
```

Listing 3.50: dss-deploy/src/DssDeploy.sol::deployTaxationAndAuctions()

As shown in the above code snippet, vat is set to trust vow to call its functions in line 245. However, if we look into the vow implementation, the only function of vat called by vow is vat. heal() which needs no authorization at all. In other words, there is no auth modifier in vat.heal() declaration.

```
      238
      function heal(uint rad) external note {

      239
      address u = msg.sender;

      240
      sin[u] = sub(sin[u], rad);

      241
      dai[u] = sub(dai[u], rad);

      242
      vice = sub(vice, rad);

      243
      debt = sub(debt, rad);
```

244

Listing 3.51: dss/src/vat.sol::heal()

We point out that the documented contract-level permission graph [2] contains the rely arrow from vat to vow, which could be removed for consistency.

```
266
         function deployShutdown(address gov, address pit, uint256 min) public auth {
267
             require(address(cat) != address(0), "Missing previous step");
269
             // Deploy
270
             end = endFab.newEnd();
272
             // Internal references set up
273
             end.file("vat", address(vat));
             end.file("cat", address(cat));
274
275
             end.file("vow", address(vow));
276
             end.file("pot", address(pot));
277
             end.file("spot", address(spotter));
279
             // Internal auth
280
             vat.rely(address(end));
281
             cat.rely(address(end));
282
             vow.rely(address(end));
283
             pot.rely(address(end));
285
             // Deploy ESM
             esm = new ESM(gov, address(end), address(pit), min);
286
287
             end.rely(address(esm));
288
```



In addition, there exists another authorization setting that does not comply with the same permission graph. Specifically, in line 282, vow trusts end because of the need of calling vow.cage() when end kicks off its first step of **Global Settlement**. However, the permission graph misses one rely arrow from vow to end. This leads to inconsistency and brings unnecessary confusions for understanding.

Recommendation Remove the unnecessary rely() from deployTaxationAndAuctions() and accordingly fix the contract-level permission graph in [2] (with the addition of rely arrow from vow to end and the removal of rely arrow from vat to vow).

```
229
         function deployTaxationAndAuctions(address gov) public auth {
230
             require(gov != address(0), "Missing GOV address");
231
             require(address(vat) != address(0), "Missing previous step");
233
             // Deploy
234
             jug = jugFab.newJug(address(vat));
235
             pot = potFab.newPot(address(vat));
236
             flap = flapFab.newFlap(address(vat), gov);
             flop = flopFab.newFlop(address(vat), gov);
237
```

```
238
             vow = vowFab.newVow(address(vat), address(flap), address(flop));
240
             // Internal references set up
241
             jug.file("vow", address(vow));
242
             pot.file("vow", address(vow));
244
             // Internal auth
245
             vat.rely(address(jug));
246
             vat.rely(address(pot));
247
             flap.rely(address(vow));
248
             flop . rely ( address (vow ) );
249
```

Listing 3.53: Revised dss-deploy/src/DssDeploy.sol::deployTaxationAndAuctions()

3.16 Collateral Check in MCD CDP Manager

- ID: PVE-016
- Severity: Informational
- Likelihood: High
- Impact: None

- Target: dss-cdp-manager/src/DssCdpManager .sol
- Category: Coding Practices [18]
- CWE subcategory: CWE-628 [12]

Description

The DssCdpManager smart contract provides an interface to manage MCD-based CDPs in a way similar to manage SCD-based CDPs. However, when opening a CDP, it does not validate the provided collateral type, i.e., ilk. In other words, a user can open a CDP by providing an arbitrary ilk. As a result, current DssCdpManager implementation allows the opening of a basically non-functional CDP and defers the warning back to the owner until a later stage when the opened CDP is being operated (e.g., via frob). It is better to add sanity check up-front to ensure the provided collateral type is currently being supported in MCD.

```
108
         // Open a new cdp for a given usr address.
109
         function open(
110
             bytes32 ilk,
111
             address usr
112
         ) public note returns (uint) {
113
             require(usr != address(0), "usr-address-0");
115
             cdpi = add(cdpi, 1);
116
             urns[cdpi] = address(new UrnHandler(vat));
117
             owns[cdpi] = usr;
```

```
118
             ilks[cdpi] = ilk;
120
             // Add new CDP to double linked list and pointers
121
             if (first[usr] == 0) {
122
                  first [usr] = cdpi;
123
             }
             if (last[usr] != 0) {
124
                  list [cdpi].prev = last [usr];
125
126
                  list [last [usr]]. next = cdpi;
127
             }
128
             last[usr] = cdpi;
129
             count[usr] = add(count[usr], 1);
131
             emit NewCdp(msg.sender, usr, cdpi);
132
             return cdpi;
133
```

Listing 3.54: DssCdpManager.sol::open()

As shown in the above code snippet, the CDP is opened without checking the validity of ilk.

Recommendation Validate the provided collateral type ilk when a new MCD CDP is being opened. The validity can simply check whether the given ilk has been initialized yet in the MCD CDP engine vat.

