Contents

Audit of MakerDAO codebase

This audit was commissioned summer of 2016, and completed August 22, 2016. Any errors or omissions are mine. The MakerDAO community has been helpful, kind and is extremely intelligent – I look forward to continuing to work together.

The scope of the audit was to analyze everything deployed through "Phase 4" for correctness, fitness and vulnerabilities. The scope specifically did not include the CDP engine, (still in development), game theory or economics around dai, or compiler toolchain ecosystem concerns. While I have made some initial comments on CDP code as it is being written, the code and comments are not definitive. Throughout, I presume that the `solc` compiler is correct, although it has not been proven correct yet to my knowledge.

This version of the document elides line by line file comments (pages 13 to 45) for security reasons. Core-oversight and the security-audit team have the complete audit.

Thank you for the chance to learn from your team and community!

Peter Vessenes, August 2016.

Overview of Audit Methodology

Most solidity audits have relied on line by line code reviews to determine if there are bugs.

Two things make this difficult for MakerDAO, first, the nested structure of dappsys and the build process mean that there are many contracts – a `dapple build` on tip will compile 120 contracts. Second, these line by line processes can miss errors that exist over a large attack surface – the auditor may not be able to keep all relevant parts of the codebase in her head during the process.

Finally, the MKR software suite (and related contracts in dappsys) seem likely to continue to grow. It's desirable to have a solid and known baseline on which the entire team can build, rather than starting from ground zero with each new push.

Additionally, I would like this audit to serve as a useful guide for less technical community members, both to enhance understanding of risk points, and provide a sense of comfort over the care being taken with the MakerDAO code base.

Therefore, I have approached the audit from a number of different angles.

1) I did a line by line review, first coming up to speed with code commits as of Piper Merriam's review, second of all code listed as deployed by the Maker security team, and third using dapple to build a 'tip' version of maker-core to assess current state of the repository. I referenced this code review off commit 7eacb0.

2) I have segmented out common errors into a number of categories, and using some developed tools and individual inspection, have searched the codebase for these error categories. For many errors, some static analysis can be helpful. I will be publishing some tools used for this audit over the next month or so.

3) I have instrumented out some charting and graphing tools to help developers find likely places in their code that could be vulnerable, especially to recursive call and 'solar storm' attacks. This code builds on the code released by rayne, and will eventually be published on my github account.

Overall Analysis

The MakerDAO solidity code is **best in class**, well written and overall safe. It has abundant testing.

I would recommend the general architecture as industry best practice.

I identified no major bugs in the deployed codebase.

I would like to highlight a number of architecture decisions other solidity teams would do well to emulate.

1. The security design is excellent. Using frontends and per-function control mechanisms is innovative for ethereum, and best in class practices. It solves a number of problems around upgrading contracts, security and control of actions, as well as provides a resilient counterdefense against "solar storm" and other attacks.
2. The code makes extensive use of the dapple testing and build infrastructure for safety. Code pushed to the `develop` branch is of consistent high quality.
3. A testing culture is pervasive, developers brainstorm new tests, and these tests are extraordinarily simple to instrument in the dapple infrastructure.
4. Testing for overflow and underflows happens consistently and throughout the constellation of solidity files.
5. Use of `call` is extremely minimal.
6. Assertions are used throughout as a way to enhance safety and enforce expectations.

Overall, the MakerDAO community should be proud of its engineering team. They are doing great work.

Areas where improvement may be desired

Complexity

MakerDAO has only one major flaw in my mind; it is complex. It is a challenge to learn to use the build tool system, various Dappsys contracts, testing framework

and layering on an understanding of the core mechanics, then ensuring that both the CDP and auction system are working well.

This level of complexity means as a community it will be additional work to slot in developers. They will need an understanding of the goals, and to be directed to more focused parts of the codebase as a start. It's probably worth thinking about pulling in a project-management type person if you would like engineering delivery speed to increase – new people will tax existing resources, and those are very busy.

Validating Builds and Contracts

A major pain point with most smart contracts is answering deployment questions:

1) Which contract is currently deployed?
2) Which version of the contract am I talking to right now?
3) Which commit hash does the contract I'm talking to go with?

Dapple holds out promise here as a way to instrument answers to these questions, but the work has not yet been done to integrate these answers into the build process. This pain point is only going to get worse as time goes on, and I would recommend fixing it be made a priority.

At minimum, I would recommend the following as a strawman for discussion:

1) Every contract should get an internal variable referring to its git commit hash. Perhaps it could be called `_commitHash`. This could be set by dapple as part of the build process. Perhaps a repository URL could be included as well.

