SAI Coin
Code Review

Jordi Baylina <jordi@baylina.cat>
Barry Whitehat<barrywhitehat@protonmail.com>
Adrià Massanet <adria@codecontext.io>
Eduardo Antuña <eduadiez@gmail.com>
Arthur Lunn <alf40k@gmail.com>
Antoni Martin<antoni76@gmail.com>
Griff Green <griff@giveth.io>

Release
V1.3 / 2017-12-15

Executive Summary

During the last month, a multidisciplinary team composed of five Solidity experts, one economics expert and one general digital currency expert have been working together with the MakerDAO team to understand, analyze and audit the security of the SAI code.

We went deep into the code and found six high severity issues, six medium severity issues and three low severity issues. Many of this issues, are either solved in the code, have some work around, have some contingency plan or are just are low severity. However, **the code has two structural issues that require a significant architectural change in order to be resolved.**

The first issue is related to PETH holders' reward mechanism. The problem comes from the way the system shares the gains and the losses between PETH holders. It does so by minting or diluting PETH tokens at specific, well defined moments in time. This means that a malicious whale can easily game the system by buying or selling huge quantities of PETH just before and after those mintings / dilutions are made. This takes the rewards that should be given to the PETH holders exposed to the risk, and gives an advantage to a malicious whale that buys PETH just before the PETH's are burned capturing an oversized portion of the distributed rewards with no risk.

The second important issue is the possibility of front running. Because any user can monitor pending transactions in Ethereum's transaction pool and then generate a transaction with a higher gas price to get it mined before any transaction they desire, malicious actors can monitor for the oracle's price updates or global settlements and mine transactions before they are mined to gain an advantage.

These two issues are problems with the implementation of this system and do not in any way undermine the concept. We believe this system can and will work once these issues are addressed. However, we think that the solutions to these issues will require weeks as opposed to days to solve. We propose potential solutions in this document.

We recommend the MakerDAO team publish the code, fix these important issues, and have more independent audits before the launch. Unfortunately we believe there are likely to be more issues in this code beyond what have been identified, and addressing the issues described in this document may introduce more bugs into the system.

Table of Contents

1. Introduction

1.1 About this review

The primary focus of this audit was to analyze the core SAI contracts. We focused our efforts on the Solidity contracts located in https://github.com/makerdao/sai commit fcec04a9486392dbf45fe6b54ed27973f25e4932/src

In order to do this review, we invested time to understand the white papers, the economics behind the model, and all the specific jargon used throughout the code.

We assembled a multidisciplinary team to get deep into the project and find as many issues as we could in the three weeks we set aside to review this code.

Scope: Solidity & Economics

We critically analyzed SAI's Solidity code and performed in-depth economic analyses in order to understand the system. We established a test deployment of the system to investigate numerous attack scenarios by simulating multiple market events and found several high severity vulnerabilities.

The Solidity portion of the audit is something our core team is most experienced in. We followed the normal pattern of searching for bugs and performing a critical review to ensure the code matches the intentions of the developers.

The economic analysis required us to bring in a team member with a classical economic background in order to understand the incentives of each actor and to understand the economic mechanisms that stabilize the SAI coin. Our opinionated economic analysis, despite having this expertise, should in no way be considered exhaustive. This system is very complex with many potential malicious actors and non-conventional attack vectors.

Out of Scope: Oracle's, User Interface & Deployment

The SAI system relies heavily upon a federated oracle and a clean deployment, neither of which were analyzed in depth.

We have not analyzed how the oracles obtain the market price, the independence and the trustiness of the oracles, or the security of those oracles. We expect that MakerDAO will explain how these oracles work and what the securities that they provide against a possible sharp crash in the ether price or some other failure in the oracle system, but this was not included in this audit.

We have not analyzed the governance contracts that will be able to modify some of the parameters and call the global settlement. They can be a single account, a multisig or a DAO contract.

User interface vulnerabilities are also out of scope in our audit. As was the case with the EtherDelta code injection attack in September 2017, there are numerous traditional attack vectors present whenever interacting directly with a web based user interface. While many of these attacks are relatively avoidable, the possibility of such attacks should still be acknowledged.

The deployment process is especially complicated and extremely critical for the SAI system to function as expected. We do have opinions on acceptable ranges of deployment parameters and upon deployment we will work with the MakerDAO team to ensure the deployment process is done correctly. If there are issues upon deployment we will make sure it is known, however we have confidence in the MakerDAO team and do not expect any issues.

1.2 Disclaimer

This document includes opinions and recommendations. All the opinions are our own and independent of the MakerDAO team. We found issues in the Solidity code and in the general design of the system but that doesn't mean that we found all of the issues.

About the recommendations, they are not intended to be the solutions to the problems. Some of them may even be bad ideas. They are only included as a starting point for the research of the solutions.

It's also important to consider.

- We did not make any low level reviews of the assembly code that will be generated by the Solidity compiler.
- We did not verify the deployment of the contracts or the initial configuration.
- We did not analyze any of the code other than the Solidity smart contracts.
- We did not analyze the governance mechanism.

Security audits like this one reduce the risks of issues, but they do not warranty bug-free code. We encourage the community, especially the MakerDAO community that will be using these contracts, to continue to analyze the code and to inform themselves before interacting with these or any other smart contracts.

1.3 Team Introduction

Jordi Baylina <jordi@baylina.cat>
@jbaylina
Over 30 years experience as a developer, and over two years spent researching Ethereum. One of the strongest Solidity developers in the world. Co-founder of the White Hat Group which played a major role in rescuing funds from TheDAO and Parity Multisig Hacks, and author of the MiniMe token contract, the elliptic curves Solidity library as well as numerous other established contracts in the space. Currently working towards formalizing the ERC-777 token standard.

Barry WhiteHat <barrywhitehat@protonmail.com>
@barrywhitehat
An anonymous white hat.

Adrià Massanet <adria@codecontext.io>
@codecontext
Over 18 years experience within the areas of security, cryptography, and digital identity software development. An established computer science engineer with several security audits conducted over the past year.

Eduardo Antuña <eduadiez@gmail.com>
@eduadiez
An Ethereum DevOps and Solidity developer with an established history of work on software engineering during the last eight years. Frequent Giveth contributor and currently a member of the Swarm City development team.

