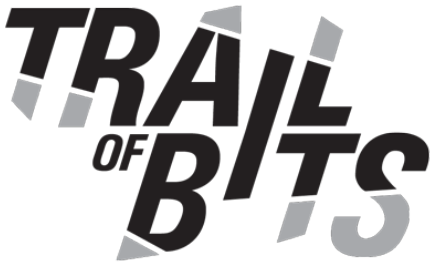

MCD Core Smart Contracts

Security Assessment

August 30, 2019

Prepared For:
Chris Smith | *Maker Foundation*
chriss@makerdao.com

Mariano Conti | *Maker Foundation*
mariano@makerdao.com

Prepared By:
JP Smith | *Trail of Bits*
jp@trailofbits.com

Rajeev Gopalakrishna | *Trail of Bits*
rajeev@trailofbits.com

Sam Moelius | *Trail of Bits*
sam.moelius@trailofbits.com

David Pokora | *Trail of Bits*
david.pokora@trailofbits.com

Changelog:
August 30, 2019: Initial delivery to Maker Foundation
October 3, 2019: Public release

Executive Summary

From July 29 through August 30, 2019, Maker Foundation engaged with Trail of Bits to review the security of the Multi-Collateral Dai (MCD) Core Smart Contracts. Trail of Bits conducted this assessment over the course of 16 person-weeks, with four engineers working from the repositories/commits listed under Coverage below.

During the first week, we focused on familiarizing ourselves with the codebase and ensuring that we could deploy and interact with the system. We ran Slither over all the contracts and began manually reviewing them. We also began work on Slither extensions to identify Solidity functions not covered by a klab specification.

During the second week, we manually reviewed code involving permissions (e.g., DSAuth, DSAuthority, DSRoles, `scripts/auth-checker`), ERC20 tokens (e.g., DSToken), and the Dai Savings Rate (e.g., Pot). We completed triage of the Slither results, and we enhanced our Slither extensions to verify that all of the MCD's uses of ABIEncoderV2 are covered by a klab specification. Finally, we began extending Echidna to verify that properties hold across gaps in block numbers and block times, up to user-determined bounds.

During the third week, we:

- Finished manually reviewing code that involves permissions.
- Began subjecting the Vat contract to multiple symbolic transactions using Manticore.
- Began manually reviewing the price oracle infrastructure (e.g., the median and oracle security manager contracts).
- Manually reviewed the uses of `ecrecover`.

During the fourth week, we:

- Improved our Manticore testing framework by expanding the contracts it analyzes and tailoring property tests to MCD.
- Continued our analysis of the DSR and its implementation.
- Continued to examine the price oracle infrastructure, including its use of Omnia, Setzer, and the Scuttlebutt protocol.
- Continued to extend Echidna.

During the fifth week, we continued to push our Manticore testing framework by tweaking parameters to expand code coverage and implementing additional MCD-specific property tests. We also revisited the deployment code (as requested by the client). In particular, we manually introduced errors into the deployment scripts and verified that the errors were caught and reported. Finally, we completed work on our Echidna extensions.

Many of our findings concern unanticipated interactions among multiple contracts (e.g., TOB-MCD-014 and several entries in Appendix B: Non-security related findings). An additional category of findings might be called “unconventional” or “unanticipated” interaction with the outside world (e.g., transactions made in an unexpected way (TOB-MCD-007, TOB-MCD-010), or in an unexpected order (TOB-MCD-001, TOB-MCD-008, TOB-MCD-009). Compared to other assessments, relatively few findings involved data validation (only TOB-MCD-011 and TOB-MCD-012). This is perhaps not surprising, as the use of klab seems to have eliminated many of these types of vulnerabilities.

While we feel the use of short names is a detriment to understanding the code, we found the code overall to be clean and easy to read. The MCD 101 document was invaluable in helping us understand how the contracts work, and the Glossary also helped significantly.

The use of klab seems to have eliminated much of the “low-hanging fruit” in terms of vulnerabilities, and we recommend its continued use. However, since the klab specification must be produced by a human, we also recommend running Manticore. Doing so can reveal errors with less manual effort. Our own Manticore efforts were limited by time and available CPUs; if we had more of either resource, we would have worked to expand Manticore’s coverage of the contracts. The Maker Foundation should consider running Manticore regularly and making it part of their CI process.

Throughout the audit, we developed several automated analyses to help discover potential bugs in the code. A high-level description of our approach is available below, and tool-specific information is available in the appendices:

- Appendix C documents our use of Slither for static analysis, particularly to estimate klab coverage.
- Appendix D documents our use of the Manticore symbolic executor to look for vulnerabilities involving multiple transactions.
- Appendix E documents our use of the Echidna property-testing framework to parametrize existing tests.

We delivered all code used in analysis along with this report to enable continuous analysis as development proceeds.

Automated Testing and Analysis

Much of the time spent on this engagement was focused on automated analysis. We began by using Slither to discover what properties were already verified by klab. We discovered that slightly more than two-thirds of functions had no klab coverage, and what properties existed were typically specific to a single function. Appendix C contains data produced by this analysis, technical details, and reproduction instructions.

These results were in line with our expectations. K specifications typically describe the behavior of a specific function, so any functions without an associated kspec will be uncovered. Function-specific coverage will not catch bugs that require multiple transactions to exploit or bugs in uncovered functions. The results also presented a clear direction for the development of our own analysis: By writing analyses that generalize over transaction sequences rather than focusing on specific functions, we could verify higher-level properties and achieve more comprehensive coverage.

Trail of Bits maintains two tools for reasoning about transaction sequences: Manticore, a symbolic executor, and Echidna, a property-based testing framework. In the course of this assessment, we employed both. We used Echidna to parametrize the existing unit tests and thus dramatically expand their coverage, and Manticore to symbolically execute short transaction sequences while detecting possible correctness issues.

The Echidna analysis is ultimately an iteration on existing testing practices. The DSS already has an impressive number of unit tests, most of which perform a specific transaction sequence and then assert an expected result. We modified these tests to use random values generated by Echidna. Properties that were previously checked after a single call sequence can now be checked after thousands of them, all randomly generated. Technical details and instructions to reproduce can be found in Appendix E.

While the Echidna analysis can be seen as an iteration on existing practices, the Manticore analysis is more novel. Manticore comes with an existing corpus of “bug detectors” that determine whether potential correctness issues occur in the execution of a call sequence. We augmented these with our own custom set of invariants, then used symbolic execution to check for their presence in short call sequences. While this was quite resource intensive, it ultimately led to TOB-MCD-011 and TOB-MCD-012. Technical details and instructions to reproduce can be found in Appendix D.

Project Dashboard

Application Summary

Name	Multi-Collateral Dai (MCD) Core Smart Contracts
Version	See Coverage below
Type	Solidity, Bash script
Platforms	EVM, POSIX

Engagement Summary

Dates	July 29 through August 30, 2019
Method	Whitebox
Consultants Engaged	4
Level of Effort	16 person-weeks

Vulnerability Summary

Total High-Severity Issues	0	
Total Medium-Severity Issues	2	■■
Total Low-Severity Issues	4	■■■■
Total Informational-Severity Issues	8	■■■■■■■■
Total	14	

Category Breakdown

Access Controls	4	■■■■
Cryptography	1	■
Data Validation	2	■■
Denial of Service	1	■
Documentation	1	■
Patching	2	■■
Timing	3	■■■
Total	14	

Engagement Goals

Trail of Bits and the Maker Foundation scoped the engagement to provide a security assessment of:

- Dai Stablecoin System (DSS) contracts (e.g., Cat, Jug, Vat)
- Governance contracts (e.g., DSChief, DSPause, DSToken)
- Price oracle infrastructure (e.g., median and oracle security module contracts)
- Adapters (GemJoin, ETHJoin, DaiJoin)
- Front-ends (e.g., DssCdpManager)
- Off-chain infrastructure (Omnia, Setzer, Scuttlebutt)
- Deployment scripts (e.g., base-deploy, deploy-core, auth-checker)

Specifically, Maker Foundation emphasized the following areas of focus:

- Code Correctness
    - Review higher invariants not expressible as a function-local property.
    - Fuzz the codebase for bad behavior induced by multiple, sequential function calls.
    - Carefully consider the failure modes of the authentication system, and ensure the system overall is resilient to individual key loss or theft.
    - Read documentation thoroughly, and ensure users can easily understand how to use the system safely and correctly.