2) Every deployment should log the commit hash of each deployed contract along with the contract's address into an off-chain repository for reference. I would recommend these go into a directory in the git repository, although tastes may vary. It's possible that `maker-core` would eventually want to instrument these into an on-chain versioning system as well. If it is implemented on-chain, then a contract constellation could be asked for its full version status including subcontracts. This would be a huge step forward in safety on the blockchain.

3) Some sort of versioning system should be implemented in solidity, allowing sanity checks in contracts like `_dest.versionGreaterThan('2.2')`

4) Dapple or some other tool could then be instrumented to validate a given contract; the contract could be queried as to commit hash, that commit hash could be downloaded and built, then the resulting code could be compared to the deployed contract's bytecode, ensuring safety. This could even be instrumented on-chain as a service that contracts pay for. With cashing it would generally be performant after the first test build. This would necessitate storing the compiler version used in the contract as well.

Other ideas might be better than these, but I can say that these enhancements would be a significant help in the state of the art, and would help your developers sleep better at night.

Naming

In general, code variables are well named. However, I intensely dislike the Solidity standard of merely capitalizing event names as logging – e.g. `Transfer()` is not a function that does something, it just logs a transfer. See my rants in the Cognitive Load section of Appendix A for more on this.

ETC/ETH discussion

ETC poses some risks for an ETH-based DAO, and some opportunities. I will summarize risks here, with the warning that these are LIVE risks for the MakerDAO system.

First, the major risk is in the ETC and ETH contracts getting subtly out of sync. The simplest attack would be to make two proposals simultaneously.

- ETH proposal – update auth function, or some other innocuous call.
- ETC proposal – replace token database with attack database.

Each of these would carry the same proposal number, and confirmations for the ETH proposal could be replayed into the ETC chain, along with the trigger. The attacker need not be a manager or otherwise privileged. After the trigger is accomplished on the ETH chain, this trigger can be replayed on the ETC chain.

From there, all ETC is vulnerable. Similarly, auction bid pricing can be mirrored between chains, causing what I term "havoc" with the intended MKR/dai price setting mechanism.

It is an open question for all DAOs how they wish to manage their ETC and ETH funds – and a 'we don't care about ETC' perspective is a valid perspective. That said, a vanilla `send` from the ETC contract with the same address as its ETH counterpart could be replayed back into ETH. I do not believe this particular "double-cross" attack is possible with current MakerDAO contracts. But, it is worth being extraordinarily cautious about these attacks.

Mitigation

The simplest mitigation is to get all MakerDAO contracts on ETC and ETH chains moved to new addresses. At this point, they will be free to go their separate ways, allowing MKR to float in each place, and providing dai to both ecosystems.

However, it is going to be challenging to completely sever the contracts while dealing with replays – many (most? all?) ETH calls are getting replayed right now, often nearly instantly. Getting all of the MakerDAO contracts moved over to new ones is a complex process, and will typically require multiple confirmation steps.

Proposed Steps

First, change frontend contracts. This will protect all contracts which can only mutate state when called through the frontend. This move could be done in one of two ways:

1. With a 'many universes' assessment contract – 10 possible destination addresses can be instrumented, creating 10 slots. The hash of a prior block can be used mod 10 to choose a slot. In over 80% of the cases, the two slots for the two universes will be different. If they are not different, then the many universes assessment can be redone. This has the nice property of not relying on a race condition, but requires a bit of upfront work to specify the frontend contract addresses.
2. With a simultaneous replay attack on the proposals – each chain gets a different proposed frontend contract change, and they are issued simultaneously. Confirmations can be done on either chain and replayed. At the end of a trigger, the frontend contracts will change.

Second, for safety, all contracts should be updated, using similar methods. With split frontends, I believe this becomes an easier matter.

This process is just a proposal, and should be discussed with the development team before any actions are taken.

Token System Considerations

Tokens are marvelous, one of the great innovations in Ethereum. However, traditional token management systems as specified by the ERC20 example code are brittle. They cannot audit their own state. They can fail terribly when underflowing, either allowances or token balances. The current MakerDAO token contracts are well written and do not have obvious flaws.

But, they are still brittle. An error introduced later during development could cause real difficulty.

I have a number of recommendations about tokens, but they will require some discussion among the MakerDAO and Dappsys architecture team. I suggest that the team dialogue on the following questions.

1. How often should we check that token state is what's expected? Are there audit functions that we should implement in solidity to check this?

2. Do we want decorators like `token_invariant` which will throw a function that changes the token supply?
3. Do we want decorators like `token_increases` or `token_may_increase` that throw if the token supply does not change as expected?
4. What are our plans if token supply is too low? Too high? Under/Overflowed? How do we wish to mitigate / remediate?
5. Is there a systemic way to check that users are not leaking tokens, getting too many, or something else bad is happening?

