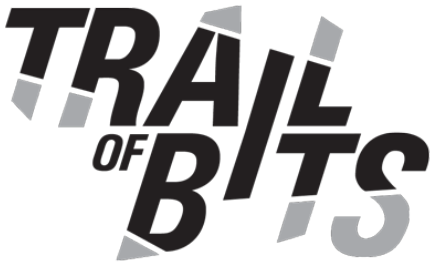

MakerDAO Liquidations 2.0

Security Assessment

March 19, 2021

Prepared For:
Chris Mooney | *Maker Foundation*
chrismooney@makerdao.com

Wouter Kampmann | *Maker Foundation*
wouter@makerdao.com

Prepared By:
Gustavo Grieco | *Trail of Bits*
gustavo.grieco@trailofbits.com

Maximilian Krüger | *Trail of Bits*
max.kruger@trailofbits.com

Executive Summary

Project Dashboard

Code Maturity Evaluation

Engagement Goals

Coverage

Automated Testing and Verification
- Automated Testing with Echidna

Recommendations Summary
- Short term
- Long term

Findings Summary
- 1. Very low values of milk.chop can block liquidations
- 2. An urn at the 0x0 address cannot be liquidated
- 3. Integer division rounding can reduce or eliminate the collateral received during an auction
- 4. Incentives to start auctions can cause gas prices to increase
- 5. Important price parameters can be changed during auctions or even during a call to take
- 6. No lower bound on the amount of collateral to buy
- 7. Insufficient validation of parameters during updates
- 8. Solidity compiler optimizations can be dangerous
- 9. Deployment scripts lack of proper validation and are error-prone

A. Vulnerability Classifications

B. Code Maturity Classifications

C. Code Quality

D. Magnitude Checking Using Slither

Executive Summary

From March 1st through March 19 2020, MakerDAO engaged Trail of Bits to review the security of their Liquidation 2.0 contracts. Trail of Bits conducted this assessment over the course of 6 person-weeks with 2 engineers. Initially working from commit `c8a134`, and then from commit `a4759e` as well as PRs 214 and 211 in the `makerdao/dss` repository and `9145de8` in the `makerdao/dss-deploy-scripts` repository.

During the first week, we focused on gaining an understanding of the codebase and started to review the `dog` contract for the most common Solidity flaws. During the second week, we focused on reviewing the `clip` operations to look for flaws that would allow an attacker to bypass access controls, disrupt or abuse the contracts in unexpected ways. The final week was dedicated to the review of the system properties as well as the more complex interactions between the liquidation contracts.

Our review resulted in 9 findings ranging from medium to informational in severity, where the most important category was missing or incorrect input validation. The most severe issues include an Integer division rounding that can reduce or eliminate the collateral received during an auction (TOB-DAI-LIQ-003) and insufficient validation of updated values during upgrades (TOB-DAI-LIQ-007, TOB-DAI-LIQ-009).

Other notable issues are related with unclear effects of economic incentives (TOB-DAI-LIQ-004) and the risks associated with the use of aggressive solc optimization (TOB-DAI-LIQ-008).

In addition to the security findings, we discuss code quality issues not related to any particular vulnerability in Appendix C. Finally, Appendix D describes a Slither script produced by Trail of Bits to check the use of magnitudes like WAD, RAY and RAD throughout the entire codebase.

Overall, the code follows a high-quality software development standard and best practices. It has a suitable architecture and is properly documented. However, we noted a lack of a proper code freeze despite an upcoming proposal for deployment. While these were only minor modifications, they can eventually accumulate to require another security assessment.

Trail of Bits recommends addressing the findings presented, expanding the property-based testing with more system invariants and performing formal verification if possible, before deploying the new liquidation contracts.

Project Dashboard

Application Summary

Name	MakerDAO Liquidations 2.0
Version	c8a134, a4759e, 9145de8, PRs 214 and 211
Type	Smart contracts
Platforms	Ethereum

Engagement Summary

Dates	March 1st through March 19 2020
Method	Whitebox
Consultants Engaged	2
Level of Effort	6 person-weeks

Vulnerability Summary

Total High-Severity Issues	0	
Total Medium-Severity Issues	3	■■■
Total Low-Severity Issues	3	■■■
Total Informational-Severity Issues	2	■■
Total Undetermined-Severity Issues	1	■
Total	9	

Category Breakdown

Data Validation	7	■■■■■■■
Timing	1	■
Undefined Behavior	1	■
Total	9	

Code Maturity Evaluation

In the table below, we review the maturity of the codebase and the likelihood of future issues. In each area of control, we rate the maturity from strong to weak, or missing, and give a brief explanation of our reasoning.

Category Name	Description
Access Controls	**Strong.** Appropriate access controls were in place for performing privileged operations by certain contracts.
Arithmetic	**Moderate.** The contracts included use of safe arithmetics. No overflows were possible in areas where these functions were not used. However, certain arithmetic operations in corner cases could produce unexpected results (TOB-DAI-LIQ-001, TOB-DAI-LIQ-003)
Assembly Use	**Strong.** The contracts only used assembly of the `rpow` function (`abaci.sol` contracts), which was formally verified prior to this audit.
Centralization	**Satisfactory.** While the protocol relied on an owner to correctly deploy the initial contracts, the ownership is renounced immediately after it. The privileged operations are handled through governance.
Contract Upgradeability	**Weak.** The contracts upgradeability mechanisms allow governance to change important parameters of each contract, however the update of such values can be error prone due to missing validations (TOB-DAI-LIQ-005, TOB-DAI-LIQ-007). Also, deployment scripts fail to properly validate system parameters (TOB-DAI-LIQ-009)
Function Composition	**Satisfactory.** While most of the functions and contracts were organized and scoped appropriately, the liquidation contracts are deeply coupled and could be refactored to simplify the access controls (Appendix C).
Front-Running	**Moderate.** While the front-running is an unavoidable part of auctions, we found it can impact negatively on the gas prices if it users are directly incentivized to call certain functions (TOB-DAI-LIQ-4)
Monitoring	**Satisfactory.** The events produced by the smart contract code were sufficient to monitor on-chain activity.
Specification	**Satisfactory.** Extensive documentation describing the functionality of the protocol was available as well as a list of system invariants. However, there was no formal specification available to review.