Arthur Lunn <alf40k@gmail.com>
@arthur.lunn
Currently finishing a degree in Computer Science and Globalization with a capstone project focused on smart contract engineering. Involved as a freelance developer in the Solidity space with a focus on governance.

Antoni Martin<antoni76@gmail.com>
@antoni76
An economist with 20 years experience in the financial sector and a firm understanding of both blockchain technology and the nuances of novel, emergent economic models facilitated by blockchain progress.

Griff Green <griff@giveth.io>
@griff
A well connected and respected person in the Ethereum space. Community manager for TheDAO, led the cleanup effort for TheDAO Hack from every angle, co-founder of the White Hat Group and Giveth. Holds a Masters Degree in Digital Currency and a frequent contributor to many projects and security audits in the space.

1.4 Methodology

In order to provide a focused analysis on the provided smart contracts we undertook the following steps:

- Understand the provided whitepaper in detail
- Understand the provided source code
- Define the primary and secondary objectives of the work, regarding the current document deliverable and the kind of analysis to be done
- Read and analyze the previous security audits done
- Iterate over a frozen code base
    - internal analysis
    - share findings with the MakerDAO team
    - clarify possible misunderstandings with them
    - analyze their fixes, when applicable
- Consolidate, write and deliver the report

The following table describes the analysis performed to the current code, and the depth to which each analysis has been done. When full, we feel the analysis is complete; when partial, we feel further analysis would be beneficial, and when supplemental we feel that the task is not in the scope of analysis but it has been also reviewed

Type	Description	Depth
Standard code review	includes the common issues that are usually reviewed in all applications: code readability, specification matching, duplicated code, parameter checking, precondition assertions, code conventions, documentation, testing, code coverage, overflow/underflows, unused variables, bounds checking, etc...	Full
Ethereum security specifics	Specific issues related to Solidity and the Ethereum execution environment, see Appendix A.	Full
Standards	Standards fulfillment	Full
Standard security review	Includes defining assets to be protected and checking STRIDE attacks on the system[1]	Partial
Application logic	Check if code logic matches with the expected behaviour defined in the white/purple papers	Partial
(Crypto)Economics	Checks for (crypto)economic points of failure, and weakness in punish/incentivization systems	Supplemental

[1] https://en.wikipedia.org/wiki/STRIDE_(security)

Decentralization	Checks for centralized points of failure	Supplemental
Simulation	Simulate the system	Supplemental

Our findings are presented in the form V0##-XX, R00#-XX, and are specified in the Review Results section of the document, using the following pattern:

Description

Description of the finding

Location

Where is located (if applicable)

Severity

The threats have been classified into five groups (Critical, High, Medium, Low, Comment) depending on the impact and the likelihood of the event. Sometimes it is really complicated to guess the likelihood or the impact of the finding. We use OWASP threat classification.

	Unknown Likelihood	Low Likelihood	Medium Likelihood	High Likelihood
High Impact	Medium	Medium	High	Critical
Medium impact	Medium	Low	Medium	High
Low impact	Comment	Comment	Low	Medium
Unknown Impact	Medium	Medium	Medium	Medium

In some issues we also describe the operational environment or the threat agent that are more likely to be applied in the finding context. See Annex A for more information about them.

Status

“Fixed” means that reviewer team considers that the issue has been repaired.
“Not fixed” means that the issue is still present.

Comments

Our thoughts on the issue. The recommendations provided are not based on a detailed and deep analysis of the underlying problem, therefore they should be considered as a reference and not a definitive solution.

2. Review Results

2.1 General considerations

The system and documentation are built using specific jargon. The glossary that defines the specific jargon can be found in the documentation[2]. Special attention should be paid to this jargon while reading this review. The advantages of using this specific jargon are explained in the purple paper[3]. It should be noted that this jargon creates a barrier to entry for developers and interested parties that want to understand the system.

The use of jargon, and the lack of comments, makes it difficult to understand the code. While this may be an advantage for experts, it is difficult for newcomers.

As previously acknowledged by the Trail of Bits audit. The security of this code is highly dependent on the deployment and the initial configuration. We believe this issue as well as the rounding issues brought up by the Trail of Bits audit are still valid. Since the Trail of Bits audit covered these rounding issues we didn't investigate the rounding errors further. If these are still seen as an issue potentially adding degrees of precision could help alleviate the rounding issues without adding complexity.

We audited the part of the DappSys libraries in the context of the SAI system. This should not be constituted as a full audit of the DappSys system and independent analysis is required in order to ensure the security of those contracts. Any contract that was used in some way through the SAI platform was checked but these contracts were not formally within scope.

It is crucial to understand that the health of the SAI system is very much linked to the health of the Ethereum system. If there is any liveness issue with the Ethereum network, if blocks cannot be mined due to a DoS attack, a 51% attack or any other reason, the integrity of the SAI system is automatically compromised as the SAI platform does not have proper fail-safes for Ethereum liveness. The full economic implications of this are not fully explored.

A full audit was not performed against the oracle solution, but on a conceptual level the strength of the SAI system is directly linked to the aggregate depth of all books linked to oracle solutions so maximum transparency for the oracle solutions should be provided.

[2] http://makerdao.com/purple/#sec-7
[3] http://makerdao.com/purple/#sec-1-3

2.2 Review results table

ID	Problem	Severity	Status
V001-HI	`join() - boom() - exit()` - $$$	High	Not Fixed
V002-HI	`exit()` before `bust()` $$$	High	Not Fixed
V003-HI	Cap Bounding Issue	High	Not Fixed
V004-HI	exit just before global settlement	High	Not Fixed
V005-MD	System is not prepared for HardForks	Medium	Not Fixed
V006-MD	Price Crash Under-collateralization Attack	Medium	Not Fixed
V007-MD	Price Front Running Attack	Medium	Not Fixed
V008-MD	Incremental Boom Burst	Medium	Not Fixed
V009-MD	Negative Collateral Attack	Medium	Not Fixed
V010-LW	ERC-20 Concern	Low	Not Fixed
V011-CM	Direct Token Transfer Concern	Comment	Not Fixed
V012-CM	Operations on unsafe CUP	Comment	Not Fixed
V013-CM	Tradeable PETH	Comment	Not Fixed
V014-CM	System values concern	Comment	Not Fixed
R002-HI	Big Gap Exit issue	High	Fixed
R003-MD	Dust CDP’st	Medium	Fixed
R004-LW	Any body can shut an empty cup	Low	Fixed
R005-CM	Drip Recursion	Comment	Fixed
R006-LW	Uncovered code	Low	Fixed

2.3 High Severity Issues

[V001-HI] `join() - boom() - exit() $$$`

Location	Severity	Status
Tub.sol & tap.sol	High	Not fixed

Description

It is possible for a malicious actor with a lot of ether (a whale) to steal a majority of benefits generated by the system's incentivisation mechanism.