- Verification System
    - Ensure the verification system is sound and applied correctly.
    - Review the set of verified properties for coverage.
    - Develop verification-aware custom tooling to identify blind spots.

- Runtime Environment Interaction
    - Review possible information-theoretical bugs (e.g., frontrunning attacks).
    - Review possible finality-related bugs (e.g., eclipse attacks).
    - Prepare for possible changes to the Ethereum runtime (e.g., changes to the gas cost model, changes to the consensus model, sharding).

- Availability
    - Ensure the system can never become “deadlocked” or unavailable.

Coverage

Trail of Bits manually reviewed and used Slither to analyze all of the following repositories:[1]

ds-auth	f783169408c278f85e26d77ba7b45823ed9503dd
ds-chief	ea05ee0413a8b3852142664a6c04d6e4923be426
ds-exec	c53aab4ba91645b30b0644972ef016b5ee606ca8
ds-guard	4678e1c74fce1542f1379f11325d7bfbbb897344
ds-math	784079b72c4d782b022b3e893a7c5659aa35971a
ds-note	beef8166f2184a4bac3d02abdb944647fd735060
ds-pause	81fd9d43e56615267a10e29710716342bcca0ce3
ds-proxy	379f5e2fc0a6ed5a7a96d3f211cc5ed8761baf00
ds-roles	01383725a4240000c0e274e55bdcf251570fd486
ds-spell	c908b7807f08661b4eca97adff6d9561d0116244
ds-stop	6e2bda69cb3cbf25a475491d9bc22969adb05993
ds-test	a4e40050b809705b15867939f5829540c50cb84f
ds-thing	5e49fcbdf4ef8ccd241423ed114576f51c42f1e0
ds-token	cee36a14685b3f93ffa0332853d3fcd943fe96a5
ds-value	d2107c1751f086aed3c38a2f433d6945444af7d6
ds-value	f3071713afbd583991637f8cfab5e0d29466dffd
ds-weth	dfada5bca7a00046c1ddc37c0c43106a8c0a4e5a
dss	7645fd00eedbad700a89d03e18dd2aa397c3d743
	526fa6afb9ea771f846b895ae0aee361876f2bdb
dss-add-ilk-spell	a43e3d47160d12ee9428ff3e0ebc5129de2caf96
dss-cdp-manager	11cace63e53d8e4fa64889701c290f741ae32330
	b0dfe6a02c876a08c8ff57c8561bd591d4c8320f
dss-deploy	4cf50f20e481a0f9026354dd45bd16bcb15e4501
	6100d63d6d1abbbfb5d57def8336b387a14b804e
dss-proxy-actions	928f13b8f384f096ac3128ac8729bfa6ff68de53
	696b9acd6040347781a5da97bc08c0890a49c9d3
erc20	f322aaca414db343337814097d2af43214bee96c
esm	e0a85d6215cec2a7786c1dcaee188a3ff393710d
gov-polling-generator	d08e43ee1a8d6daac3fc0bc4aee5a0c92f62c2e8
line-spell	bd40ebd89d28a2428a7500027b27c3888f369e01
	394ae373b59a2636a5f830f42986a468d492d70a
median	a5f39fab14f3b3bcd9576072da59984af8952606
multicall	b8771d9fe2b1429ae95cae622c4d880fd897562a
oracles-v2	fa6f5782b072d638f8f6b505c7d6dc726dd97e87
osm	504c47437916e29d918a9d1f40eb1f7595f3e9ce

[1] The appearance of two commit hashes means that the repository was upgraded midway through the assessment.

```
proxy-registry                1aa2ba356802a66f2de1f0ff78fabe1756b905a5
scd-mcd-migration             ebc09b3094ca1befeb08c799727d6a59a23a1427
setzer                        bc8100bbfd5b2b0b8495058019ef297c7d319e9b
testchain-medians             23524894915202452f5bb39e7c1d4375a4482c4f
testchain-pause-proxy-actions b33c2d9c8354b294e3d4e7d7da8f60a63780790f
                              8ab93d145b11101138447af3888420ee5753e2cd
token-faucet                  d7349d13f6cd83e8d0aa21e93544988fab0b6b24
vote-proxy                    6fdbee3ac48bb915e715668374c1deba95cdb6f6
```

Manual review resulted in TOB-MCD-001, TOB-MCD-005, TOB-MCD-007, TOB-MCD-008, TOB-MCD-009, TOB-MCD-010, TOB-MCD-013, and TOB-MCD-014, as well as entries in Appendix B.

Slither identified TOB-MCD-002, TOB-MCD-006, and some entries in Appendix B. Our Slither extensions determined which contracts were and were not covered by klab specifications, leading to TOB-MCD-003. In addition:

- The DSS and governance contracts were deployed and the contents of their `wards` (i.e., `rely`/`deny`) mappings were extracted. We then checked the results against what was verified by the `auth-checker` script. This effort led to TOB-MCD-004. We carried out a similar analysis on the Vat's `can` (i.e., `hope`/`nope`) mapping.

- The DSS contracts and DssCdpManager were subject to multiple symbolic transactions using Manticore. Manticore includes a standard battery of tests for "bad behavior" (e.g., reads from uninitialized storage, integer overflows, etc.). In addition, we wrote MCD-specific property tests. These efforts resulted in TOB-MCD-011 and TOB-MCD-012, as well as entries in Appendix B.

- The deployment scripts had errors manually introduced into them, and we verified that those errors were caught and reported. This effort did not result in any findings, but did increase our confidence in the correctness of the deployment scripts.

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

❑ **Document that auctions are susceptible to transaction-reordering attacks** (TOB-MCD-001). This will alert users to the risk while alternative solutions are explored.

❑ **Use neither `ABIEncoderV2` nor any other experimental Solidity features** (TOB-MCD-002). Refactor the code to avoid passing or returning arrays of strings to and from functions. `ABIEncoderV2` has been the source of numerous bugs, and its use presents unnecessary risk.

❑ **Update the `k-dss` repository to link against the current `master` branch of each system component** (TOB-MCD-003). Alternatively, document the version of each MCD module that is verified. This will help to ensure that the code verified by klab is also the code that is deployed.

❑ **Adjust the `auth-checker` script so that it verifies the presence of the three edges mentioned in** TOB-MCD-004. This will help to ensure that the code functions properly once deployed.

❑ **Consider ways that MCD's numerous permissions mechanisms might be consolidated** (TOB-MCD-005). Consolidating the permissions mechanisms would make it easier to verify that they have been configured correctly. This would, in turn, make it less likely for bugs to arise in the future.

❑ **Remove the check that allows `transferFrom` to succeed without approval if `src` refers to `msg.sender`** (TOB-MCD-006). Update all code which depended on `transferFrom` not expecting approval in this case.

❑ **Don't require "locking" to earn interest on Dai** (TOB-MCD-007). Instead of keeping Dai Savings Rate logic in `pot.sol`, apply it to the ERC20 balance whether it's locked or not. Alternatively, implement liquidity controls such as time-restricted withdrawals so "locking" is more meaningful.

❑ **Use two non-ERC20 functions, allowing a user to increase and decrease the approval, to work around the known ERC20 race condition** (TOB-MCD-008). Ensure

users are aware of this extra functionality, and encourage them to make use of it when appropriate.