Testing & Verification	**Satisfactory.** The contracts included a sufficient number of unit tests, however it lacks more advanced testing techniques such as fuzzing or symbolic execution.

Engagement Goals

The engagement was scoped to provide a security assessment of the MakerDAO Liquidations 2.0 protocol in the `makerdao/dss` repository.

Specifically, we sought to answer the following questions:

- Do system components have appropriate access controls?
- Is it possible to manipulate the system by front-running transactions?
- Is it possible for participants to steal or lose tokens or collateral?
- Are there any instances where arithmetic overflow may affect the system?
- Can participants perform denial-of-service or phishing attacks against any of the system components?
- Can flash loans negatively affect the system's operation?
- Does arithmetic regarding liquidations and shutdown bookkeeping hold?

Oracle attacks were outside the scope of this assessment.

Coverage

The engagement was focused on the following components:

- **Dog contract:** the liquidation process starts when a user calls `dog.bark` indicating a vault to liquidate. This contract ensures that only vaults that are undercollateralized can be liquidated, confiscating their collateral to start an auction. We reviewed the soundness of the arithmetic, the accuracy of bookkeeping operations, and as well as access controls that ensured functionality could only be invoked by the appropriate core contracts. The review was performed manually and using automatic tools.
- **Clip contract:** The liquidation process is performed with a dutch auction that allows users to instantly buy a fraction or the total collateral on sale using the `clip.take` function. This function also allows users to buy without collateral, if they leverage DeFi applications like flash-loans. We reviewed this contract to ensure liquidations were carried out properly, especially in the event of price crashes and unexpected values in the vaults to liquidate. We also checked the access controls that ensured functionality could only be invoked by the appropriate core contracts. The review was performed manually and using automatic tools.
- **Abaci contracts:** These contracts specify price versus time as a function of the initial price of an auction and the time at which it was initiated. The implemented ones are linear, exponential and stairstep exponential decrease. We reviewed the soundness of the arithmetic as well as the accuracy to make sure they compute valid prices. The reviews were performed manually and using automatic tools.
- **Other DAI contracts**: the rest of the MakerDAO contracts included a minimal amount of modifications, the most relevant one was the addition of the `end.snip` function to cancel any active auctions during the shutdown. This code was manually reviewed to ensure it worked as expected.
- **Deployment scripts**: These are a set of bash scripts that deploy the DAI contracts to a certain Ethereum chain. We took a quick look to manually review how these are used and how potential errors are avoided using proper validation of critical system parameters.
- **Access controls.** Many parts of the system exposed privileged functionality that should only be invoked by other system contracts. We reviewed these functions to ensure they could only be triggered by the intended components and that they do not contain unnecessary privileges that may be abused.
- **Arithmetic.** We reviewed calculations for logical consistency, as well as rounding issues and scenarios where reverts due to overflow may negatively impact use of the protocol.

Off-chain code components were outside the scope of this assessment.

Automated Testing and Verification

Trail of Bits used automated testing techniques to enhance coverage of certain areas of the contracts, including:

- Slither, a Solidity static analysis framework. Slither can statically verify algebraic relationships between Solidity variables. We used Slither to detect invalid or inconsistent usage of the contracts' APIs across the code in scope. Additionally we wrote a custom Script that uses Slither to check for consistent and correct usage of magnitudes (WAD, RAY, RAD) within the code in scope. See Appendix D for details.
- Echidna, a smart contract fuzzer. Echidna can rapidly test security properties via malicious, coverage-guided test case generation. We used Echidna to test the expected system properties, including `dog` and `clip` operations and expected prices from the `abaci` contracts.

Automated testing techniques augment our manual security review but do not replace it. Each technique has limitations: Slither may identify security properties that fail to hold when Solidity is compiled to EVM bytecode; Echidna may not randomly generate an edge case that violates a property. To mitigate these risks, we had extended fuzzing campaigns of 36 hours with Echidna and then manually reviewed all the results.

Automated Testing with Echidna

We managed to collect and implement 29 Echidna properties.

Property	Result
If a vault is liquidable, then dog.bark cannot revert unexpectedly	**FAILED** (TOB-DAI-LIQ-001)
After a successful call to dog.bark, the resulting vault cannot be dusty	PASSED
After a successful call to dog.bark, if the vault was dusty, then the resulting vault is empty	PASSED
After a successful call to dog.bark, if the resulting vault has art == 0, then the resulting vault is empty	PASSED
If a vault was not liquidable or it was already liquidated, a call to dog.bark should fail	PASSED
In the context of a dog.bark execution, ink <= dink	PASSED

In the context of a dog.bark execution, art <= dart	PASSED
In the context of a dog.bark execution, tab > 0	PASSED
In the context of a dog.bark execution, dink > 0	PASSED
In the context of a dog.bark execution, usr cannot be 0x0	**FAILED** (TOB-DAI-LIQ-002)
If an auction is not active, calling clip.yank should fail	PASSED
If an auction is active, calling clip.yank should never fail and the resulting auction should be inactive	PASSED
If the redo flag is false, then clip.redo should fail	PASSED
After a successful call to clip.redo, the redo flag should be false and price cannot be zero.	PASSED
if the preconditions are met, clip.take should never revert	PASSED
In the context of a clip.take execution, tab >= owe	PASSED
In the context of a clip.take execution, lot >= slice	PASSED
In the context of a clip.take execution, if owe == 0 then slice == 0	PASSED
In the context of a clip.take execution, if slice == 0 then owe == 0	**FAILED (TOB-DAI-LIQ-003)**
The price computation in LinearDecrease never reverts unexpectedly	PASSED
The price computation in ExponentialDecrease never reverts unexpectedly	PASSED
The price computation in StairstepExponentialDecrease never reverts unexpectedly	PASSED
The value returned by price in LinearDecrease decreases over time	PASSED
The value returned by price in LinearDecrease is zero after tau	PASSED
The value returned by price in ExponentialDecrease decreases over time	PASSED

The value returned by price in ExponentialDecrease decays to zero	PASSED
The value returned by price in StairstepExponentialDecrease decreases over time	PASSED
The value returned by price in StairstepExponentialDecrease decays to zero	PASSED

When performing a long fuzzing campaign, Echidna was able to cover every reachable Solidity line in the `vat, dog, pip, spotter, abaci` contracts. It also covered every line in `clip`, expect for the following branches:

- `tab >= dust && mul(lot, price) >= dust` in line 295,
- `tab < dust` in line 377,
- `tab == 0` in line 410,

We manually reviewed these cases to make sure no property can fail.