The smart contracts distribute benefits through a mechanism in the `tap.sol` contract. When a surplus of `SAI` is generated, they are sent to the `tap.sol` contract. Later, when a user exchanges that `SAI` for `PETH`, those `PETH` are burned. This exchange and burn process is handled by the `boom()` function.

This attack requires a whale to execute 3 operations in the same transaction through use of a specially crafted contract:

1. `join()` with as many `ETH` as they can. At this point the attacker would control a large share of all the `PETH`.
2. `boom()` to exchange the `PETH` for `SAI`, burning the `PETH`. This burn is equivalent to a traditional dividend payout with the `PETH`, holders receiving payouts proportional to individual `PETH` balances. The attacker holds a large percentage of the `PETH` and will therefore get most of the systems rewards.
3. `exit()` getting back all the ether invested in step 1, plus the benefits of step 2 with zero risk.

Because all this is done in the same transaction, this allows the attacker to collect an unfair portion of the rewards generated by the system without risk.

This bug brings us to the following questions:

1. When should the profits for PETH holders be generated?
2. When and how should these rewards be distributed?
3. To whom are rewards distributed?
4. Is this distribution fair?

The system generates 3 kinds of benefits:

1.- The benefits that come from the `gap` at the time any exchange is made within the system.
2.- The benefits generated by `tax` in a continuous based system `drip()`

3.- The benefits from the liquidation penalty, `axe`, after a `CDP` is liquidated

The first benefits are generated and distributed at the moment of the exchange which is acceptable. The individual *paying* the benefits controls the benefit *timing,* thus there exists a negligible surface for exploitation.

The other two benefits are distributed when `boom()` is called, and those benefits are distributed proportionally to the `PETH` holders at that specific block. Since an individual *receiving* benefits controls the *timing,* there is a much larger surface for exploitation.

This distribution does not coincide with who is really holding the risk of the system. This attack exploits this difference by calling `join()` before the `boom`, and `exit()` after.

Comments

To fix this vulnerability requires rewriting an important part of the smart contracts.

One possible solution is to create a `Reward` token. This token is minted and distributed on a regular basis and is proportional to the `PETH` holders of the system at a given time. For example, this could be done in the function that updates the price from the oracle.

These rewards may be modulated also depending on the risk that `PETH` holders are holding at any time. For example, the system may want to distribute more reward tokens when the price is falling fast than when the price is falling slowly or is rising.

Then, when benefits are shared in `boom()`, the `Reward` token distribution can be applied to distribute the benefits.

Because the `Reward` tokens are accumulated over time, a `PETH` holder that `join()`'d just before `boom()` will receive zero benefits, because they will not have received any `Reward` token yet.

One final detail to optimize this implementation is that the `Reward` tokens should be burned or diluted with time. In other words, when the benefits are paid they should go mostly to the `PETH` rewarded in the last days and weeks, not two or three years ago.

One way to achieve this effect, is to multiply the reward tokens distributed at time `t` by a factor that's exponentially growing.

$$RewardsToDistribute = f(\Delta price) * e^{(t/\tau)}$$

This allows τ to be adjusted to ponderate more or less the near past.

[V002-HI] `exit()` before `bust()` $$

Location	Severity	Status
Tub.sol & tap.sol	High	Not Fixed

Description

When a price falls hard, `PETH` holders may want to `exit()` the system before the dilution is applied. Here a large holder can call:
1. `free()` to withdraw `PETH` and then `exit()` to take `WETH` to the limit that their `cup` allows.
2. `bust()` to mint more `PETH` watering down all `PETH` holders.
3. `join()` again to get `PETH` from the `WETH` and then `lock()` the `PETH` back to their CDP to keep as safe as they were at the beginning of the transaction.

They can do this with no risk if it all happens in a single transaction. They can use this to reduce other users `PETH` holdings while increasing theirs. And thus increase their proportional share of the underlying `gems`. This transaction gives a net gain for the `bust()` caller.

After a dilution, the collateralization of the remaining CDP's is lower (because `per` is lower). If the attacker holds a huge proportion of `PETH's` this can be used to "attack" the other `cups` making them less safe and possibly biting them in the same transaction. It can cause an avalanche effect especially in a bear market situation.

This bug is especially problematic because it allows a large holder to dilute smaller users. Also, it is possible for a cartel to form solely to operate this attack as it will scale better with the more `PETH` that you have. A smart contract could be written to run this attack at scale with zero trust required.

This attack uses the same flaw on the system that V001-HI had, but in this case it applies to the `PETH` holders.

Comments

One possible solution to this is to not allow `exit()` (or any other transaction that increases the full system risk) while the price is falling at a certain rate.

See V008-MD which also applies. The idea is that the exchange rate should take in account the dilution that is going to happen after a `bite()`.

[V003-HI] Cap Bounding Issue

Location	Severity	Status
tub.sol	High	Not Fixed

Description

The system operates continuously until the amount of SAI minted is greater than the `cap` value that is set by the governance mechanism. When this happens it becomes impossible to mint more SAI.

When this happens, the value of SAI may become more valuable than its paired currency. As you need SAI to withdraw locked currency, and there is no source of SAI other than the market.

Comments

The obvious solution is to remove the `cap` limit or else to update this value dynamically as it is approached. However, it is important to note that if it is feasible for a single user or group of users to breach the `cap` limit. They can use this to manipulate the ability of other people to call `draw()` and interact with other important features.

The existence of the ability to stop SAI production opens significant attack vectors that can be used to sow FUD and manipulate the users of this system.

Another option is to increase `gap` proportionally when the system gets closer to the `cap`. This would make the system dynamics not change so abruptly.

Here is a good place to mention also that if the system holds an important part of all the ETH total supply, this can affect the ETH-SAI market and may also have some implications in the economical model.