❑ **Implement mechanisms to watch for abuses of the `permit` method, as described in** TOB-MCD-009. If such abuses are detected, the Maker Foundation can take steps to alert its users, and possibly accelerate development of an alternative solution.

❑ **Modify `permit` to explicitly disallow having a `holder` of 0** (TOB-MCD-010). There seems to be no legitimate reason for this address to hold Dai.

❑ **Have `file` methods revert when the `what` argument is unrecognized, like in Figure 2 of** TOB-MCD-011. This will help to identify situations where an incorrect `what` argument is used in a `file` call (e.g., because the argument was misspelled).

❑ **Add a `what` argument to the Spotter `file` method that lacks such an argument** (TOB-MCD-012). This will help to distinguish it from the other `file` method. The similarity of their present signatures could lead to errors.

❑ **Update the Medium post and whitepaper to clarify the function of the DSR** (TOB-MCD-013). This will help to prevent misunderstandings concerning how changes in the DSR affect the Dai supply.

❑ **Eliminate the requirement in `vow.flop` that `vat.dai(address(this)) == 0`** (TOB-MCD-014). This will eliminate a potential Denial of Service attack,

Long Term

❑ **Investigate using a Dutch or sealed-bid auction for each type of auction** (TOB-MCD-001). Use of either one could eliminate the possibility of a transaction-reordering attack.

❑ **Integrate static analysis tools like Slither into your CI pipeline to detect unsafe pragmas** (TOB-MCD-002). This will identify features in the Solidity compiler that might otherwise be assumed safe.

❑ **Consider whether using one instance of a DSGuard contract could meet your permissions needs** (TOB-MCD-005). A simpler permissions model would be easier to reason about, and would therefore be less error-prone.

❑ **Review the ERC20 specification and verify your contracts meet the standard** (TOB-MCD-006). When interfacing with external ERC20 tokens, be wary of popular tokens that do not properly implement the standard (e.g., many tokens do not include return values for `approve`, `transfer`, `transferFrom`, etc.).

❑ **Avoid situations where using Dai through a third-party smart contract is preferable to using Dai through a MakerDAO smart contract** (TOB-MCD-007). This will dramatically reduce security risks in untrusted, third-party code and prevent many scams that impersonate legitimate Dai tooling.

❑ **Maintain one nonce per Dai holder-spender pair, and add a second type of signed message** (analogous to `permit`) to set a spender's nonce to the max of the nonce's current value and a constant (TOB-MCD-009).

❑ **Audit uses of `ecrecover` for authentication issues** (TOB-MCD-010). Ensure that its failure case is explicitly handled. Consider replacing it with an ECDSA library.

❑ **Incorporate fuzzing or symbolic execution into your CI, and regularly review the results**. We found TOB-MCD-011 using Manticore, which produced successful calls to `file` methods with garbage `what` arguments.

❑ **Ensure consistency among function signatures as new functions are introduced to the code base** (TOB-MCD-012). Such checks will help to prevent future situations where two functions have overly similar signatures.

❑ **Regularly review all public documentation for accuracy as Dai functionality is updated** (TOB-MCD-013). This will help to catch errors within such documentation, and promote understanding within the community on how MCD functions.

❑ **Investigate whether `heal` should be called upon entry to `flop`** (TOB-MCD-014). If such a call does introduce additional risks, then it presents an alternative means to preventing a potential Denial of Service attack.

Findings Summary

#	Title	Type	Severity
1	Auctions are susceptible to transaction-reordering attacks	Timing	Low
2	ABIEncoderV2 is not production-ready	Patching	Informational
3	`k-dss` is out of sync with other repositories	Patching	Informational
4	auth-checker's use of checkRely is incomplete	Access Controls	Informational
5	Too many notions of "permission"	Access Controls	Informational
6	ERC20 transferFrom often does not follow spec	Access Controls	Informational
7	Dai Savings Rate locking is ineffective	Access Controls	Medium
8	Race condition in the ERC20 "approve" function may lead to token theft	Timing	Informational
9	Race condition involving Dai "permit" nonces	Timing	Informational
10	Anyone can approve themselves to take Dai owned by address 0	Cryptography	Low
11	"file" methods do not revert when "what" argument is unrecognized	Data Validation	Low
12	Spotter's "file" method lacks a "what" argument	Data Validation	Low
13	Documentation of Dai Savings Rate is inaccurate	Documentation	Informational

14	A Denial of Service attack can obstruct Flop auctions	Denial of Service	Medium

1. Auctions are susceptible to transaction-reordering attacks

Severity: Low
Type: Timing
Target: Flip, Flap, and Flop

Difficulty: High
Finding ID: TOB-MCD-001

Description
MCD features three types of auctions. Each is susceptible to a transaction-reordering attack, where a miner replaces a legitimate bid with their own bid:

- A Flip auction occurs when a Collateral Debt Position (CDP) loses too much of its value relative to the Dai borrowed against it. The collateral is auctioned off for Dai.
- A Flap auction occurs to dispense Dai collected as stability fees. The Dai is auctioned off for MKR. The MKR corresponding to the winning bid is burned.
- A Flop auction occurs when bad debt must be covered. A descending amount of MKR is auctioned off for a fixed amount of Dai (i.e., the MKR bids are given in descending order). An amount of MKR corresponding to the winning bid is minted.

Each of these auctions features a mandatory bid increase of 5%. (For Flop auctions, increasing the new bid by 5% must still make that bid less than the preceding bid of MKR.) If an auction participant is the victim of a transaction-reordering attack, this mandatory bid increase penalizes them. The victim must choose to either give up on bidding or incur a 5% penalty on their next bid.

Also, Flap auctions are susceptible to a distinct type of transaction-reordering attack. If a miner observes a large bid in a Flap auction, the miner knows that a large amount of MKR is about to go out of circulation, which will lead to an MKR price increase. The miner could submit a buy order on an exchange for MKR prior to mining the bid, which would enable the miner to obtain MKR at an unfairly low price.

Exploit Scenario
Eve is both a miner and a holder of Dai. Eve currently holds the highest bid on some CDP. Eve notices that Bob has submitted a bid of X Dai, more than her own bid. Eve mines (i.e., wins the race for) the next block with her own bid of X Dai ahead of Bob's. Bob loses the cost of his gas and must choose to either stop bidding or bid ≥ 1.05 * X Dai.

Recommendation
Short term, document that auctions are susceptible to transaction-reordering attacks.

Long term, investigate using a Dutch[2] or sealed-bid auction (the latter via a commit-reveal scheme) for each type of auction.

[2] Investopedia: Dutch Auction: "A Dutch auction also refers to a type of auction in which the price on an item is lowered until it gets a bid. The first bid made is the winning bid and results in a sale, assuming that the price is above the reserve price."

2. ABIEncoderV2 is not production-ready

Severity: Informational
Type: Patching
Target: `cat.sol`, `end.sol`, `jug.sol`

Difficulty: High
Finding ID: TOB-MCD-002

Description
The contracts use the new Solidity ABI encoder, `ABIEncoderV2`. This encoder is still experimental and is not ready for production use.

More than three percent of all GitHub issues for the Solidity compiler are related to experimental features, with `ABIEncoderV2` constituting the vast majority of them. Several issues and bug reports are still open and unresolved. More than 20 high-severity bugs over the past year have been associated with `ABIEncoderV2`, and some are so recent they have not yet been included in a Solidity release.

For example, earlier this year a severe bug was found in the encoder and was introduced in Solidity 0.5.5.

Exploit Scenario
The MakerDAO contracts are deployed. After the deployment, a bug is found in the encoder. As a result, the contracts are broken and can be exploited, perhaps to incorrectly value a CDP.