Recommendations Summary

This section aggregates all the recommendations made during the engagement. Short-term recommendations address the immediate causes of issues. Long-term recommendations pertain to the development process and long-term design goals.

Short term

☐ **Properly validate all the external parameters such as `milk.chop`.** This will avoid unexpected reverts when starting an auction. TOB-DAI-LIQ-001

☐ **Either ensure that liquidations of the urn at `0x0` can succeed if the urn should be liquidated, disallow creating an urn at `0x0` or properly document this corner case.** This will avoid unexpected reverts. TOB-DAI-LIQ-002

☐ **Properly document that Integer division rounding can reduce or eliminate the collateral received during an auction.** This will make sure users are aware that the amount of collateral received can be reduced or zero**.** TOB-DAI-LIQ-003

☐ **Consider delaying the compensation obtained when starting a new auction, to add uncertainty, making it less attractive for front-runners.** This will make it impossible for common automated front-runners to simulate the effects of single transactions in order to find profits. TOB-DAI-LIQ-004

☐ **Consider recommending to restart all the active auctions when a governance vote that modifies important parameters is passed.** This will avoid auctions to use outdated parameters TOB-DAI-LIQ-005

☐ **Consider adding an additional check to specify a minimal amount of collateral to buy.** This will reduce the possibility of buying an amount of collateral that is smaller than the user expects. TOB-DAI-LIQ-006

☐ **Properly validate important parameters to ensure that parameter updates can not stop contracts from functioning correctly under any circumstances.** In particular, disallow setting addresses to `0x0` and add `require(clip.ilk == ilk)` to the `if` body in the second line of the `file` function in *Figure 7.2*. TOB-DAI-LIQ-007

☐ **Measure the gas savings from optimizations, and carefully weigh them against the possibility of an optimization-related bug.** This will mitigate the possibility of deploying incorrect code. TOB-DAI-LIQ-008

☐ **Properly validate config values before executing any deployment transactions.** This will avoid deploying contracts with invalid parameters. TOB-DAI-LIQ-009

Long term

☐ **Review the preconditions of every external call and make sure they are considered when the system invariants are defined.** Use Echidna or Manticore to test them. TOB-DAI-LIQ-001, TOB-DAI-LIQ-002

☐ **Review the rounding effect of integer division in your code base and bound the remainder.** Use Echidna or Manticore to test that the bounds are tight. TOB-DAI-LIQ-003

☐ **Review the economic incentives of the system.** This will make sure they produce the intended effects**.** TOB-DAI-LIQ-004

☐ **Investigate and specify the effects of parameter changes during processes involving multiple transactions that depend on those parameters.** This will ensure there are no undesired effects when parameters are changed while interactions depending on them are in progress. TOB-DAI-LIQ-005

☐ **Review the usability of each function and make sure you mitigate each potential issue.** TOB-DAI-LIQ-006

☐ **Add more data validation to guard against misconfiguration.** This will prevent the system from losing funds or entering an inconsistent state. TOB-DAI-LIQ-007, TOB-DAI-LIQ-009

☐ **Monitor the development and adoption of Solidity compiler optimizations to assess their maturity.** This will allow to use optimizations when the risks are reduced TOB-DAI-LIQ-008

Findings Summary

#	Title	Type	Severity
1	Very low values of `milk.chop` can block liquidations	Data Validation	Low
2	An urn at the 0x0 address cannot be liquidated	Data Validation	Informational
3	Integer division rounding can reduce or eliminate the collateral received during an auction	Data Validation	Medium
4	Incentives to start auctions can cause gas prices to increase	Timing	Low
5	Important price parameters can be changed during auctions or even during a call to take	Data Validation	Low
6	No lower bound on the amount of collateral to buy	Data Validation	Informational
7	Insufficient validation of parameters during updates	Data Validation	Medium
8	Solidity compiler optimizations can be dangerous	Undefined Behavior	Undetermined
9	Deployment scripts lack of proper validation and are error-prone	Data Validation	Medium

1. Very low values of `milk.chop` can block liquidations

Severity: Low
Type: Data Validation
Target: `dog.sol, clip.sol`

Difficulty: High
Finding ID: TOB-DAI-LIQ-001

Description
Certain parameters used during the first liquidation step can block liquidations.

Any user can start the new liquidation procedure with a call to the `dog.bark` function specifying an ilk, an urn and keeper address. This function will invoke `clip.kick` with certain values:

```
function bark(bytes32 ilk, address urn, address kpr) external returns (uint256 id) {
    ...

    {   // Avoid stack too deep
        // This calcuation will overflow if dart*rate exceeds ~10^14
        uint256 tab = mul(due, milk.chop) / WAD;
        Dirt = add(Dirt, tab);
        ilks[ilk].dirt = add(milk.dirt, tab);

        id = ClipperLike(milk.clip).kick({
            tab: tab,
            lot: dink,
            usr: urn,
            kpr: kpr
        });
    }

    emit Bark(ilk, urn, dink, dart, due, milk.clip, id);
}
```

Figure 1.1: part of the `bark` *function in* `dog.sol`.