Having a `cap` for the `PETH` instead of a SAI with the dynamic `gap` may be a good starting point to approach this issue.

[V004-HI] `exit()` just before global settlement

Location	Severity	Status
tap.sol, top.sol, tub.sol	High	Not Fixed

Description

In many situations, a global settlement in an undercollateralized system will implicate a dilution, so if any `PETH` holder sees a `cage()` transaction in the pending transaction queue, it's going to be worth it for them to try to `exit()` as many `PETH`s as they can and let the `cups` reach their maximum risk, getting as much SAI's as possible. Of course this depends on the `_fix` used which can also be monitored.

As an example, imagine that the price of ETH goes down very much, but in the system, because of the rate limit, the price is still quite high. It's decided to do a total settlement for a price in the middle/low price. Then it's profitable for a user to just `draw()` as much as he can or `free()`, `exit()` as much as he can in your `cups`.

In other words: Because of the front-running effect you are not prioritising the SIA withdraws against `PETH` withdraws in a `cage()` (global settlement) situation.

The optimal strategy when a global settlement transaction appears in the mempool is:

1. If you have a `cup` that has zero `ire` and the`_fix` price is high, `exit()`
2. Else, if the `_fix` is high, maximize your leverage by withdrawing more SAI.

This means that in a perfect system, as soon as a global settlement transaction appears, the system will become undercollateralized.

In most cases, people will exit before a global settlement happens. In general if you want to keep the value for `SAI` holders, this damages the `PETH` holders. Therefore they will set the `cups` to the maximum risk and `exit()` with as much as they can.

Comments

This is a front running bug similar to V005-MD and V006-MD. It can be solved by applying the same techniques described there.

2.4 Medium Severity Issues

[V005-MD] System not prepared for HardForks

Location	Severity	Status
n/a	Medium	Not Fixed

Description

In case of an important Hard Fork where the value of both resulting fork is expected to be different to 0, the system does not provide a good mechanism that allows PETH holders to capture the value in the resulting chains.

Comments

In case that the secondary fork is going to capture a low part of the value, then the optimal strategy is to `cage()` the secondary chain with a very high price.

In case of a more symmetric fork, the thinks get more complicated, and in this implementation, the system should be global settled before the fork.

[V006-MD] Price Crash Under-Collateralization Attack

Location	Severity	Status
n/a	Medium	Not Fixed

Description

If there is a pending transaction in the transaction pool that indicates there will soon be a crash on the price of ether it is possible for an attacker to `join()` the system with a large amount of ether, convert to `SAI`, and `lock/draw` on a `cup` to reach maximum risk. After the price update is applied, this `cup` will immediately be undercollateralized.

This potentially leads to a situation where the `cup`'s ratio goes to under 1 which should never happen. Since the attacker can process this instantly, there is no potential downside and relative to other ether holders the attacker will experience less lost value. Assuming that the market bounces back from a flash crash this leads to a net-gain with little to no risk. Even if the market does not bounce back, this scenario leads to a more beneficial state at a cost to system participants.

Comments

If the fall price rate is limited, then this attack is more difficult, but SAI may break the peg `SAI/$`. This situation may bring to a massive removal of the collateral in order to avoid the unavoidable dilution that will happen in near future. To avoid this, when the real market price is not adjusted by the system price because of this limitation in the decay rate, the system could enter in a state where nobody can `exit(), free()` or `draw().`

[V007-MD] Price Front Running Attack

Location	Severity	Status
n/a	Medium	Not Fixed

Description

V004-MD is just an extreme example of a set of vector attacks that we describe more genericly in this issue.

This system assumes liveness in order to update the price oracles. However, the liveness guarantees of Ethereum are quite weak. The delay for the system to be aware of the actual price is at least a block, but this delay could even be increased by a malicious actor running a DOS attack. Also, regular events on the chain like congestion, hard forks, etc, can carry this delay to higher values as well.

Currently, miners have the power to select the order in which transactions are executed, making it possible to front run price updates. An attacker will be able to buy before a price update and then sell afterwards, making a profit on the difference between the two prices. The `gap` parameter should be discounted to include the profit from front running.

This delay allows bigger arbitration opportunities in the market.

However, during a DOS attack or a long period where no price update is possible this attack becomes more dangerous. Assuming the attacker is running the dos +51% attack, they can select the transactions that are included in a certain block and reject others. The optimal strategy is for the attacker to select their prefered price from the steam of price update transactions. And then `bite()` and `bust()` these `cups`. Taking the `gems` and then using the `gems` from which they profited to attack even bigger `cups`.

This potentially opens additional attack vectors that must be investigated further.

Comments

One potential mitigating factor is to freeze the price update while DOS attack is happening.

Currently, the oracle creates a transaction so that each block they make a new transaction to update the price.

Changing this paradigm slightly so that each user transaction contains an oracle singed message of the current price this could remove the possibility to front run the price update transactions. As each transaction would contain this price update with its timestamp.

Another possible way to solve it would be to use the price oracles to broadcast each transaction to the system, including the current price in each transaction. This oracle could decide upon the definite order of transactions. This would make the system less susceptible to transaction ordering attacks.

The former allows for block level price updates ~15 seconds. The latter allows for per transaction price update.

[V008-MD] Incremental Boom Bust

Location	Severity	Status
tub.sol, tap.sol	Medium	Not Fixed

Description

When a `boom()` or `bust()` that is going to mint/burn tokens is pending, the exchange rate calculated within any `join()`, `exit()`, `boom()`, `bust()` may not take into account those tokens that will be burned/minted.

This generates a set of inconsistencies if those operations are done with a simple transaction or are split in many chunks.

Comments

The way to solve this is to take into account the `PETH` tokens that are going to be minted/burn when the exchange rate price is calculated. The transaction should still mint/burn just the needed tokens for the transactions, but should take in account all the tokens that will be mint/burn to calculate the actual exchange rate.