Game Theoretic Concerns

Withholding Approval

A four of six multisig for most approved actions raises questions about what happens if three of the six disagree on a proposed action. In most cases, this kind of dissension is precisely what the instrumentation is for – the community will rely on humans to decide important things.

In the general case, a majority will be required to *take action*. This is the intent of the rule.

However, there may be cases where inaction is deemed obviously undesirable. In that case, a minority can force it on the rest of the community. For example, if there is a security breach requiring quick movement, and three of the six can be somehow compelled to wait, or perhaps even subject to attack, the contracts could be at risk.

Because of the way the contracts are structured, four of the six could remove the other two participants permanently from the contracts, replacing them with 'sybiled' addresses. This would be a reduction in security. Even worse, the burden put on approvers is high – they will need to be able to read source code and think critically about it to do a good job as gatekeepers.

I would like to see copious logging around authority changes as an early mitigation, although this would usually just tell the community what happened, rather than prevent.

I would also suggest that the MakerDAO community make available a small 'audit' budget, one for each of the six approvers, to be used at their sole discretion to retain an outside inspection of worrisome code. In most cases this won't be necessary, but I think it could provide an extra layer of safety and security to approvers who are less technical.

Creating DAI price movements

It is possible, if a party is willing to pay for it, to create long term dai price instability. This is a systemic risk to the MakerDAO community. I have not

calculated the expense of holding a long-term price delta, (or creating oscillations in the price), but I think it would be worth having economists or game theorists in the community spend some time getting bounds on these numbers. As market trading and usage increases, I *believe* that it will become more expensive to de-stabilize the dai pricing, but it would be nice to prove this.

As an example, Bitcoin and Ethereum mining costs can be used to estimate the costs of subversion of the network over a given period of time. This is very helpful for reasoning about fitness of the currencies to an intended use. I suggest MakerDAO do some work to estimate the cost to griefers and other concerted attackers here.

Underflow and Overflow Errors

Like all maker-core code, care has been put into addressing this common set of errors. In particular, util/safety.sol has functions `safeToAdd` and `safeToSub` which check for under and overflow scenarios. Later contracts decorated as assertive also check and `throw` if a possible underflow or overflow could occur. While former revisions of the codebase contain both under and overflow errors, the MakerDAO team is clearly keeping an eye out. I did not identify existing underflow or overflow errors that were realistic during my final audit. But they have a pernicious tendency to creep in; I recommend ongoing vigilance.

Overflow

Default integer types in solidity are uint256. This makes overflows unlikely for most use cases. However, there circumstances where overflows can occur. In particular, solidity casts uints down to the smallest type it can when `var` is used. Thus a loop like for (var i = 0; ...) will overflow at i = 256.

Overflow checking is abundant throughout maker-core. I found no realistic overflow bugs in the currently deployed code.

Underflow

Underflow bugs with unsigned integers can be pernicious. Anecdotally, maker-core have in the past thought less about underflow than overflow during coding. For example, this sort of construct occured in the phase2-era codebase – not incorrect, but demonstrating where developer attention is focused.

```
uint value;
uint balance;
...
function withdraw(uint value) {
if (value >= balance) {
```

```
  if safeToAdd(balance) {
   doSomething();
  }
    balance -= value;
}
}
```

To reiterate, the code as written is not incorrect. However, because `withdraw` is vulnerable to a recursive call attack through `doSomething()`, if this function is called twice during recursion, it's possible that `balance` could become negative. Because `balance` is a `uint`, it will roll over, and become very large.

Because `balance` is now very large, the next call of `withdraw` will be allowed to withdraw any amount left in the contract.

If `balance` were an `int`, not a `uint`, then `withdraw()`'s sanity check will fail even after a successful recursive call. For this reason, I recommended to the maker-core developers that they convert most `uints` to `ints` throughout the codebase.

This led to vigorous debate, with the prevailing decision being to leave these as `uints`. There are a variety of reasons put forth, some compelling. In particular, compatibility with ERC-20 token tools would require an additional layer, adding complexity. Nevertheless, I reiterate here that the code will have less terrible failure modes when underflowing if these items are of type `int`.

Recursive Call Errors

The initial codebase as deployed had multiple recursive call vulnerabilities. Some were fixed organically as the code quality improved, and one in particular was fixed shortly before TheDAO was hacked.

Much has been written about recursive calls, and the MakerDAO team is well aware of the risks.

A few things mitigate most recursive call concerns in the codebase.