The `kick` function has its own preconditions:

```
function kick(
    uint256 tab,  // Debt                   [rad]
    uint256 lot,  // Collateral             [wad]
    address usr,  // Liquidated CDP
    address kpr   // Keeper that called dog.bark()
) external auth lock isStopped(1) returns (uint256 id) {
    // Input validation
    require(tab  >          0, "Clipper/zero-tab");
    ...
}
```

Figure 1.2: part of the `kick` *function in* `clip.sol`.

However, if the `tab` input is 0, the call to `kick` will revert, causing the first step of the liquidation to fail. This can be caused by a zero or very small value of `milk.chop`.

Exploit Scenario
A governance vote updates certain parameters, including `milk.chop` with an incorrect one (e.g. zero). This goes unnoticed until a large collateral price drop triggers some liquidations. Users calling `dog.bark` could have their transactions reverting unexpectedly. As a result of that, a new governance vote must be done as soon as possible to update the incorrect value when liquidations are needed.

Recommendation
Short term, properly validate all the external parameters such as `milk.chop`. This will avoid unexpected reverts when starting an auction.

Long term, review the preconditions of every external call and make sure they are considered when the system invariants are defined. Use Echidna or Manticore to test them.

2. An urn at the 0x0 address cannot be liquidated

Severity: Informational
Type: Data Validation
Target: `dog.sol, clip.sol`

Difficulty: High
Finding ID: TOB-DAI-LIQ-002

Description
An urn at the 0x0 address cannot be liquidated.

Any user can start the new liquidation procedure with a call to the `dog.bark` function specifying an ilk, an urn and keeper address. This function will invoke `clip.kick` with certain values:

```
function bark(bytes32 ilk, address urn, address kpr) external returns (uint256 id) {
    ...

    {   // Avoid stack too deep
        // This calcuation will overflow if dart*rate exceeds ~10^14
        uint256 tab = mul(due, milk.chop) / WAD;
        Dirt = add(Dirt, tab);
        ilks[ilk].dirt = add(milk.dirt, tab);

        id = ClipperLike(milk.clip).kick({
            tab: tab,
            lot: dink,
            usr: urn,
            kpr: kpr
        });
    }

    emit Bark(ilk, urn, dink, dart, due, milk.clip, id);
}
```

Figure 2.1: part of the `bark` *function in* `dog.sol`.

The `kick` function has its own preconditions:

```
function kick(
    uint256 tab,  // Debt                   [rad]
    uint256 lot,  // Collateral             [wad]
    address usr,  // Liquidated CDP
    address kpr   // Keeper that called dog.bark()
) external auth lock isStopped(1) returns (uint256 id) {
    // Input validation
    require(tab   >           0, "Clipper/zero-tab");
    require(lot   >           0, "Clipper/zero-lot");
    require(usr   !=  address(0), "Clipper/zero-usr");
    ...
}
```

Figure 2.2: part of the `kick` *function in* `clip.sol`.

However, if the `urn` address is 0x0, the call to `kick` will revert, causing the first step of the liquidation to fail. The frob function allows any user to create a non-null urn at the 0x0 address.

Exploit Scenario
Alice calls `frob` to create an urn at `u = 0x0`. If the urn ever becomes unsafe and a keeper calls `dog.bark(..., 0x0, …)` to initialize liquidation, it will fail because `clip.kick(…, …, 0x0, …)` always reverts. As a result of that, an urn that can't be liquidated exists in the system.

Recommendation
Short term, either ensure that liquidations of the urn at `0x0` can succeed if the urn should be liquidated, disallow creating an urn at `0x0` or properly document this corner case. This will avoid unexpected reverts.

Long term, review the preconditions of every external call and make sure they are considered when the system invariants are defined. Use Echidna or Manticore to test them.

3. Integer division rounding can reduce or eliminate the collateral received during an auction

Severity: Medium
Type: Data Validation
Target: `clip.sol`

Difficulty: High
Finding ID: TOB-DAI-LIQ-003

Description
Under certain circumstances, the computation of the amount of collateral received during an auction can be reduced or even eliminated due to integer division rounding.

Once an auction is started, users can call `clip.take` in order to buy a certain amount of collateral using DAI:

```
      function take(
        uint256 id,           // Auction id
        uint256 amt,          // Upper limit on amount of collateral to buy  [wad]
        uint256 max,          // Maximum acceptable price (DAI / collateral) [ray]
        address who,          // Receiver of collateral and external call address
        bytes calldata data   // Data to pass in external call; if length 0, no call is done
    ) external lock isStopped(2) {

            ...
            // Purchase as much as possible, up to amt
            uint256 slice = min(lot, amt);  // slice <= lot

            // DAI needed to buy a slice of this sale
            owe = mul(slice, price);

            // Don't collect more than tab of DAI
            if (owe > tab) {
                // Total debt will be paid
                owe = tab;                  // owe' <= owe
                // Adjust slice
                slice = owe / price;        // slice' = owe' / price <= owe / price == slice
<= lot
            } else if (owe < tab && slice < lot) {
                // if slice == lot => auction completed => dust doesn't matter
                (,,,, uint256 dust) = vat.ilks(ilk);
                if (tab - owe < dust) {     // safe as owe < tab
                    // if tab <= dust, buyers have to buy the whole thing
                    require(tab > dust, "Clipper/no-partial-purchase");
                    // Adjust amount to pay
                    owe = tab - dust;       // owe' <= owe
                    // Adjust slice
                    slice = owe / price;    // slice' = owe' / price < owe / price == slice
< lot
                }
            }

            // Calculate remaining tab after operation
            tab = tab - owe;  // safe since owe <= tab
            // Calculate remaining lot after operation
            lot = lot - slice;
```

```
            ...
    }
```

Figure 3.1: part of the `take` *function in* `clip.sol`.

However, if the total debt to be collected in the auction (`tab`) minus the `dust` is lower than the `price`, then the debt to be offset (`owe`) is lower than the `price`. As a result the computation of received collateral (`slice`) in the highlighted line will return zero despite the debt to be offset (`owe`) being positive. This is due to truncating integer division.