Here is a draft of what `bust()` could look like if this fix were applied. (The same should be done for `boom()`, `join()` and `exit()`).

```
function bust(uint wad) {
    // x = [ extra skr Needed to mint in order to cover all the woe after dilution ]
    // ask(joi + x) = woe =>
    // woe = (joi + x) * gap * s2s' =>
    // woe = (joi + x) * gap * tag'                                     / par =>
    // woe = (joi + x) * gap * pip * per'                               / par =>
    // woe = (joi + x) * gap * pip * (  pie / (skr.totalSupply() + x)  ) / par =>
    // woe * par * (skr.totalSupply() + x) = (joi + x) * gap * pip  * pie =>
    // x * woe * par + skr.totalSupply() * woe * par = joi * gap * pip *pie + x * gap * pip * pie =>
    // x * (woe * par - gap * pip * pie) )  = joi * gap * pip * pie - skr.totalSupply() * woe * par   =>
    // x = (joi * gap * pip * pie - skr.totalSupply() * woe * par) / (woe * par - gap * pip * pie) )

    var x = (joi * gap * pip * pie - skr.totalSupply() * woe * par) / (woe * par - gap * pip * pie) );

    if (x<0) x=0;

    // _ask  = wad * gap * s2s'                                          =>
    // _ask  = wad * gap * tag'()                                    / par =>
    // _ask  = wad * gap * per'()                              * pip / par =>
    // _ask  = wad * gap * (  pie / (skr.totalSupply() + x)  ) * pip / par

    var _ask = wad * gap * (  pie / (skr.totalSupply() + x)  ) * pip / par;

    require(_ask <= woe()); // can't flop into surplus

    if (wad > fog() ) skr.mint(sub(wad, fog()));
    skr.push(msg.sender, wad);
    sai.pull(msg.sender, _ask);
    heal();
}
```

[V009-MD] Negative Collateral Attack

Location	Severity	Status
tub.sol	Medium	Not Fixed

Description

It is possible to create a situation where there is a negative collateral in the system. By doing this an attacker can bring the `pie()` to zero. This means an attacker can create any number of `PETH` they wish bring the `per()` very low.

Under this situation, because any number of `PETH` can be generated, the system may have a very low `per` and some rounding issues may arise.

```
Initial params: mat = 1.35, axe=1.15 pip=500, gap=tax=fee=1
hat=10000000

1. join(10ETH)
2. open()
3. lock(10ETH)
4. draw(3700)
5. Set pip = 300
6. bite()
7. bust(10ETH)
8. exit(10ETH)
9. bust(1000000ETH)  <- We can generate as many PETH as we want.
(per = 0)
10. join(1ETH) <- I generate 1 PETH without sending any `gem`...
```

Comments

This is an edge case.

One possible solution to this is to not allow the system to fully uncollauncollateralize and maintain always a small amount of `gems` in `tub`.

It is important also to apply the solution in V008-MD to limit the PETH generated in case of dilution. In the solution explained there, you can see that the number of `skr` generated are limited.

This case should also be taken in account when the rounding issues are studied, as the may become important in this specific circumstance.

2.5 Low Severity Issues

[V010-LW] ERC-20 Concern

Location	Severity	Status
DsToken.sol, WETH.sol, App.jsx	Low	Not Fixed

Description

Some functions do not currently follow the ERC-20 token standard[4]:

File	ERC-20 description	DSToken
DSToken.sol	function name() constant returns (string name)	bytes32 public name;
DSToken.sol	function symbol() constant returns (string symbol)	bytes32 public symbol;
DSToken.sol	function decimals() constant returns (uint8 decimals)	uint256 public decimals;

[4] https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20-token-standard.md

DSToken.sol	A token contract which creates new tokens SHOULD trigger a Transfer event with the _from address set to 0x0 when tokens are created.	Only DSToken.mint() event is triggered
WETH.sol	A token contract which creates new tokens SHOULD trigger a Transfer event with the _from address set to 0x0 when tokens are created.	Only WETH.Deposit() event is triggered in deposit()
WETH.sol	(implicit) totalSupply() is the sum of the balanceOf of all possible accounts	Token does not enforces the invariant since it is possible to send ether to function via contract selfdestruct.
App.jsx[5]	To prevent attack vectors clients SHOULD make sure to create user interfaces in such a way that they set the allowance first to 0 before setting it to another value for the same spender	Seems that allowance is not set to zero in code.

2.6 Additional Comments (Non-Severe Issues)

[V011-CM] Direct Token Transfer Concern

Location	Severity	Status
DSToken.sol, tub.sol, tap.sol	Comment	Not Fixed

[5] not in audit scope, but also checked

Description

There is no security mechanism in place to prevent and/or deal with people naively sending ether, or any token for that matter, directly to any contract.

Right now, when tokens are sent to those contracts the effect is either the tokens are locked or the tokens are distributed between `PETH` holders.

- `PETH` → `tub` → `PETH` is locked.
- `PETH` → `tap` → `PETH` is exchanged for `SAI` and then back to `PETH` that's burned if there is a surplus, so the effect is that `PETH` is distributed between the other `PETH`'s.
- `SAI` → `tub` → `SAI` is locked.
- `SAI` → `tap` → `SAI` is exchanged for the `PETH` that is burned, so extra `SAI` is distributed between all `PETH`'s.
- `GEM` → `tub` → `GEMs` are distributed between PETH holders
- `GEM` → `tap` → `GEM` are locked

- Sending any other token to `tap` or `tub` locks the token in the contract.

Comments

Add protections to the contracts to make any erroneous sends of ether throw, and also add a mechanism where an owner can extract foreign tokens from the contracts (see https://github.com/Giveth/giveth-common-contracts/contracts/Escapable.sol).

This issue can not be considered a bug in the system because we can not prevent user mistakes, but we have seen that implementing this functionality can save a lot of funds from being lost, especially for contracts that are recipients of tokens as in this case.

[V012-CM] Operations on Unsafe CDP

Location	Severity	Status
tub.sol	Comment	Not Fixed

Description

Currently, users can lock funds into an unsafe CDP that remains unsafe after the `draw` operation. It seems that there is no gain for the user, even more, the CDP could be `bite()`n with extra damage for the CDP owner due to bad top-up calculations.

Comments

Users operating close to the `mat` should check that the `cup` is safe after the `draw` operation in an atomic transaction if this check is not done automatically by the current UI. This will be useful because a user may not know when signing a transaction if their `cup` will be safe once that transaction gets mined. So they may want it to fail to save them the extra that would be lost in their CDP.

[V013-CM] Tradeable PETH

Location	Severity	Status
n/a	Comment	Not Fixed

Description

Since PETH follows ERC-20, it is possible to trade PETH/SAI & PETH/ETH or other combinations in secondary markets with state channels and high frequency trading capabilities.