1. The `auth` decorator is used liberally.
2. In most cases, basic auth ("owner" auth) is replaced or augmented with specific limits on function calls.
3. There are very few places `call` is used. In particular, `trigger()` must be used to trigger a call.

It is possible that recursive call vulnerabilities can creep in through concerted attacks. I'm not aware of any attacks that can be accomplished without the collusion / duping of existing `auth`ed contract participants.

However if any of the standard-form contracts in a given environment can be replaced, much of MakerDAO could be affected. Most problematic would

be `frontend` contracts, but difficulties could arise anywhere, with risks up to and including the loss of all funds. The only way I can see that this would happen would be through compromising four of the six multisig signers, most likely through innocuous seeming code which did something bad. For this reason and others, I have recommended that the multisig signers be allocated a small personal audit budget to be used at their sole discretion in case of a worrisome `trigger` request.

Areas the Audit Does Not Address

Economics

The audit does not assess the various hard-coded constants for pricing incentivization around dai. It does not address or assess the likelihood that dai will remain stable in price in relation to the underlying. It does not provide firm calculations on the cost/benefit of subverting dai (or possibly MKR) for attackers, whether that incentivization is internally motivated, or external.

I would recommend some theoretical work be done on these issues, though. Particularly around the costs of subversion.

Test Coverage

The MakerDAO testing code is best in class. I did not thoroughly audit the tests for logic or code errors.

I would recommend the testing infrastructure to other solidity coding teams.

Low Severity Concerns

Stuffing Attacks

If a contract is sent money through `send` or `call`, then the contract's fallback function is called. Ordinarily solidity allows calling code to check the success or failure of the `send` or `call` as follows:

```
if (contract.send(1 ether)) {
    // Success! Do something here
} else {
    // Failure! Do something else
}
```

In general, solidity will return `true` for these if the call did not raise an exception, most typically via `throw`.

However, if there is no fallback function, then solidity will return a long number; this is a technical decision by the solidity team. Unfortunately, this number will not be zero, and will therefore return `true` in these tests after coercion to a `bool`.

This sucks. It's a bad language decision. It means you may think you were successful sending ether somewhere, but won't have been. I'm calling this a stuffing attack because you sort of stuff the value back in the sending contract. Better names are welcome, though.

In general this will not be a showstopping vulnerability, because value is coming back to the sending contract. However, for contracts that rely on balance being exactly what is tracked through some other means, this could cause problems. I can imagine a reasonable looking contract that underflows a customer balance after stuffing, for instance.

I have noted where the code may be vulnerable to stuffing in the file by file sections, but I rate this low risk right now because these attacks are largely theoretical, have not been seen in the wild, and do not provide an avenue for pulling value out of the vulnerable contracts in the general case.

Underflowing Numbers

When the `var` construct is used to initialize an integer, the smallest possible uint fitting the assignment is used. So for example, solidity interprets `var i = 255` as `uint8 i = 255`. This can create surprising situations. There are a number of areas in the testing code which use this formulation; because it's testing code in old commits, I have not evaluated which may be incorrect. In any event, the code should be scanned for these, and actual sized `uint`s should be used throughout.

In production code, I found one that is almost problematic, but avoids it, perhaps by luck.

- `easy_multisig.sol`: Line 141

`var confs = a.confirmations;` If a.confirmations is less than 256, confs will be a uint8. If confs were increased later in the code, this would cause an overflow. However, `confs` is a dead variable, unused in the rest of the code, and the entire line can be deleted as it is written now.

Discussion of Files in Maker-Core

I started with the maker-core repository using `git clone --recursive`. At time of final analysis, the tip commit was "7eacb0".

I used the command `find . | grep sol | grep -v test > filelist.txt` to create an initial list of files. This resulted in a list of about 220 files. (See Appendix A if you are interested.)

Note that some of these files may not be included in the deployed contracts. As mentioned above, I instrumented dapple with the `dump-directory` directive to output exact files used. dapple reports roughly 120 contracts as pending for compilation, after deduplication.

The astute reader will notice that many of these contracts appear to be the same file; this is the case. The dapple build mechanism will include nested directories (hence the `submodule init` in the git pull). However, because of git's submodule semantics, there is no guarantee that the *versions* of each file are the same.

This recurring theme, that is, 'which files are deployed?' is a pain point for MakerDAO. I recommend a set of next steps in "Validating Builds and Commits".

File by File comments:

I evaluated each of the files by hand, line by line, with the exception of the maker-distributor/maker-user files. I believe these are old, and not in use.

Pages 13 through 45 of the original report have been withheld for review by core-oversight and the security audit team.