Exploit Scenario
Alice wants to buy collateral from live auctions. Instead of getting as much as possible from a single auction, Alice takes a smaller amount from different auctions. As a result, the integer division rounding can significantly reduce the amount of collateral received. In extreme cases, it can be even zero.

Recommendation
Short term, properly document this corner case to make sure users are aware that the amount of collateral received can be reduced or zero.

Long term, review the rounding effect of integer division in your code base and bound the remainder. Use Echidna or Manticore to test that the bounds are tight.

4. Incentives to start auctions can cause gas prices to increase

Severity: Low
Type: Timing
Target: `clip.sol`

Difficulty: Low
Finding ID: TOB-DAI-LIQ-004

Description
The use of rewards to (re)start auctions can have unintended consequences on gas prices.

Any user can start the new liquidation procedure with a call to the `dog.bark` function specifying an ilk, an urn and keeper address. This function will invoke `clip.kick`:

```
    function kick(
        uint256 tab,  // Debt                   [rad]
        uint256 lot,  // Collateral             [wad]
        address usr,  // Liquidated CDP
        address kpr   // Keeper that called dog.bark()
    ) external auth lock isStopped(1) returns (uint256 id) {
        ...

        // incentive to kick auction
        uint256 _tip  = tip;
        uint256 _chip = chip;
        uint256 coin;
        if (_tip > 0 || _chip > 0) {
            coin = add(_tip, wmul(tab, _chip));
            vat.suck(vow, kpr, coin);
        }

        emit Kick(id, top, tab, lot, usr, kpr, coin);
    }
```

Figure 4.1: part of the `kick` *function in* `clip.sol`.

This function includes the immediate distribution of rewards to the `kpr` address passed to `dog.bark` if `tip > 0` or `chip > 0`.

However, since the call to start auctions is front-runnable, the incentives can leave a profitable margin for automated front-runners. If there is a large number of auctions opened by front-runners with higher gas prices, it can result in gas prices temporarily increasing, causing the opposite effect as the one expected.

Exploit Scenario
Alice wants to get the reward for starting new auctions. She sends her transactions to start liquidations. Front-running bots looking for the same reward produce a temporary increase in the gas cost. Alice is forced to increase the gas price of her transaction, offsetting the incentive received.

Recommendation

Short term, consider delaying the compensation obtained when starting a new auction, to add uncertainty, make it less attractive for front-runners and impossible to detect for common automated front runners that simulate the effects of single transactions to find profits.

Long term, review the economic incentives of the system to make sure they produce the intended effects.

5. Important price parameters can be changed during auctions or even during a call to take

Severity: Low
Type: Data Validation
Target: `clip.sol`

Difficulty: High
Finding ID: TOB-DAI-LIQ-005

Description
Important auction parameters can be changed while auctions depending on them are in progress; with unclear effects.

The `clip` contract computes auction prices and keeps track of auction progress using some important parameters such as `buf`, `tail` and `cup`.

```
    function kick(
        uint256 tab,  // Debt                   [rad]
        uint256 lot,  // Collateral             [wad]
        address usr,  // Liquidated CDP
        address kpr   // Keeper that called dog.bark()
    ) external auth lock isStopped(1) returns (uint256 id) {
        ...
        uint256 top;
        top = rmul(getPrice(), buf);
        require(top > 0, "Clipper/zero-top-price");
        sales[id].top = top;
        ...
    }
```

Figure 5.1: part of the `kick` *function in* `clip.sol`.

Such parameters can be changed by governance using the `file` function:

```
    // --- Administration ---
    function file(bytes32 what, uint256 data) external auth lock {
        if      (what == "buf")         buf = data;
        else if (what == "tail")       tail = data;           // Time elapsed before auction
reset
        else if (what == "cusp")       cusp = data;           // Percentage drop before
auction reset
        else if (what == "chip")       chip = uint64(data);   // Percentage of tab to
incentivize (max: 2^64 - 1 => 18.xxx WAD = 18xx%)
        else if (what == "tip")         tip = uint192(data);  // Flat fee to incentivize
keepers (max: 2^192 - 1 => 6.277T RAD)
        else if (what == "stopped") stopped = data;           // Set breaker (0, 1 or 2)
        else revert("Clipper/file-unrecognized-param");
        emit File(what, data);
    }
```

Figure 5.2: `file` *function in* `clip.sol`.

However, it is not clear what are the effects for the users participating in the auctions, since these are not going to be restarted after the parameters are changed. Moreover, if a

callback is used in `take`, It is possible to trigger the change of `vat.dust` in the middle of the execution.

```
      function take(
        uint256 id,           // Auction id
        uint256 amt,          // Upper limit on amount of collateral to buy  [wad]
        uint256 max,          // Maximum acceptable price (DAI / collateral) [ray]
        address who,          // Receiver of collateral and external call address
        bytes calldata data   // Data to pass in external call; if length 0, no call is done
    ) external lock isStopped(2) {

            ...
            // Purchase as much as possible, up to amt
            uint256 slice = min(lot, amt);  // slice <= lot

            // DAI needed to buy a slice of this sale
            owe = mul(slice, price);

            // Don't collect more than tab of DAI
            if (owe > tab) {
                // Total debt will be paid
                owe = tab;                  // owe' <= owe
                // Adjust slice
                slice = owe / price;        // slice' = owe' / price <= owe / price == slice
<= lot
            } else if (owe < tab && slice < lot) {
                // if slice == lot => auction completed => dust doesn't matter
                (,,,, uint256 dust) = vat.ilks(ilk);
                if (tab - owe < dust) {     // safe as owe < tab
                    // if tab <= dust, buyers have to buy the whole thing
                    require(tab > dust, "Clipper/no-partial-purchase");
                    // Adjust amount to pay
                    owe = tab - dust;       // owe' <= owe
                    // Adjust slice
                    slice = owe / price;    // slice' = owe' / price < owe / price == slice
< lot
                }
            }

            ...
            if (data.length > 0 && who != address(vat) && who != address(dog_)) {
                ClipperCallee(who).clipperCall(msg.sender, owe, slice, data);
            }
            ...

    }
```

Figure 5.3: part of the `take` *function in* `clip.sol`.