Comments

While this issue is not independently critical, there are unexplored economic ramifications of this choice that could potentially be exploited. Since PETH price could be altered by a transaction, this will have direct impact in the markets, generating price turbulence and arbitration opportunities.

[V014-CM] System Values Concern

Location	Severity	Status
Tap.sol, top.sol, tub.sol	Comment	Not Fixed

Description

Certain system parameters could lead to systemic instability or will break the system.

```
1 <= tub.gap < 2        // The <2 is not used
0 < tap.gap < 2         // Fixed
1 <= axe                // Fixed
1 < mat                 // In the code is <=
1 <= tax                // Fixed
1 <= fee                // Fixed
```

Comments

There are other economical limitations that may be added to the contract to protect some abuses from the users governing the system.

2.7 Resolved Issues

[R002-HI] Gap Should Not Be Taken into Account Once the System Is caged

Location	Severity	Status
tap.sol, tub.sol	High	Fixed

Description

If bid() takes into account gap when off == true it incentivizes a delayed exit()

When the system is globally settled, it does not make any sense to apply the `gap` when the `exit()` is called.

Comments

This bug is fixed in the repo and does not apply to the current system because `gap=1`

[R003-MD] Dust CDP’s

Location	Severity	Status
tub.sol	Medium	Fixed

Description

Having a lot of CDP's in the system with a very small amount of collateral on each, opens some attack vectors, and also complicates the UI and the clients that need to process all these `cups`.

If a user wants to make a cup for 1 million dollars. The optimal strategy is not to create a single `cup` as this `cup` someone can call `bite()` a `cup` with a single transaction. If an attacker wants to reduce the chances of someone calling `bite()` on their `cup`, they can make 1 million `cups` for a single dollar. This will cost ~300,000 gas per `cup`. The cost to `bite()` a `cup` is ~120,000. Because an attacker can open a `cup` at a time of low network traffic they can likely use a much lower gas price. But since `bite()` is time sensitive this will likely require a higher gas price.

This also means that instead of a single transaction to `bite()` this `cup`, it will take 1,000,000 transactions. Each costing ~120,000 gas which will take ~ 15,000 blocks assuming blockGasLimit of 8,000,000. Which will take 2.6 days.

This is an extreme example and it will take more than 2.6 days to create all of these cups. But, there are two important advantages that the `cup` maker has

1. They can create these `cups` during a period of low network usage.
2. The reward for `bite()`ing so many `cups` will be significantly lower than for `bite()`ing cups with larger amounts or ether in them. Therefore the attacker can reduce the chances of their `cup` being `bite()`n.

Note: If a malicious attacker creates a lot of dust CDPs, this can overflow a web3 UI. Also the clients will have trouble processing all of these `cups` potentially increasing the sync time.

Comments

Set a requirement of `PETH` per `cup`. `PETH` should be either 0 or 0.01 `PETH`. Would help to mitigate these kind of attacks.

[R004-LW] Anybody Can shut n Empty `cup`

Location	Severity	Status
tub.sol	Low	Fixed

Description

`shut()` should be protected to be called only by the owner of the `cup`.

Comments

This could be annoying if there is an attacker that `shut()` the `cup` just after the owner `open()`d it.

[R005-CM] Drip Recursion

Location	Severity	Status
tub.sol	Comment	Fixed

Description

Currently, the `drip()` function is using recursion unnecessarily. A call to `din()` within `drip()` has to call `chi()` which in turn calls `drip()`. Since the only thing `chi()` does in this situation is return `_chi` the `din()` call can be replaced with a direct in-line calculation.

This approach may benefit from increased efficiency and has the added bonus of avoiding a recursive call.

Comments

Previous code

```
      if (tax != RAY) {  // optimised
           inc = rpow(tax, age);
           var din_ = din();
           _chi = rmul(_chi, inc);
           sai.mint(tap, sub(din(), din_));
      }
```

Tub.sol lines 190-195

Potential revisions

```
      if (tax != RAY) {  // optimised
          inc = rpow(tax, age);
          var din_ = rmul(rum, _chi); // replace din()
          _chi = rmul(_chi, inc);
          sai.mint(tap, sub(rmul(rum, _chi), din_)); // replace din()
      }
```

`Tub.sol lines 190-195`

As you can see the correct version of `_chi` is still used in both cases and several additional steps are avoided.

[R006-LW] Uncovered code

Location	Severity	Status
SaiVox DsGuard SaiTop SaiTub deployment	Low	Fixed

Description

Some parts of code are not covered by unit tests.

Comments

Having a 100% test coverage could reveal some unexpected smart contract behaviours.

3. Recommendations

3.1 Short-term

From the technical point of view, the first priority of the system should be to fix the incentivization system so that the tax and the penalty fee is sent to the `PETH` holders that really handle the risk. Fixing the incentivization system would require a full rewrite of important parts of the smart contracts. In the Comments section of issues V001-HI and V002-HI, we include a possible way to solve this. We believe that this rewrite can be easily done by the MakerDAO team, but this can take some time to finish, because it needs to go to all the steps of the development process again: definition, development, testing and auditing.

Additionally, we recommend opening the code to the scrutiny of the community. This not only means making the code open source, it is also includes commenting the code documenting the code, changing the names of the variables and the functions to more intuitive names, etc.

Before the launch, a bug bounty should be opened, and this is much more effective and easier to do if the code is less cryptic and has been open for review for a longer period of time.

It is worth mentioning again, that the security of this system greatly relies on the deployment process of the contract. MakerDAO team added a deployment script **`fab.sol`** that will significantly simplify the deployment and their verification.

Another thing that we would recommend to mitigate any unexpected security issue is to setup a time-lock and an escape hatch in the Wrapped ETH token. That would create a system that delays, for a programmable time, the conversion from WETH to ETH. And has an `escapeHatch()` function that a security guard can call in case of an emergency (a big red button). If the button is pushed, then all the ETH goes to a trusted account/multisig. This mechanism could be disabled in the future when time has proven that the system is safe. This mechanism can be understood as an alternative to the global settlement. This would help very much in unforeseeable case where there is some problem with the global settlement and it cannot be applied.

There are other improvements that may fix some of the possible issues that should be evaluated.