Recommendation
Short term, do not use either `ABIEncoderV2` or any other experimental Solidity features. Refactor the code to avoid passing or returning arrays of strings to and from functions.

Long term, integrate static analysis tools like Slither into your CI pipeline to detect unsafe pragmas.

3. k-dss is out of sync with other repositories

Severity: Informational
Type: Patching
Target: `dss-deploy-scripts`

Difficulty: Low
Finding ID: TOB-MCD-003

Description
`k-dss` provides klab specifications for an unspecified version of MCD. It is out of sync with the `master` branches of the repositories it verifies.

The klab specification for `Flapper.kick` provided by the `k-dss` repo references a new version of the `dss`, where `flap.sol` took input parameters that were different from the version used by MCD deployment scripts:

- Verified in `k-dss`: 057fdfa5e974dca4dee5f9238f61a0f0ce2aa9c4
- Version in `dss-deploy-scripts`: 880d592091d5582adc2fda0bdb56c76e3b7457c3

The provided specification for `Flapper.kick` references a function prototype:

```
interface kick(uint256 lot, uint256 bid)
```

Figure 1: klab specification for `Flapper.kick` *(*`dss.md`*).*

But the version of `flap.sol` provided by `dss-deploy-scripts` only provides the function:

```
function kick(address gal, uint lot, uint bid)
```

Figure 2: `Flapper.kick` *declaration provided by* `dss-deploy-scripts` (`flap.sol`*).*

Exploit Scenario
A developer/tester deploys MCD to their desired network using `dss-deploy-scripts`, assuming the core contracts are verified by the kspecs that `k-dss` provides. However, the kspecs provided by `k-dss` are out of sync with the contracts deployed by `dss-deploy-scripts`. This does not become apparent until after deployment, leaving the user questioning the validity of the code deployed.

Recommendation
Update the `k-dss` repository to link against the current `master` branch of each system component. Alternatively, document the version of each MCD module that is verified.

4. auth-checker's use of checkRely is incomplete

Severity: Informational
Type: Access Controls
Target: `scripts/auth-checker`

Difficulty: High
Finding ID: TOB-MCD-004

Description
The `auth-checker` script checks permissions settings within a deployment. Specifically, the script's `checkRely` method checks the presence or absence of an edge within a contract's `wards` mapping. The problem: There are edges that must be present that are not checked. If the edges are missing, all privileged access to a contract could be revoked, and that contract would be locked from future privileged actions.

Applying `checkRely` to all pairs of variables within `out/addresses.json` results in the graph in Figure 1 on the next page. There are three edges in this graph that are not checked by the `auth-checker` script. They are:

```
    DEPLOYER → FAUCET
MCD_JOIN_DAI → MCD_DAI
     MCD_POT → MCD_VAT
```

We presume the first edge is needed only for testing. However, the DaiJoin contract needs "`DEPLOYER → FAUCET`" to call the Dai contract's `mint` and `burn` methods. Similarly, the Pot contract needs "`MCD_POT → MCD_VAT`" to call the Vat contract's `suck` method. Therefore, the `auth-checker` script should verify that these edges are present.

Exploit Scenario
A change is made to the deployment process, causing the Pot contract to lose privileged access to the Vat contract. The `auth-checker` script misses this failure, and the Pot contract is unable to mint Dai. Confidence in the system is lost.

Recommendation
Short term, adjust the `auth-checker` script so that it verifies the presence of the three edges mentioned above.

Long term, consider ways that MCD's numerous permissions mechanisms might be consolidated. (See TOB-MCD-005.) Consolidating the permissions mechanisms would make it easier to verify that they have been configured correctly. This would, in turn, make it less likely for similar bugs to arise in the future.

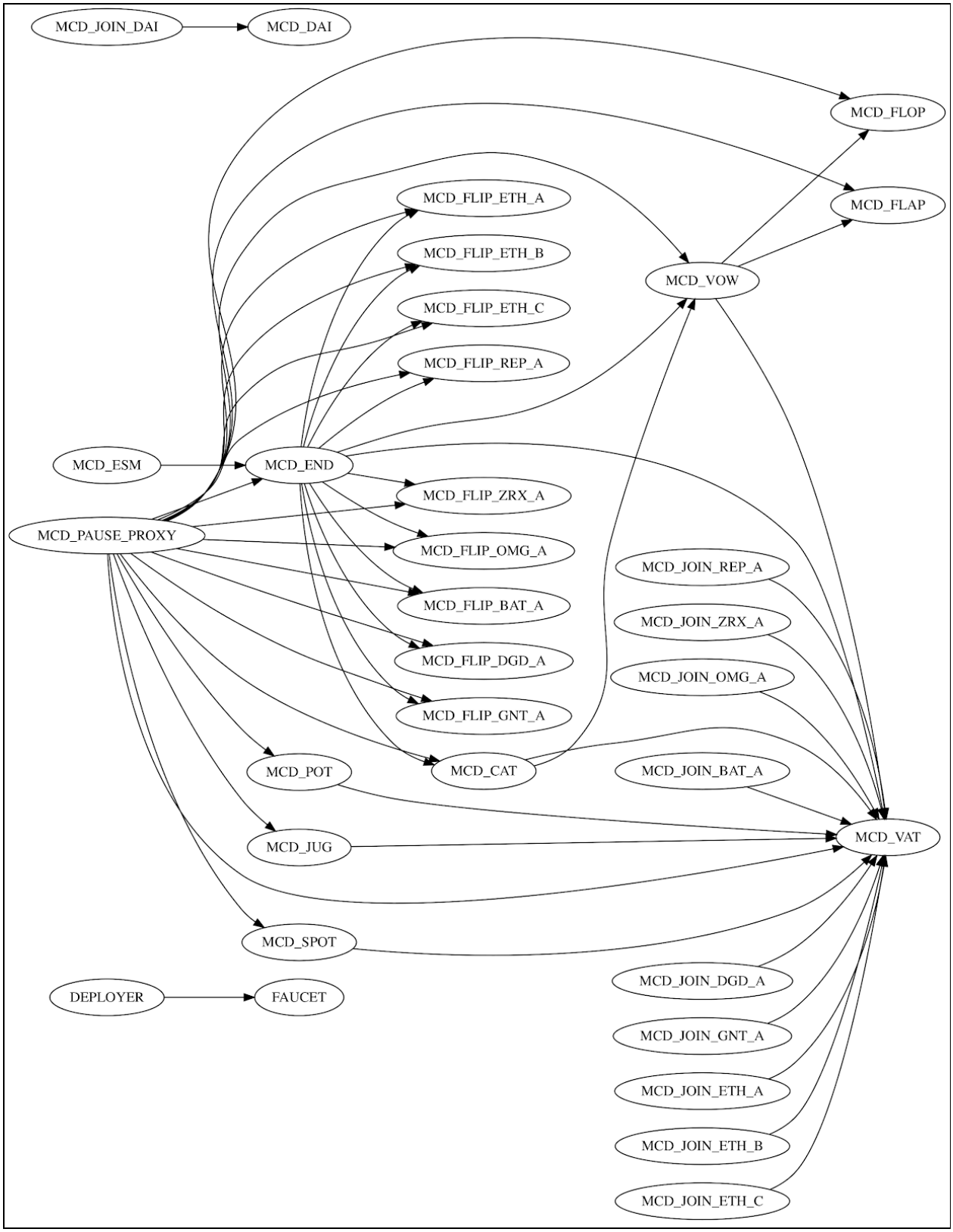

Figure 1: The result of applying `checkRely` *to all pairs of variables within* `out/addresses.json`*. An edge from X to Y indicates that* `checkRely(X,Y)` *results in* `AUTHED (0x00...01)`*.*

5. Too many notions of “permission”

Severity: Informational | Difficulty: High
Type: Access Controls | Finding ID: TOB-MCD-005
Target: DSRoles, DSAuth, DSAuthority, DSToken, `dss/src/*.sol`

Description
Numerous mechanisms are used to enforce permissions within MCD:

- Within any instance of a DSRoles contract, four distinct mappings enforce permissions. Those mappings involve users that have roles, and these roles have capabilities.
- Within any token, an inherited `isAuthorized` method is used to enforce permissions. The `isAuthorized` method, in turn, uses a DSAuthority contract to enforce permissions via its `canCall` method.
- Within the Cat, Dai, End, Flap, Flip, Flop, Jug, Pot, Spot, Vat, and Vow contracts, a `wards` mapping enforces permissions.
- Within the Vat contract, a `can`/`wish` mapping is used in addition to the `wards` mapping to enforce permissions.