This will allow to compute `owe` with an old value of `dust`, which means that the use of this number is technically incorrect and could violate the invariant that says that no partial purchase of dusty vaults can be performed.

Exploit Scenario

Alice starts an auction. The computation of the top price is performed using the current buf value. A governance vote is performed after that, causing the buf value to be decreased. Users are forced to pay more for the collateral in the auction that Alice started, since the top value is not checked after it started.

Recommendation
Short term, consider recommending to restart all the active auctions when a governance vote that modifies important parameters is passed.

Long term, investigate and specify the effects of parameter changes during processes involving multiple transactions that depend on those parameters.

6. No lower bound on the amount of collateral to buy

Severity: Informational
Type: Data Validation
Target: `clip.sol`

Difficulty: Low
Finding ID: TOB-DAI-LIQ-006

Description
There is no guarantee that a minimal amount of collateral will be bought when calling `take`, putting users in risk of paying for unprofitable liquidations.

Once an auction was started, users should call `clip.take` in order to buy a certain amount of collateral using DAI:

```
    function take(
      uint256 id,           // Auction id
      uint256 amt,          // Upper limit on amount of collateral to buy  [wad]
      uint256 max,          // Maximum acceptable price (DAI / collateral) [ray]
      address who,          // Receiver of collateral and external call address
      bytes calldata data   // Data to pass in external call; if length 0, no call is done
  ) external lock isStopped(2) {
      ...
      // Ensure price is acceptable to buyer
      require(max >= price, "Clipper/too-expensive");

      uint256 lot = sales[id].lot;
      uint256 tab = sales[id].tab;
      uint256 owe;

      {
          // Purchase as much as possible, up to amt
          uint256 slice = min(lot, amt);  // slice <= lot

      ...
}
```

Figure 6.1: part of the `take` *function in* `clip.sol`.

Before performing the liquidation, this function checks the price to verify that it is acceptable to the buyer. Also, it determines the amount of collateral to buy. While there is an upper limit for the amount of collateral to buy, there is no lower limit for that. This can cause users to buy amounts of collateral that are smaller than expected.

Exploit Scenario
Alice and Bob want to buy collateral from live auctions with different strategies. While Alice tries to get as much amount as possible from a single auction to be profitable, Bob tries to take smaller amounts from different ones. If Alice's transaction succeeds, the call to `take` will not allow Bob to get collateral, since there will be not enough left for him. However, if Bob's transaction succeeds, the call to `take` will still allow Alice to buy collateral, since there

is only an upper limit to the collateral. So, Alice can get a small and unprofitable amount of collateral, wasting gas on the transaction.

Recommendation
Short term, consider adding an additional check to specify a minimal amount of collateral to buy.

Long term, review the usability of each function and make sure you mitigate each potential issue.

7. Insufficient validation of parameters during updates

Severity: Medium
Type: Data Validation
Target: `clip.sol, dog.sol`

Difficulty: High
Finding ID: TOB-DAI-LIQ-007

Description
The parameter updates in liquidation contracts, which are performed by governance, lack basic validation where obvious mistakes are not caught.

There are two notable issues when performing parameter updates in the `clip` and `dog` contracts.

1. Any privileged user of the `clip` can call `clip.file(what, data)` with `data == 0x0`:

```
    function file(bytes32 what, address data) external auth lock {
        if (what == "spotter") spotter = SpotterLike(data);
        else if (what == "dog")    dog = DogLike(data);
        else if (what == "vow")    vow = data;
        else if (what == "calc")  calc = AbacusLike(data);
        else revert("Clipper/file-unrecognized-param");
        emit File(what, data);
    }
```

Figure 7.1: `file` *function in* `clip.sol`.

The result can block the normal usage of liquidations.

2. Any privileged user of the `Dog` can call `Dog.file(ilk, "clip", clip)` with `clip.ilk != ilk`.

This results in the `Dog` using a Clipper to handle collateral types which Clipper is not supposed to handle. This can at least prevent the necessary selling of collateral, cause the unnecessary selling of collateral, and cause the accounting to become inconsistent.

```
    function file(bytes32 ilk, bytes32 what, address clip) external auth {
        if (what == "clip") ilks[ilk].clip = clip;
        else revert("Dog/file-unrecognized-param");
        emit File(ilk, what, clip);
    }
```

Figure 7.2: part of the `file` *function in* `dog.sol`.

Exploit Scenario

1. Alice deploys a clipper `WBTCClipper` and configures it to handle a new collateral type WBTC. To complete adding WBTC to the system, she calls `Dog.file(ilk, "clip", WBTCClipper)`. Alice mistakenly chooses `ilk` to be the wrong 32 byte vector, the one for ETH. This mistake goes unnoticed.

2. Users open vaults backed by WBTC
3. Sometime later token WBTC has risen to the price of $100k. ETH falls to $1000 causing keepers to call `Dog.bark` for ETH backed vaults. Due to the misconfiguration `Dog.bark` calls `clip.kick` with the Clipper that is configured for WBTC. Because of the price difference this results in large amounts of WBTC being auctioned off, despite all WBTC vaults being overcollateralized. It also results in the system not getting rid of undercollateralized ETH. All of this causes the MCD system to lose funds and causes `vat` accounting to become inconsistent.

Additionally `clip.take` calls `dog.digs` with `ilk` WBTC. This decreases the `dirt` for WBTC which has previously been increased for ETH, thereby causing the accounting of `Dog` to be inconsistent.

Recommendation
Short term, properly validate important parameters to ensure that parameter updates can not stop contracts from functioning correctly under any circumstances. In particular, disallow setting addresses to `0x0` and add `require(clip.ilk == ilk)` to the `if` body in the second line of the `file` function in *Figure 7.2*.

Long term, add more data validation to guard against misconfiguration that can cause the system to lose funds or to enter an inconsistent state.