```
contract VatLike {
6
7
        function ilks(bytes32) public view returns (uint, uint, uint, uint);
8
        function urns(bytes32, address) public view returns (uint, uint);
9
        function hope(address) public;
10
        function flux(bytes32, address, address, uint) public;
        function move(address, address, uint) public;
11
        function frob(bytes32, address, address, address, int, int) public;
12
13
        function fork(bytes32, address, address, int, int) public;
14 }
17
        // Open a new cdp for a given usr address.
18
        function open(
19
           bytes32 ilk,
20
            address usr
21
        ) public note returns (uint) {
22
            require(usr != address(0), "usr-address-0");
24
            (, uint rate,,,) = vat.ilks(ilk);
25
            require(rate != 0);
27
            cdpi = add(cdpi, 1);
28
            urns[cdpi] = address(new UrnHandler(vat));
29
            owns[cdpi] = usr;
30
            ilks[cdpi] = ilk;
```

```
32
            // Add new CDP to double linked list and pointers
33
            if (first[usr] == 0) {
34
                 first [usr] = cdpi;
35
            }
            if (last[usr] != 0) {
36
37
                 list [cdpi]. prev = last [usr];
38
                 list [last [usr]]. next = cdpi;
39
            }
40
            last [usr] = cdpi;
41
            count[usr] = add(count[usr], 1);
43
            emit NewCdp(msg.sender, usr, cdpi);
44
            return cdpi;
45
```

Listing 3.55: Revised DssCdpManager.sol

3.17 Other Suggestions

Due to the fact that compiler upgrades might bring unexpected compatibility or inter-version consistencies, it is always suggested to use fixed compiler versions whenever possible. As an example, we highly encourage to explicitly indicate the Solidity compiler version, e.g., pragma solidity 0.5.0; instead of pragma solidity ~0.5.0;.

Moreover, we strongly suggest not to use experimental Solidity features or third-party unaudited libraries. If necessary, refactor current code base to only use stable features or trusted libraries. In case there is an absolute need of leveraging experimental features or integrating external libraries, make necessary contingency plans.

4 Conclusion

In this audit, we have analyzed the Multi-Collateral Dai (MCD) implementation. During our auditing process, we are constantly impressed by the thinkings behind the Multi-Collateral Dai (MCD). It is indeed a rather complex system with various functionalities, but the entire system is cleanly designed and engineered. The related smart contracts are also neatly organized and elegantly implemented. Those identified issues are promptly confirmed and fixed.

Meanwhile, we emphasize that smart contracts are still in an early, but exciting stage of development. As disclaimed in Section 1.4, we greatly welcome any constructive feedbacks or suggestions regarding this report, including our methodology, audit findings, or potential gaps in scope/coverage.



5 Appendix

Basic Coding Bugs 5.1

5.1.1**Constructor Mismatch**

- Description: Whether the contract name and its constructor are not identical to each other.
- Result: Not found
- Severity: Critical

5.1.2 Ownership Takeover

- Description: Whether the set owner function is not protected. Shield
- Result: Not found
- Severity: Critical

5.1.3 **Redundant Fallback Function**

- Description: Whether the contract has a redundant fallback function.
- Result: Not found
- Severity: Critical

5.1.4 Overflows & Underflows

- Description: Whether the contract has general overflow or underflow vulnerabilities [24, 25, 26, 27, 29].
- Result: Not found
- Severity: Critical

5.1.5 Reentrancy

- <u>Description</u>: Reentrancy [30] is an issue when code can call back into your contract and change state, such as withdrawing ETHs.
- Result: Not found
- <u>Severity</u>: Critical

5.1.6 Money-Giving Bug

- Description: Whether the contract returns funds to an arbitrary address.
- <u>Result</u>: Not found
- Severity: High

5.1.7 Blackhole

- Description: Whether the contract locks ETH indefinitely: merely in without out.
- <u>Result</u>: Not found
- Severity: High

5.1.8 Unauthorized Self-Destruct

- Description: Whether the contract can be killed by any arbitrary address.
- Result: Not found
- Severity: Medium

5.1.9 Revert DoS

- Description: Whether the contract is vulnerable to DoS attack because of unexpected revert.
- <u>Result</u>: Not found
- Severity: Medium

5.1.10 Unchecked External Call

- Description: Whether the contract has any external call without checking the return value.
- Result: Not found
- Severity: Medium

5.1.11 Gasless Send

- Description: Whether the contract is vulnerable to gasless send.
- Result: Not found
- Severity: Medium

5.1.12 Send Instead Of Transfer

- Description: Whether the contract uses send instead of transfer.
- <u>Result</u>: Not found
- Severity: Medium

5.1.13 Costly Loop

- <u>Description</u>: Whether the contract has any costly loop which may lead to Out-Of-Gas exception.
- Result: Not found
- Severity: Medium

5.1.14 (Unsafe) Use Of Untrusted Libraries