It appears that many (if not all) of these permissions mechanisms could be implemented using one instance of a DSGuard contract. However, it also appears the DSGuard contract is used only for testing.

Exploit Scenario
A developer, Alice, adds a function to a contract that requires permission. In granting permission to contract X to call the function, Alice inadvertently grants permission to contract Y to call the function. Alice’s error is the result of confusion over MCD’s many permissions mechanisms. Eve exploits the flaw for financial gain.

Recommendation
Short term, consider which of the above permissions mechanisms might be consolidated. To be clear, we’re not suggesting that you change any of the permissions themselves, just how they are implemented.

Long term, consider whether using one instance of a DSGuard contract could meet your permissions needs.

6. ERC20 transferFrom often does not follow spec

Severity: Informational
Type: Access Controls
Target: `tokens.sol, dai.sol, base.sol, token.sol`

Difficulty: Low
Finding ID: TOB-MCD-006

Description
If the message sender is the source of a `transferFrom` call, approval will not be considered, and the transfer will initiate immediately. This breaks invariants expected of `transferFrom`.

Traditionally, the `transferFrom` method moves tokens from one account to another, provided the source account has approved the sender to send such an amount using the ERC20 method `approve`. However, some ERC20 tokens in the MCD do not require approval if the sender is the source of the account:

```
if (src != msg.sender) {
    require(_approvals[src][msg.sender] >= wad,
        "ds-token-insufficient-approval");
    _approvals[src][msg.sender] = sub(_approvals[src][msg.sender], wad);
}
```

Figure 1: `transferFrom` *allowance checks (dss-deploy/src/tokens.sol#L43-L46).*

The following contracts harbor this problem:

- dss/src/dai.sol#L74-L77
- dss-deploy/src/tokens.sol#L43-L46
- ds-token/src/base.sol#L51-L54
- ds-token/src/token.sol#L49-L52

Although it may seem intuitive to allow the owner of the account balance to transfer funds without approval, external tooling may rely on invariants which are now broken.

Exploit Scenario
Alice sends a transaction which invokes `transferFrom`, assuming it will fail if no allowance/approval was set beforehand. Instead, the transfer succeeds if the source of the funds is also `msg.sender`. Alice's funds are lost.

Recommendation
Short term, remove the check that allows transferFrom to succeed without approval if `src` refers to `msg.sender`. Update all code which depended on `transferFrom` not expecting approval in this case.

Long term, review the ERC20 specification and verify your contracts meet the standard. When interfacing with external ERC20 tokens, be wary of popular tokens that do not properly implement the standard (e.g., many tokens do not include return values for `approve`, `transfer`, `transferFrom`, etc.).

References

- ERC20 Token Standard
- Missing return value bug—at least 130 tokens affected
- Explaining unexpected reverts starting with Solidity 0.4.22

7. Dai Savings Rate locking is ineffective

Severity: Medium
Type: Access Controls
Target: `pot.sol`

Difficulty: Medium
Finding ID: TOB-MCD-007

Description
The Dai Savings Rate (DSR) is intended to allow people to lock Dai and thereby earn interest. However, Dai earns interest only when it is locked in the Pot contract. This encourages the use of nonstandard ERC20 proxies for Dai. The proxy token keeps the underlying asset locked and earning interest, but still available for trading. These proxies are not under the control of MakerDAO and can pose a threat to the ecosystem.

Exploit Scenario
Bob holds Dai and wants to earn interest without sacrificing liquidity. He deposits his Dai with a popular contract written by Alice. The contract locks it and gives him a new ERC20 token, "ADai," which he can later redeem for his original Dai, plus interest. However, this contract is buggy, and he actually loses all of his holdings to a hacker. He vows never to use Dai again.

Recommendation
Don't require "locking" to earn interest on Dai. Instead of keeping Dai Savings Rate logic in `pot.sol`, apply it to the ERC20 balance whether it's locked or not. Alternatively, implement liquidity controls such as time-restricted withdrawals so "locking" is more meaningful.

Going forward, avoid any situation where using Dai through a third-party smart contract is preferable to using Dai through a MakerDAO smart contract. This will dramatically reduce security risks in untrusted, third-party code and prevent many scams that impersonate legitimate Dai tooling.

8. Race condition in the ERC20 “approve” function may lead to token theft

Severity: Informational
Type: Timing
Target: `dai.sol, base.sol, token.sol`

Difficulty: High
Finding ID: TOB-MCD-008

Description
A known race condition in the ERC20 standard, on the `approve` function, could lead to token theft.

The ERC20 standard describes how to create generic token contracts. Among others, an ERC20 contract defines these two functions:

- `transferFrom(from, to, value)`
- `approve(spender, value)`

These functions give permission to a third party to spend tokens. Once the function `approve(spender, value)` has been called by a user, `spender` can spend up to the `value` of the user’s tokens by calling `transferFrom(user, to, value).`

This schema is vulnerable to a race condition, where the user calls `approve` a second time on a `spender` that has already been allowed. If the spender sees the transaction containing the call before it has been mined, the spender can call `transferFrom` to transfer the previous value and still receive the authorization to transfer the new value.

Exploit Scenario
1. Alice calls `approve(Bob, 1000)`. This allows Bob to spend 1,000 tokens.
2. Alice changes her mind and calls `approve(Bob, 500)`. Once mined, this will decrease to 500 the number of tokens that Bob can spend.
3. Bob sees the second transaction and calls `transferFrom(Alice, X, 1000)` before `approve(Bob, 500)` has been mined.
4. If Bob’s transaction is mined before Alice’s, Bob will transfer 1,000 tokens. But once Alice’s transaction is mined, Bob can call `transferFrom(Alice, X, 500)`. Bob has transferred 1,500 tokens even though this was not Alice’s intention.

Recommendation
One common workaround is to use two non-ERC20 functions, allowing a user to increase and decrease the approval (see `increaseApproval` and `decreaseApproval` of StandardToken.sol#L63-L98). Ensure users are aware of this extra functionality and encourage them to make use of it when appropriate.

9. Race condition involving Dai “permit” nonces

Severity: Informational
Type: Timing
Target: `dai.sol`

Difficulty: High
Finding ID: TOB-MCD-009

Description
A Dai `permit` call is essentially a restricted form of an `approve`[3] call that can be signed offline and submitted by a third party. Each signed `permit` call features a nonce. The nonces must be used in strictly increasing order (i.e., Alice’s nonce n can only be used once nonces 0 through n-1 have been used). However, such a mechanism is susceptible to attack.

In the following discussion, we assume Alice does not have access to the blockchain; otherwise, she could call `approve` directly.

Suppose Eve holds a `permit` call signed by Alice with nonce n, but Eve has not yet submitted the call to the blockchain. By holding onto the call, Eve prevents `permit` calls signed by Alice with higher nonces from being processed.

Alice’s only means of recourse appears to be to sign a new permit call P’, also with nonce n, and to ask Bob to submit P’ on her behalf. However, upon seeing P’ submitted to the blockchain, Eve can then submit P, causing P’ to appear invalid (because it reuses the nonce used by P). Alice would then have to sign a third permit call P” to undo the effects of P and/or redo the effects of P’. She would have to again ask Bob to submit P” on her behalf.