8. Solidity compiler optimizations can be dangerous

Severity: Undetermined
Type: Undefined Behavior
Target: Makefile

Difficulty: Low
Finding ID: TOB-DAI-LIQ-008

Description
MakerDAO Liquidations 2.0 has enabled optional compiler optimizations in Solidity and increased their runs from 200 to 1000000.

There have been several bugs with security implications related to optimizations. Solidity compiler optimizations are disabled by default in 0.6.12, and it is unclear how many contracts in the wild actually use them. Therefore, it is unclear how well they are being tested and exercised.

High-severity security issues due to optimization bugs have occurred in the past. A high-severity bug in the `emscripten`-generated `solc-js` compiler used by Truffle and Remix persisted until late 2018. The fix for this bug was not reported in the Solidity CHANGELOG. Another high-severity optimization bug resulting in incorrect bit shift results was patched in Solidity 0.5.6.

A compiler audit of Solidity from November, 2018 concluded that the optional optimizations may not be safe. Moreover, the Common Subexpression Elimination (CSE) optimization procedure is "implemented in a very fragile manner, with manual access to indexes, multiple structures with almost identical behavior, and up to four levels of conditional nesting in the same function." Similar code in other large projects has resulted in bugs.

There are likely latent bugs related to optimization, and/or new bugs that will be introduced due to future optimizations.

Exploit Scenario
A latent or future bug in Solidity compiler optimizations—or in the Emscripten transpilation to `solc-js`—causes a security vulnerability in the MakerDAO contracts.

Recommendation
Short term, measure the gas savings from optimizations, and carefully weigh them against the possibility of an optimization-related bug. This will mitigate the possibility of deploying incorrect code.

Long term, monitor the development and adoption of Solidity compiler optimizations to assess their maturity.

9. Deployment scripts lack of proper validation and are error-prone

Severity: Medium
Type: Data Validation
Target: `dss-deploy-scripts`

Difficulty: High
Finding ID: TOB-DAI-LIQ-009

Description
The deployments scripts implemented in bash lack basic validation where obvious mistakes are not caught.

The MakerDAO team developed a set of bash scripts to compile, deploy and verify permissions of the DAI contracts. The parameter of the deployments are specified in a JSON file:

```
{
  "description": "Mainnet deployment",
  "pauseDelay": "0",
  "vat_line": "778000000",
  "vow_wait": "561600",
  "vow_dump": "250",
  "vow_sump": "50000",
  "vow_bump": "10000",
  "vow_hump": "500000",
  "cat_box": "10000000",
  "dog_hole": "10000000",
  "jug_base": "0",
  "pot_dsr": "0",
  "end_wait": "262800",
  …
}
```

Figure 9.1: header of `mainnet.json` *config file*.

The numeric values are specified as strings, which is correct, however, there are little or no checks that the values are valid. For instance, the `set-dog-hole` script only provides minimal validation:

```
# Get config variables
CONFIG_FILE="$OUT_DIR/config.json"
# Get addresses
loadAddresses

log "SET DOG HOLE:"

Hole=$(jq -r ".dog_hole | values" "$CONFIG_FILE")
if [[ "$Hole" != "" && "$Hole" != "0" ]]; then
    Hole=$(echo "$Hole"*10^45 | bc)
    Hole=$(seth --to-uint256 "${Hole%.*}")
    calldata="$(seth calldata 'file(address,address,address,bytes32,uint256)' "$MCD_PAUSE"
"$MCD_GOV_ACTIONS" "$MCD_DOG" "$(seth --to-bytes32 "$(seth --from-ascii "Hole")")" "$Hole")"
    sethSend "$PROXY_DEPLOYER" 'execute(address,bytes memory)' "$PROXY_PAUSE_ACTIONS"
"$calldata"
fi
```

Figure 9.2: part of the `set-dog-hole` script.

In that script, using invalid values have unexpected effects:

- If "abc" is used, `bc` will interpret it as a variable with value 0 as default, so the resulting `hole` will 0.
- If "O01" is used (where the first character is a letter O instead of zero) bc will silently convert it to a 9, and the resulting `hole` will be: 90100000000000000000000000000000000000000000000

Other scripts like `set-ilks-dust` do not perform any validation:

```
tokens=$(jq -r ".tokens | keys_unsorted[]" "$CONFIG_FILE")
for token in $tokens; do
    ilks=$(jq -r ".tokens.${token}.ilks | keys_unsorted[]" "$CONFIG_FILE")
    for ilk in $ilks; do
        dust=$(jq -r ".tokens.${token}.ilks.${ilk} | .dust" "$CONFIG_FILE")
        dust=$(echo "$dust"*10^45 | bc)
        dust=$(seth --to-uint256 "${dust%.*}")
        calldata="$(seth calldata 'file(address,address,address,bytes32,bytes32,uint256)'
"$MCD_PAUSE" "$MCD_GOV_ACTIONS" "$MCD_VAT" "$(seth --to-bytes32 "$(seth --from-ascii
"${token}-${ilk}")")" "$(seth --to-bytes32 "$(seth --from-ascii "dust")")" "$dust")"
        sethSend "$PROXY_DEPLOYER" 'execute(address,bytes memory)' "$PROXY_PAUSE_ACTIONS"
"$calldata"
    done
done
```

Figure 9.3: part of the `set-ilks-dust` script.

Exploit Scenario
The MakerDAO team prepares a contract upgrade but accidentally introduces an invalid character in the json file used for configuration. As a result of that, the contracts are deployed with invalid or unexpected parameters.

Recommendation
Short term, properly validate config values before executing any deployment transactions.

Long term, add more data validation to guard against misconfiguration that can cause the system to lose funds or to enter an inconsistent state.