One of those is the limit of the price fall set by the oracles with some degraded functionality during the periods where the price is maladjusted. This would solve many issues like the risk PETH-holders might be bitten in an unexpected huge fall of the price, and possible attacks in the oracle system.

Also, think about implementing a Proxy-Oracle system (see issue V004-MD and V006-MD). That would be a federated oracle system that relies on the users' transactions and the setPrice() transactions in a way that users can not mine any transaction with a past price.

From the code development perspective, we have to recommend to have more tests (never will there be enough). Using some tools like continuous integration, code coverage or formal verification tools can also help a lot, and adding some assertion invariants into the code is always a good thing.

3.2 Long-term

We are aware that MakerDAO is working hard on the new version of the DAI. It's announced that this new version will correct some of the problems of the SAI version, it will add collateralization with multiple coins and will have a better system for closing unsafe position.

That version should not only include the improvements derived from the SAI, but also some improvements that are evolving in the smart contracts space. As an example, evaluating the possibility for DAI to become an ERC223 or ERC777 token, could be interesting.

In our opinion it is important to continue working for a more decentralized system. The oracle system or the governance system are examples of subsystems that will bring more decentralization.

Appendix A. Supplemental analysis

A.1 Decentralization

Points of centralization

- Oracles
- Governance delegates
- SAI market manipulation from whales
- Ethereum whales

Oracles

Oracles may manipulate the prices of the system. We assume that there are many oracles and they will act fairly.

Governance delegate

Certain system parameters may be updated by authorized users. This depends on how the system is deployed and configured. Once the system is deployed, the system will be transparent. What can be done and which parameters can be changed by a given role will be visible. Independent of visibility, this may became a centralization point unless this role is linked to a DAO-like governance smart contract.

SAI market manipulation from whales

A whale that holds a huge percentage of SAIs would be able to take actions that can manipulate the function of the system. For example, it can force a low price or a high price of the SAI in order to adjust `per` and then take advantage of the slow adjustment of `per`.

ETH whales

ETH whales can manipulate to a certain point the price of ETH, this can be used to take advantage of the system by buying or selling SAIs according the price they fixed. While the total supply of ETH is increased and the distribution is more sparse, these kind of attacks, become more improbable.

A.2 Economic Concerns

Unit of Account

SAI's objective is to offer a stable coin solution that could easily be used by any user in the world instead of USD. A measure of value to buy any kind of good or service. In order to project and maintain stability, SAI needs to be perceived as truthful and correctly align the right incentives between its stakeholders.

The mechanics in place should maintain the peg in all situations. To do this, there needs to be a clear policy on what to do in the case of hard forks. Ethereum is likely to undergo many hard forks in the near future, and in a scenario where two chains are believed by the market to continue after the hard fork, the SAI token is likely to be valued by the market at over $1 prior to the fork because a second SAI token on the new chain will be created and could have value. This is easily solved by communicating a policy of a global settlement for all new tokens created via hard forks. It is important that this policy be well known to the general community participating in this market, and the chain that will be maintained be decided upon well in advance of the fork.

Medium of exchange

As a medium of exchange, SAI needs to provide trust through safety and transparency to all parties involved. As a mean of payment a stable development is required to ensure a broad acceptance. In this context:

- How up and (mainly) downturns are managed, in concrete extremely fast and deep, as they trigger margin calls for all the CDPs backing the collateral?
- Safeguards and boundaries for opportunistic misbehaviors or unintentional accidents, to be clarified/developed:
    - Adaptive reaction velocity, mostly from the oracle and governance operators to adjust variable parameters (TFRM / Price sensitivity)
    - Initial analysis to set up protective static parameters (e.g. `mat`/ Liquidation ratio `cuff`/ Liquidation penalty `chop`/ Debt ceiling `cork`/ Stability fee `crop`/ Limbo duration `calm`
    - A previous timestamp for each transaction ensuring an updated and adjusted rate
- If CDPs are executed or a global settlement is triggered, what is the expectation about how this can impact on the different economic agents. Are there projections or a business case to analyze it?
- Has been considered that due to factors, as information delay, can occur a "parallel market"? Is this arbitrage opportunity a menace to keep the value generated inside the MakerDAO community?
- During an ether bull market a very good option for an investor is to leverage his position as much as possible. Although this is a private

decision, is this leverage ratio monitored to avoid instability? (We presume bots will perform this task)

Store of Value

To be an effective store of value, SAI must maintain liquidity even in unpredictable emergencies and provide assurances it will be protected against exploits. Therefore what are the incentives for an individual to invest escrowing collateral and create PETHs:

- Interest Rate over SAI generated (0% currently) What rationale is used to calculate reward parameters such as this one?
- Leverage percentage (Margin).
- If the SAI price is high, generate SAI's to arbitrate the market and return the SAI price at its stable position.
- If the SAI is low, buy SAI's in the market to return the debt at lower price.
- Collect the `gaps` in the `join()/exit()/bust()/boom()` exchanges.
- Collect the rest of the people's CDPs' penalty fee if they are liquidated
- PETH community's incentives to make the project succeed.
- Users that need SAI's will probably go to the markets first, but they can go here if prices are expensive.
- Risk that the PETH is diluted if the system gets undercollateralized.
- To avoid margin calls, does it makes sense to add a time window for borrowers when the collateral is in danger ("Anger/ Worry/ Panic/ Grief/ Dread" stages are applicable for SAI)?

In order to build up scalable soundness and reputation so that it can be recognized by a major part of the society that SAI fulfills the functions of money, it seems necessary to further clarify and discuss the risk and incentive issues remarked in this section.

A.3 Simulations

We ran the following simulations of the system:

Unsupervised CDP during 2017

using exactly the ETH progression in pip for ETH prices during 2017, hat=10M, mat=1.35, axe=1.0, tax=1.0, tapGap=1.0, tubGap=1.0, way=1.0, creating a 20 cups among all the year in equal distribution, checking the cups safety each 2h. CDR was not supervised by any bot.

Results are

```
risk   CDP killed
----- -----------
2.4   10%
2.2   10%
2.0   20%
1.8   25%
1.6   40%
1.4   70%
```

Unsupervised CDP during 15-18 2016/06 (DAO attack)

using exactly the ETH progression in pip for ETH prices during DAO attack from 2016-06-15 to 2016-06-18, hat=10M, mat=1.35, axe=1.0, tax=1.0, tapGap=1.0, tubGap=1.0, way=1.0, creating a 20 CDR among all the year in equal distribution, checking the cups safety each 10 minutes. CDR are not supervised by any bot.