Exploit Scenario
Eve holds onto a `permit` call signed by Alice, authorizing Eve to spend on Alice’s behalf. By holding onto the call, Eve prevents `permit` calls signed by Alice with higher nonces from being processed. Eventually Alice gets tired of waiting and signs a new `permit` call revoking Eve’s spending privileges. But by then, Alice’s funds have already been drained.

Recommendation
Short term, implement mechanisms to watch for abuses of the `permit` method, as described above.

Long term:

1. Rather than maintain one nonce per Dai holder, maintain one nonce per Dai holder-spender pair, and
2. Add a second type of signed message (analogous to `permit`) that allows a Dai holder to set a spender’s nonce to the max of the nonce’s current value and a constant (e.g., like in Figure 1).

[3] The `permit` method can set a spender’s allowance to the minimum or maximum allowable, but nothing in between. In this way, `permit` does not offer the full generality of `approve`.

Such messages would allow a Dai holder to invalidate a nonce without having to worry whether the corresponding `permit` message had already been submitted.

```
function invalidate_nonces(address holder, address spender, uint256 new_nonce,
                           uint8 v, bytes32 r, bytes32 s) public
{
    ...
    uint256 nonce = nonces[holder][spender];
    nonces[holder][spender] = max(nonce, new_nonce);
    ...
}
```

Figure 1: Hypothetical "`invalidate_nonces`" implementation.

10. Anyone can approve themselves to take Dai owned by address 0

Severity: Low
Type: Cryptography
Target: `dai.sol`

Difficulty: Low
Finding ID: TOB-MCD-010

Description
The `permit` method in Dai.sol uses `ecrecover` to check the signer of a pre-signed approval message. `ecrecover` returns 0 on an invalid signature rather than reverting. If `permit` is called with an invalid signature and a `holder` of 0, it will execute as if the signature is valid. This means any tokens sent to the address 0 (e.g., for burning) can be claimed by anyone who first calls `permit`, then `transferFrom`.

Exploit Scenario
An ICO uses 0 as a proof-of-burn address for buying tokens with Dai. The tokens they expect to be burned are actually stolen by an enterprising hacker who abuses `permit`.

Recommendation
In the short term, modify `permit` to explicitly disallow having a `holder` of 0.

Long-term, carefully audit all uses of `ecrecover` for authentication. Ensure that its failure case is explicitly handled. Consider replacing it with an ECDSA library.

References
- Solidity Documentation
- Dai transfers to 0

11. “file” methods do not revert when “what” argument is unrecognized

Severity: Low
Type: Data Validation
Target: DSS contracts

Difficulty: High
Finding ID: TOB-MCD-011

Description
Many of the DSS contracts feature one or more `file` methods for setting contract parameters. The contract parameter name when present (see TOB-MCD-012) is called `what`. None of the `file` methods revert when the `what` argument is unrecognized. So, setting a non-existent parameter appears to succeed.

As an example, End’s `file` implementation appears in Figure 1. A better implementation, one that reverts when the `what` argument is unrecognized, appears in Figure 2.

```
function file(bytes32 what, address data) external note auth {
    if (what == "vat")  vat = VatLike(data);
    if (what == "cat")  cat = CatLike(data);
    if (what == "vow")  vow = VowLike(data);
    if (what == "spot") spot = Spotty(data);
}
```

Figure 1: End’s actual file implementation (dss/src/end.sol#L242-L247).

```
function file(bytes32 what, address data) external note auth {
    if (what == "vat")  vat = VatLike(data);
    else if (what == "cat")  cat = CatLike(data);
    else if (what == "vow")  vow = VowLike(data);
    else if (what == "spot") spot = Spotty(data);
    else revert();
}
```

Figure 2: A better End file implementation.

Exploit Scenario
An MCD administrator wishes to change an MCD parameter using a `file` method, but the administrator misspells the parameter name. The call appears to succeed, leading the administrator to believe the parameter has been changed when it has not.

Recommendation
Short term, have `file` methods revert when the `what` argument is unrecognized, as shown in Figure 2.

Long-term, incorporate fuzzing or symbolic execution into your CI, and regularly review the results. We found this solution when using Manticore, which produced successful calls to `file` methods with garbage `what` arguments.

12. Spotter's "file" method lacks a "what" argument

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

Difficulty: High
Finding ID: TOB-MCD-012

Description
Many of the DSS contracts feature one or more `file` methods for setting contract parameters. Nearly all of those methods feature a `what` argument that contains the parameter name. However, one of Spotter's `file` methods lacks such an argument. Consequently, the method's signature is very similar to that of another `file` method, creating the potential for confusion.

The Spotter `file` method that lacks a `what` argument appears in Figure 1. The Spotter `file` method with a similar signature appears in Figure 2.

```
function file(bytes32 ilk, address pip_) external note auth {
    ilks[ilk].pip = PipLike(pip_);
}
```

Figure 1: Spotter file method that lacks a `what` *argument (dss/src/spot.sol#L71-L73).*

```
function file(bytes32 what, uint data) external note auth {
    if (what == "par") par = data;
}
```

Figure 2: Spotter file method with a signature similar to Figure 1 (dss/src/spot.sol#L74-L76).

Exploit Scenario
An MCD administrator wishes to change the value of Spotter's `par` parameter. However, the new value is encoded as an address instead of a uint, due to a bug in the administrator's client software. As a result, the administrator instead sets the `pip` value for a non-existent ilk called `par`.

Recommendation
Short term, add a `what` argument to the `file` method in Figure 1.

Long-term, try to ensure consistency among function signatures as new functions are introduced to the code base.

13. Documentation of Dai Savings Rate is inaccurate

Severity: Informational
Type: Documentation
Target: Whitepaper

Difficulty: N/A
Finding ID: TOB-MCD-013

Description
The Dai Savings Rate (DSR) does not constrict Dai supply. However, both the whitepaper and the MakerDAO Medium post (which currently appear first in an online search for "Dai Savings Rate") state that it does. This could lead Dai users to misunderstand the DSR's function.

Recommendation
Update both the Medium post and the whitepaper to clarify the function of the DSR.

Regularly review all public documentation for accuracy as Dai functionality is updated.

14. A Denial of ervice attack can obstruct Flop auctions

Severity: Medium
Type: Denial of Service
Target: `flop.sol, vat.sol, vow.sol`

Difficulty: Low
Finding ID: TOB-MCD-013

Description
In order to initiate a Flop auction, the Vow contract requires that it have a zero Dai balance within the Vat. An unprivileged user can send a small amount of Dai to the Vow within the Vat. In doing so, the user prevents the Vow from initiating a Flop auction until it calls `heal`.

The code for Vow's `flop` method appears in Figure 1. The crucial bit is the line that requires `vat.dai(address(this)) == 0`. The code for the Vat's `move` method appears in Figure 2. Note that anyone can call the `move` method. Thus, an unprivileged user can cause the required condition to fail simply by calling `move`.

```
function flop() external note returns (uint id) {
    require(sump <= sub(sub(vat.sin(address(this)), Sin), Ash));
    require(vat.dai(address(this)) == 0);
    Ash = add(Ash, sump);
    id = flopper.kick(address(this), uint(-1), sump);
}
```

Figure 1: Vow's `flop` *method (dss/src/vow.sol#L119-L124).*

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

Figure 2: The Vat's `move` *method (dss/src/vat.sol#L144-L148).*

Exploit Scenario
Eve runs a stable coin that competes with Dai. Whenever Eve sees a call to `vow.flop` posted to the blockchain, Eve posts a call of the form `vat.move(…, vow, …)` with a high gas price and a trivial amount of Dai. The high gas price causes Eve's transactions to be mined before those involving `vow.flop`. By staving off calls to `vow.flop` for extended periods of time, Eve causes bad debt to remain uncovered, and confidence in Dai declines.