A. Vulnerability Classifications

Vulnerability Classes	
Class	**Description**
Access Controls	Related to authorization of users and assessment of rights
Auditing and Logging	Related to auditing of actions or logging of problems
Authentication	Related to the identification of users
Configuration	Related to security configurations of servers, devices, or software
Cryptography	Related to protecting the privacy or integrity of data
Data Exposure	Related to unintended exposure of sensitive information
Data Validation	Related to improper reliance on the structure or values of data
Denial of Service	Related to causing system failure
Error Reporting	Related to the reporting of error conditions in a secure fashion
Patching	Related to keeping software up to date
Session Management	Related to the identification of authenticated users
Testing	Related to test methodology or test coverage
Timing	Related to race conditions, locking, or order of operations
Undefined Behavior	Related to undefined behavior triggered by the program
<other class>	<add any other class which is missing for the audit, delete the row if everything fits into prior ones>

Severity Categories	
Severity	**Description**
Informational	The issue does not pose an immediate risk, but is relevant to security best practices or Defense in Depth
Undetermined	The extent of the risk was not determined during this engagement

Low	The risk is relatively small or is not a risk the customer has indicated is important
Medium	Individual user's information is at risk, exploitation would be bad for client's reputation, moderate financial impact, possible legal implications for client
High	Large numbers of users, very bad for client's reputation, or serious legal or financial implications

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploit was not determined during this engagement
Low	Commonly exploited, public tools exist or can be scripted that exploit this flaw
Medium	Attackers must write an exploit, or need an in-depth knowledge of a complex system
High	The attacker must have privileged insider access to the system, may need to know extremely complex technical details, or must discover other weaknesses in order to exploit this issue

B. Code Maturity Classifications

Code Maturity Classes	
Category Name	**Description**
Access Controls	Related to the authentication and authorization of components.
Arithmetic	Related to the proper use of mathematical operations and semantics.
Assembly Use	Related to the use of inline assembly.
Centralization	Related to the existence of a single point of failure.
Upgradeability	Related to contract upgradeability.
Function Composition	Related to separation of the logic into functions with clear purpose.
Front-Running	Related to resilience against front-running.
Key Management	Related to the existence of proper procedures for key generation, distribution, and access.
Monitoring	Related to use of events and monitoring procedures.
Specification	Related to the expected codebase documentation.
Testing & Verification	Related to the use of testing techniques (unit tests, fuzzing, symbolic execution, etc.).

Rating Criteria	
Rating	**Description**
Strong	The component was reviewed and no concerns were found.
Satisfactory	The component had only minor issues.
Moderate	The component had some issues.
Weak	The component led to multiple issues; more issues might be present.
Missing	The component was missing.

Not Applicable	The component is not applicable.
Not Considered	The component was not reviewed.
Further Investigation Required	The component requires further investigation.

C. Code Quality

The following recommendations are not associated with specific vulnerabilities. However, they enhance code readability and may prevent the introduction of vulnerabilities in the future.

General

- **Consider refactoring the `dog` and `clip` contracts in a single one.** These two contracts are tightly coupled and can only work with each other. They need special values and permissions that could be avoided if it was a single contract. This will make the code easier to understand, maintain, and review.

Clip.sol:

- **Correct the comment that documents the `kpr` parameter of the `kick` function.** The comment that says "Keeper that called dog.bark()" is incorrect since any address can call the `dog.bark` address, and therefore, indirectly call `kick`. This will make the code easier to understand, maintain, and review.
- **Consider renaming the `usr` param of `clip.kick` to `urn` for consistency.**
    - The vault address is called `urn` in params of `dog.bark.`
    - The vault address is called `usr` in params of `clip.kick.`

D. Magnitude Checking Using Slither

During the audit, Trail of Bits developed a Script which processes the output of the static analysis done by Slither. It uses the output of the static analysis to check for correct usage of the RAD, RAY and WAD magnitudes.

The script keeps track of magnitudes across operations. It then checks that functions are always called with specific correct magnitude combinations and values assigned to variables always have specific magnitudes. For example, any variable called `ink` should have the magnitude `WAD` (`10**18`). As another example, `min(a, b)` must only be called with variables of the same magnitude and will return a value of that magnitude. No incorrect uses of magnitudes were found.

The magnitude checker script was implemented ad-hoc for MakerDAO and doesn't guarantee coverage. Trail of Bits will deliver a proof-of-concept of it and iterate on implementation to address these shortcomings and open-source it in future.

Following are examples of how the expected magnitudes are specified:

```
WAD = ConstantMag(10 ** 18)
RAY = ConstantMag(10 ** 27)
RAD = ConstantMag(10 ** 45)
BLN = ConstantMag(10 ** 9)

T = GenericMag("T")

MAKER_MATH: Dict[str, FuncSpec] = {
    "min": FuncSpec([T, T], [T]),
    "sub": FuncSpec([T, T], [T]),
    "add": FuncSpec([T, T], [T]),
    # https://github.com/makerdao/dss/blob/master/DEVELOPING.md#multiplication
    "mul": FuncSpec([WAD, RAY], [RAD]),
    "rmul": FuncSpec([WAD, RAY], [WAD]).overload([RAY, RAY], [RAY]).overload([RAD, RAY],
[RAD]),
```

Figure C.1: part of the `MAKER_MATH` *dictionary in* `maker-magcheck.py`.

```
        "kick": FuncSpec([RAD, WAD, NoMag, NoMag], [NoMag]),
        "getPrice": FuncSpec([], [RAY]),
        "redo": IgnoreFunc(),
        "take": FuncSpec([NoMag, WAD, RAY, NoMag, NoMag]),
        "_remove": IgnoreFunc(),
        "count": IgnoreFunc(),
        "getId": IgnoreFunc(),
        "getStatus": FuncSpec([NoMag], [NoMag, RAY]),
        "status": FuncSpec([NoMag, RAY], [NoMag, RAY]),
        "yank": IgnoreFunc(),
```

Figure C.2: part of the `CONTRACT_TO_FUNC_TO_MAGS` *dictionary in* `maker-magcheck.py`.

```
NAME_IMPLIES_MAG: Dict[str, BaseMag] = {
    "ink": WAD,
    "art": WAD,
    "dart": WAD,
    "rate": RAY,
    "dust": RAD,
    "top": RAY,
    "Dirt": RAD,
    "Hole": RAD,
```

Figure C.3: part of the `NAME_IMPLIES_MAG` *dictionary in* `maker-magcheck.py`.