- Description: Whether the contract use any suspicious libraries.
- <u>Result</u>: Not found
- Severity: Medium

5.1.15 (Unsafe) Use Of Predictable Variables

- <u>Description</u>: Whether the contract contains any randomness variable, but its value can be predicated.
- Result: Not found
- Severity: Medium

5.1.16 Transaction Ordering Dependence

- Description: Whether the final state of the contract depends on the order of the transactions.
- <u>Result</u>: Not found
- Severity: Medium

5.1.17 Deprecated Uses

- Description: Whether the contract use the deprecated tx.origin to perform the authorization.
- <u>Result</u>: Not found
- <u>Severity</u>: Medium

5.2 Semantic Consistency Checks

- <u>Description</u>: Whether the semantic of the white paper is different from the implementation of the contract.
- Result: Not found
- Severity: Critical

5.3 Additional Recommendations

5.3.1 Avoid Use of Variadic Byte Array

- <u>Description</u>: Use fixed-size byte array is better than that of byte[], as the latter is a waste of space.
- <u>Result</u>: Not found
- Severity: Low

5.3.2 Use Fixed Compiler Version

- Description: Use fixed compiler version is better.
- Result: Not found
- Severity: Low

5.3.3 Make Visibility Level Explicit

- Description: Assign explicit visibility specifiers for functions and state variables.
- Result: Not found
- Severity: Low

5.3.4 Make Type Inference Explicit

- <u>Description</u>: Do not use keyword var to specify the type, i.e., it asks the compiler to deduce the type, which is not safe especially in a loop.
- <u>Result</u>: Not found
- Severity: Low

5.3.5 Adhere To Function Declaration Strictly

- <u>Description</u>: Solidity compiler (version 0.4.23) enforces strict ABI length checks for return data from calls() [1], which may break the the execution if the function implementation does NOT follow its declaration (e.g., no return in implementing transfer() of ERC20 tokens).
- Result: Not found
- Severity: Low

References

- [1] axic. Enforcing ABI length checks for return data from calls can be breaking. https://github. com/ethereum/solidity/issues/4116.
- [2] Maker Foundation. Kovan deploy. https://github.com/makerdao/dss/wiki/Auth# kovan-deploy.
- [3] HaleTom. Resolution on the EIP20 API Approve / TransferFrom multiple withdrawal attack. https://github.com/ethereum/EIPs/issues/738.
- [4] MITRE. CWE-1120: Excessive Code Complexity. https://cwe.mitre.org/data/definitions/1120. html.
- [5] MITRE. CWE-1164: Irrelevant Code. https://cwe.mitre.org/data/definitions/1164.html.
- [6] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.
- [7] MITRE. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.
- [8] MITRE. CWE-369: Divide By Zero. https://cwe.mitre.org/data/definitions/369.html.
- [9] MITRE. CWE-440: Expected Behavior Violation. https://cwe.mitre.org/data/definitions/440. html.
- [10] MITRE. CWE-474: Use of Function with Inconsistent Implementations. https://cwe.mitre.org/ data/definitions/474.html.

- [11] MITRE. CWE-561: Dead Code. https://cwe.mitre.org/data/definitions/561.html.
- [12] MITRE. CWE-628: Function Call with Incorrectly Specified Arguments. https://cwe.mitre.org/ data/definitions/628.html.
- [13] MITRE. CWE-668: Exposure of Resource to Wrong Sphere. https://cwe.mitre.org/data/ definitions/668.html.
- [14] MITRE. CWE-754: Improper Check for Unusual or Exceptional Conditions. https://cwe.mitre. org/data/definitions/754.html.
- [15] MITRE. CWE-862: Missing Authorization. https://cwe.mitre.org/data/definitions/862.html.
- [16] MITRE. CWE CATEGORY: 7PK Security Features. https://cwe.mitre.org/data/definitions/ 254.html.
- [17] MITRE. CWE CATEGORY: 7PK Time and State. https://cwe.mitre.org/data/definitions/ 361.html.
- [18] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/ 1006.html.
- [19] MITRE. CWE CATEGORY: Behavioral Problems. https://cwe.mitre.org/data/definitions/438. html.
- [20] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/ 840.html.
- [21] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.
- [22] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699. html.
- [23] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_ Rating_Methodology.

- [24] PeckShield. ALERT: New batchOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10299). https://www.peckshield.com/2018/04/22/batchOverflow/.
- [25] PeckShield. New burnOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-11239). https://www.peckshield.com/2018/05/18/burnOverflow/.
- [26] PeckShield. New multiOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-10706). https://www.peckshield.com/2018/05/10/multiOverflow/.
- [27] PeckShield. New proxyOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10376). https://www.peckshield.com/2018/04/25/proxyOverflow/.
- [28] PeckShield. PeckShield Inc. https://www.peckshield.com.
- [29] PeckShield. Your Tokens Are Mine: A Suspicious Scam Token in A Top Exchange. https: //www.peckshield.com/2018/04/28/transferFlaw/.
- [30] Solidity. Warnings of Expressions and Control Structures. http://solidity.readthedocs.io/en/ develop/control-structures.html.