Recommendation
Short term, eliminate the requirement that `vat.dai(address(this)) == 0`.

Long term, investigate whether `heal` should be called upon entry to `flop`.

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
Timing	Related to race conditions, locking, or order of operations
Undefined Behavior	Related to undefined behavior triggered by the program

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 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. Non-security related findings

This appendix contains findings that do not have immediate security implications.

- **`load-addresses` seems to try to do something that is impossible.** The `dss-deploy-scripts` README states, "The load-addresses script reads contract addresses from `out/addresses.json` and exports them as environment variables." However, in bash at least, this is not possible. (See Can I export a variable to the environment from a bash script without sourcing it?)

- **Many functions do not follow the check-effects-interactions pattern.** Slither reports several functions as being reentrant (e.g., `ESM.fire()` in Figure 1). A reentrant function is not vulnerable if there is a guarantee that any contract called by the function is trusted (e.g., `end` in Figure 1 is trusted). However, a better approach is to follow the check-effects-interactions pattern and avoid the problem altogether.

```
function fire() external note {
    require(!fired,  "esm/already-fired");
    require(full(),  "esm/min-not-reached");

    end.cage();

    fired = true;
}
```

Figure 1: `ESM.fire()` *(esm/src/ESM.sol#L57-L64).*

- **Significant code duplication exists among the contracts.** For example, each of the Cat, Dai, End, Flap, Flip, Flop, Jug, Pot, Spot, Vat, and Vow contracts feature the code in Figure 2. In another extreme example, a 23-line `rpow` function appears essentially unchanged in both the Jug and Pot contracts.

```
mapping (address => uint) public wards;
function rely(address usr) public note auth { wards[usr] = 1; }
function deny(address usr) public note auth { wards[usr] = 0; }
modifier auth { require(wards[msg.sender] == 1); _; }
```

Figure 2: Code duplicated across the contracts in `dss/src/*.sol`.

- **The field names of `Flipper.Bid` and `Flippy.Bid` do not match.** The definitions of `Flipper.Bid` and `Flippy.Bid` are given in Figures 3 and 4, respectively. The definitions differ in their names for the sixth field, `usr` vs. `urn`.

```
struct Bid {
    uint256 bid;
    uint256 lot;
    address guy;  // high bidder
    uint48  tic;  // expiry time
    uint48  end;
    address usr;
    address gal;
    uint256 tab;
}
```

Figure 3: `Flipper.Bid` *(dss/src/flip.sol#L49-L58).*

```
struct Bid {
    uint256 bid;
    uint256 lot;
    address guy;
    uint48  tic;
    uint48  end;
    address urn;
    address gal;
    uint256 tab;
}
```

`Figure 4: Flippy.Bid` *(dss/src/end.sol#L61-L70).*

- **`WETH9_` does not check for overflows (e.g., does not use SafeMath).** For example, `WETH9_.deposit()` appears in Figure 5. The lack of overflow checks could, say, cause Ether or tokens to be lost. This code appears to be used only for testing. Take care to ensure it is not used elsewhere.

```
function deposit() public payable {
    balanceOf[msg.sender] += msg.value;
    emit Deposit(msg.sender, msg.value);
}
```

Figure 5: `WETH9_.deposit()` *(ds-weth/src/weth9.sol#L34-L37).*

- **`RestrictedTokenFaucet` does not check that ERC20 transfers succeed.** For example, `RestrictedTokenFaucet.gulp(...)` appears in Figure 6. Failing to check that a transfer succeeds could make it impossible for an Ethereum address to draw from the faucet.

```
function gulp(address gem) external  {
    require(list[address(0)] == 1 || list[msg.sender] == 1,
        "token-faucet/no-whitelist");
    require(!done[msg.sender][gem],
        "token-faucet/already-used_faucet");
    require(ERC20Like(gem).balanceOf(address(this)) >= amt,
        "token-faucet/not-enough-balance");
    done[msg.sender][gem] = true;
    ERC20Like(gem).transfer(msg.sender, amt);
}
```

Figure 6: `RestrictedTokenFaucet.gulp(...)` *(token-faucet/src/RestrictedTokenFaucet.sol#L35-L41).*

- **The code that initiates Flop auctions is fragile.** The call in the Vow contract that initiates a Flop auction is:

  ```
  id = flopper.kick(address(this), uint(-1), sump);
  ```

  If no bid is placed in the auction, the Flop contract will attempt to mint `uint(-1)` MKR for the Vow contract (i.e., `address(this)` in the above call). Such an attempt will fail, as the MKR contract checks for overflows. However, this seems like a fragile way of avoiding such undesirable behavior.

- **`file` and `join` in the Pot contract should call `drip`.** To ensure the most accurate calculation of `chi`, calling the Pot contract's `file` method should invoke `drip` before updating `dsr`. Otherwise, the update can be effectively retroactive. Similarly, `drip` should be called on `join` to make sure `pie` is calculated with maximum accuracy.

- **Updating the Stability Fee should call `drip`.** Similarly, Jug's `file` method should invoke `drip` before updating `duty`.

- **Requiring that oracle addresses have distinct upper 8 bits may be overly restrictive.** The median contract currently requires that addresses of all oracles have distinct upper 8 bits (see Figure 7). However, it is conceivable that the addresses of two oracles could have the same upper 8 bits. Thus, this condition may be overly restrictive.

```
uint8 slot = uint8(uint256(signer) >> 152);
require((bloom >> slot) % 2 == 0, "Oracle already signed");
bloom += uint256(2) ** slot;
```

Figure 7: median.sol#L85-L87.

- **The toll modifier in `median.sol` and `osm.sol` is unlikely to be effective.** The files `median.sol` and `osm.sol` feature a `toll` modifier that tries to restrict certain view functions only to whitelisted contracts. However, since the protected data resides on a public blockchain, a determined adversary could extract the data and present it to their own contracts.

- **The DssCdpManager can write to the 0th CDP's next field.** Multiple, doubly linked lists use the 0th CDP to represent their start and end. Thus, writing to the 0th CDP's next field is, at best, inconsistent. However, such a write can occur via lines in Figures 8 and 9.

```
        list[list[cdp].prev].next = list[cdp].next;
```

Figure 8: DssCdpManager.sol#L121.

```
        list[last[dst]].next = cdp;
```

Figure 9: DssCdpManager.sol#L138.

- **The "tick" function can be called on an uninitialized Flip or Flop bid.** Flip's and Flop's `tick` methods should require that `bids[id].guy != address(0)` as other methods that operate on bids do. This will reduce the possibility that someone can successfully call `tick` with the wrong `id`.

C. Measuring klab specifications with Slither

This appendix describes scripts in which MCD contracts can be analyzed with Slither, a Solidity static analysis framework. These scripts identify TOB-MCD-003 and show that the behavior of functions behind all calls that use ABIEncoderV2 are tested by klab specifications.

Sample output:

```
[...]
140/456 functions are directly covered by a kspec
4/456 functions are reached by a kspec
312/456 functions are not reached by a kspec

Could not find function for klab spec:Flapper.kick(uint256,uint256)
Could not find 1/244 functions referenced in klab spec.
```

Overall results of the analysis show that (for non-test contracts):

- 30.7% of the functions are directly covered by a kspec.
- 1.75% of functions are otherwise reachable via calls from a kspec'd function.
- 67.54% of functions are not reached by a kspec.

Slither identified that kspec coverage was generally polarizing: Contracts with kspecs typically have function coverage close to 100%, while contracts without kspecs typically have function coverage close to 0%.