During the DAO attack occurs this drop off:

```
2016-06-18 12:39:00  12.83413059
2016-06-18 12:40:00   8.97674967
2016-06-18 12:41:00  12.20483879
```

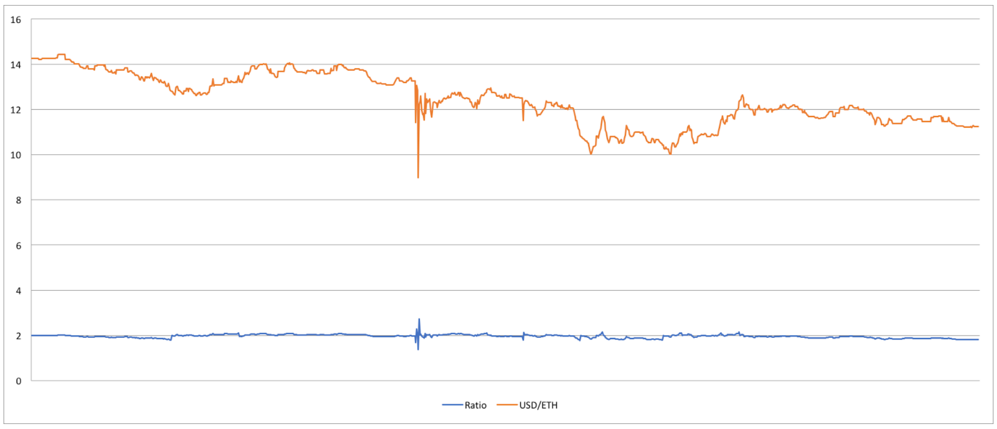
```
risk   CDP killed
----- -----------
3.4   0%
3.2   5%
3.0   15%
2.8   30%
2.6   65%
2.4   65%
2.2   75%
2.0   90%
```

Trivial bot-supervised CDP during 15-18 2016/06 (DAO attack)

Taking this one step further we developed a simple bot designed to simulate human input aimed towards maintaining cups at a safe value during the DAO crash. We used the same base-line parameters, the only difference was the automation of cup safety updates. The aim of this bot is to keep the ratio to a fixed value with a ±10% margin. With reasonable response times we found the following risk values through the critical block times:

Ratio	Max	Min	Avg
150% - 170%	217.54%	109.93%	159.58%
160% - 180%	231.13%	116.80%	169.64%
170% - 190%	244.73%	123.67%	179.50%
180% - 200%	258.33%	130.54%	189.44%
190% - 210%	271.92%	137.41%	195.96%
200% - 220%	285.52%	144.28%	207.34%
210% - 230%	299.11%	151.15%	219.79%
220% - 240%	312.71%	158.02%	229.55%

During the crash the following shows the ratio of a cup handle by a bot working with 190%-210% ratio:



This means in order to maintain safe cups through the DAO crash a user would need to be maintaining a typical risk of ~200% or better. This is based on *active participation* in the system throughout the crash with frequent adjustments.

Appendix B. Security checks

Assets

Considered as threat targets, smart contracts and user interface application logic should enforce using the underlying Ethereum technology to guarantee the self-protection and non-by-passability of the access controls for the following assets:

ETH	Token value
SAI	Token value managed by platform
PETH	Token value managed by platform
GEM / WETH	Token value managed by platform
SC Ownership	Ownership of smart contracts
SC ACL	Authorization of smart contract calls
CDP Ownership	Ownership of CDPs
Participation Incentives	Profitability among SAI/PETH users
Interoperability	Able to interoperate with other systems
Service reputation	Community (users of) SAI token reputation

Threat Agents

We use the following categories to understand who might attack the smart contracts:

Accidental Discovery	An ordinary user stumbles across a functional mistake in your application, just using a web browser, and gains access to privilege information or functionality.
Privileged account mistake	A mistake (e.g. client library failures or fat finger errors) performed from an account that have special privileges on the smart contracts.
Automated Malware	Programs or scripts, which are searching for known vulnerabilities, and then report them back to a central collection site.
The Curious Attacker	A security researcher or ordinary user, who notices something wron with the application, and decides to pursue further.

Script Kiddies	Common renegades, seeking to compromise or deface applications for collateral gain, notoriety, or a political agenda.
The Motivated Attacker	Potentially, a disgruntled staff member with inside knowledge or a paid professional attacker.
Whales	Big economic agents abusing blockchain economic models for financial gain.
Miners	Miners abusing blockchain protocols for financial gain.
Big Organizations	Highly trained professionals and criminals cracking or abusing blockchain smart contracts and protocols for financial gain, reputation destruction or geopolitical cyber warfare.

Operational Environment

We use the following operative scenarios when analyzing threats:

Normal operation	Ethereum ecosystem is stable
Network congestion	High amount of pending transactions makes a Ethereum network congestion
High volatility	ETH/$ price is having a high volatility
Fork	A contentious fork of Ethereum that produces 2 seperate networks.
Unexpected fatal bug / global settlement	An unexpected critical bug is found on the smart contracts
Subsystems failure	Subsystems (e.g. oracles)
Operators failure	Smart contract operators fails or delays the execution
Whale attacks	Economic wales gone wild

Common Security Problems Checklist

Trivial Errors

- Shadowing
- Misnamed variables
- Missing require guards
- Variable confusion
- String Literals/Magic Numbers
- Standards Conformity
- Tx.origin use
- Erroneous Constant State Functions

Race Conditions

- Re-entrancy
- Recursion
- External/Cross Contract Calls

Front Running

Miner specific attacks

- Blocktime Attacks
- Transaction Reordering Attacks

DoS

- Revert
- Block Gas Limit/Dynamic Loops

Integer Overflow & Underflow

Template Misuse/Plugin Trust Issues

Dynamic Array Memory Manipulation

ABI Length Encoding Exploit

Forced Balance Manipulation

Malicious Higher Order Bit Parameters

Economic Attacks

Unsolved Solidity Bugs

Bad function access control

Replay/nonce attacks

Library initialization

Post-deployment ACLs