When assessing previously found vulnerabilities, we found that having kspecs associated with contracts did not ensure the contracts werere bug-free. Specifically, we observed that, despite all functions in `Pot` and `End` having associated kspecs, critical bugs such as instant Dai interest generation[4] and collateral theft during the end process[5] remained.

Contract coverage shown below indicates that the `dss` module is well-specified; however, contracts outside of `dss` would benefit from increased specification.

[4] Earn free DAI interest (inflation) through instant CDP+DSR in one tx: The Pot is susceptible to instant interest generation, which unbalances the Dai-collateral relationship.
[5] Steal collateral during `end` process, by earning DSR interest after `flow`: The Pot and End contracts are susceptible to collateral theft.

Name	Covered (%)	Covered (direct)	Covered (indirect)	Not Covered
BAT, CatFab, DGD, DSChief, DSChiefApprovals, DSChiefFab, DSGuard, DSGuardFactory, DSPause, DSPauseProxy, DSProxy, DSProxyCache, DSProxyFactory, DSRoles, DSStop, DSThing, DaiFab, DaiJoinFab, DssDeploy, DssProxyActions, ESM, ETHJoin, EndFab, FlapFab, FlipFab, FlopFab, GNT, GemBag, GemJoin1, GemJoin2, GemJoin3, GemJoin4, GovActions, GovPollingGenerator, JugFab, LineSpell, Median, MedianBATUSD, MedianDGDUSD, MedianETHUSD, MedianGNTUSD, MedianOMGUSD, MedianREPUSD, MedianZRXUSD, MultiLineSpell, Multicall, MulticallHelper, OMG, OSM, PauseFab, PotFab, ProxyRegistry, REP, RestrictedTokenFaucet, SpotFab, Spotter, TestchainPauseProxyActions, TokenFaucet, Value, VatFab, VoteProxy, VoteProxyFactory, VowFab, ZRX	0%	0	0	*
DSMath	16.67%	0	2	10
DSTokenBase	16.67%	0	1	5
DSAuth	33.33%	0	1	2
DSToken	45.45%	5	0	6
DSValue	50%	2	0	2
Flapper	90%	9	0	1
Cat, Dai, DaiJoin, End, Flipper, Flopper, GemJoin, Jug, Pot, Vat, Vow	100%	*	*	0

Note: The contracts shown above include all compiled contracts with non-empty function bodies in the `dss-deploy-scripts` repository.

Running the scripts

The MCD project reuses contract names and therefore requires updates to crytic-compile and Slither for out-of-the-box use. To work around this issue, we refactored contracts that had the same name with an additional suffix— "_DUP_<contractname>"—to avoid naming collisions. For example, we renamed `VatLike` in `cat.sol` to `VatLike_DUP_CAT`.

After refactoring, `run.py` and `analysis.py` were used to export a compiled code archive for Slither analysis. Basic metrics regarding functions not reachable/resolved from klab specifications are output with the script, while detectors can be run on the archive exported to `./EXPORTED_ARCHIVE.zip`.

To set up the script, have the latest versions of Slither and crytic-compile installed from GitHub and then install the tabulate dependency using the following command:

```
pip3 install tabulate
```

After installing tabulate, clone the `dss-deploy-scripts` repository, and unzip the provided script archive in the root of the repository. Afterwards, open a terminal, change directory to the root of the repository, and execute the following command:

```
python3 ./scripts/mcd-slither-analysis/run.py
```

This will analyze all contracts and provide basic coverage metrics.

Note: Upon running the analysis, the `EXPORTED_ARCHIVE.zip` containing the compiled contracts will be created in the current working directory. Subsequent execution of the analysis script will opt to use this exported archive as cache instead of recompiling. Due to the prerequisite contract name refactoring, the exported archive generated by Trail of Bits was included to avoid any refactoring requirement from the Maker Foundation team if they wish to quickly run the script. Example output has also been provided.

D. Symbolic exploration of DSS with Manticore

Manticore is a symbolic execution tool for analysis of smart contracts and binaries. We used Manticore to subject the MCD contracts to multiple symbolic transactions. This appendix summarizes our approach and the results of our analysis.

We directed our efforts toward the DSS contracts (e.g., Cat, Jug, Vat) as these represent the "core" around which the larger MCD system is built. Later in the assessment, we also incorporated the DssCdpManager.

Our Manticore script deploys the MCD contracts in a manner modeled after the `dss-deploy-scripts` repository. The Manticore script then symbolically executes some number of transactions against some subset of those contracts.

Given the intensity of our workload, we directed Manticore's search in a couple of ways:

- First, we did not allow Manticore to perform `rely` or `deny` calls following deployment. We had already exposed as much attack surface as possible, and there was little (if any) attack surface that `rely` could additionally expose. Conversely, `deny` could only reduce attack surface.

- Second, whenever a call required an ILK, we instructed Manticore to use either "DAI" or "ETH," corresponding to the two types of collateral we deployed. This saved Manticore from having to guess these values.

Manticore has a set of detectors for common Ethereum-related errors. We enabled all of these. In addition, we instrumented the DSS contracts with code to check for potential MCD-specific errors. Examples:

- The Vat features an `init` method for an ILK. We added an `inited` field to an Ilk to indicate that the `init` function had been called for that Ilk. We then sprinkled `assert` statements throughout the Vat to verify that, whenever an Ilk was used, its `inited` flag was set, i.e., it had been initialized. We were unable to get any of these assertions to fail, suggesting that the Vat uses only Ilks that have been initialized.

- We added the `invariants` function in Figure 1 to the DssCdpManager, and sprinkled calls to `invariants` throughout that contract. In certain calls, e.g., those involving `give`, the assertion that `list[0].next == 0` failed.

```
    function invariants() internal view {
        assert(urns[0] == address(0));
```

```
        assert(list[0].prev == 0);
        assert(list[0].next == 0);
        assert(owns[0] == address(0));
        assert(ilks[0] == bytes32(0));
    }
```

Figure 1: A function added to DssCdpManager.

While examining the successful runs produced by Manticore, we noticed some peculiar-looking calls to `file`. This led to TOB-MCD-011 and TOB-MCD-012. In addition, use of MCD-specific property tests like those described above led to entries in Appendix B concerning the DssCdpManager and Flip's and Flop's `tick` functions.

E. Converting unit tests to property tests with Echidna

Echidna is a property testing tool for Ethereum smart contracts. We used Echidna to generalize existing MakerDAO unit tests to cover a more diverse set of behaviors.

We generalized existing tests by taking static values and making them function parameters, then modifying `ds-test`'s `fail` to cause an assertion violation. This means that Echidna can execute these tests with random parameters, and should any assertions fail, alert with the parameters used.

```
    function test_join(uint amt0, uint amt1) public {
        setUp();
        amt1 = amt1 % amt0;
        address urn = address(this);
        gold.mint(amt0);
        assertEq(gold.balanceOf(address(this)), amt0);
        assertEq(gold.balanceOf(address(gemA)), 1000 ether);
        gemA.join(urn,                          amt0);
        assertEq(gold.balanceOf(address(this)),    0 ether);
        assertEq(gold.balanceOf(address(gemA)), 1000 ether +
amt0);
        gemA.exit(urn,                          amt1);
        assertEq(gold.balanceOf(address(this)), amt1);
        assertEq(gold.balanceOf(address(gemA)), 1000 ether + amt0
- amt1);
    }
```

Figure 1: A modified version of `test_join` *from* `vat.e.sol`.

This allowed us to take the existing test engineering and use it to achieve greater coverage. We can turn what was previously a single test case into thousands of different test cases with minimal effort.

Running the Echidna analysis

We added ABIv2 support to Echidna to support this audit. We are working to get this support upstreamed into `hevm`, and then we will remove it from our own codebase and make an official release.

In the interim, build the `echidna-test` from GitHub and invoke `echidna-test vat.e.sol --config dai_conf.yaml` to run the tests.