# Sai

## Security Assessment

**Sai Smart Contracts**
**October 24, 2017**


Prepared For:
Andy Milenius | *DappHub*
andy@dapphub.com

Prepared By:
Josselin Feist | *Trail of Bits*
josselin@trailofbits.com

Mark Mossberg | *Trail of Bits*
mark@trailofbits.com


**Changelog**

| | |
|---|---|
| Octoboer 24, 2017: | Initial report delivered |
| December 15, 2017: | Added Appendix E with retest results |

# Table of Contents

# Executive Summary

From September 19 to October 24, DappHub engaged with Trail of Bits to conduct an assessment of the Sai system, and the Dappsys libraries and DS-Chief project. All assessed code was written in Solidity, with the exception of a small number of shell scripts. Trail of Bits conducted these assessment over the course of 8 person-weeks with two engineers.

Trail of Bits completed the assessment using manual, static, and dynamic analysis techniques over a period of five calendar weeks. The first week focused on understanding Sai at a high level through documentation and code, which was checked for common Solidity flaws. The second week focused on examining the main contracts for more nuanced errors. The third week focused on reviewing the authorization system as well as exploring mathematical exceptions as a means of functionality disruption. The last two weeks focused on investigating the possibility and implications of numerical errors in Sai and auditing DS-Chief (ea8759a0) for common Solidity errors. Overall, most of the audit resources were devoted to the Sai main contracts (SaiTub, SaiTap, SaiTop) and DS-Math, which were deemed highest priority, and received the most scrutiny. The other various libraries and DS-Chief were given lower priority.

The assessment identified a variety of issues in Sai, including issues of high severity. The most severe may lead to tokens incorrectly generated for free, trapped tokens, and denial of service. Though not directly part of Sai, for completeness, a relevant high severity design flaw of the ERC20 standard enabling token theft was also reported. Other reported issues involved various implications of errors in configuration and deployment of the system as a whole. Inconsistencies of low severity were discovered between documentation and code in both Sai and DS-Chief, which may lead to incorrect use of the contracts.

Overall, the code reviewed is of excellent quality, written with obvious awareness of current smart contract development best practices. Sai excels in the area of system design. Its interfaces are well designed and its use of patterns such as *pull* vs *push* token transfer displays maturity. The emphasis on constant time functions and simple business logic is another sign of robust Solidity code. In the area of numerical computing, a notoriously complex field, Sai can be improved. DSMath provides a solid, correct foundation for fixed point computing; however, its higher level usage in Sai requires vigilance and updates to ensure that numerical errors are anticipated and handled gracefully by the system.

## Engagement Goals

The goal of the engagement was to evaluate the security of the Sai system with specific focus on potential numerical issues enabling stolen or trapped tokens. Specifically, we sought out answers to the following questions:

- Is it possible for an attacker to steal or trap tokens?
- Is it possible to interfere with the settlement mechanism?
- Are the arithmetic calculations trustworthy?

# Project Dashboard

**Application Summary**

| | |
|---|---|
| Name | Sai |
| Version | e138cbdc |
| Type | Ethereum Smart Contract |
| Platform | Solidity |

**Engagement Summary**

| | |
|---|---|
| Dates | September 19 - October 24, 2017 |
| Method | Whitebox |
| Consultants Engaged | 2 |
| Level of Effort | 8 person-weeks |

**Vulnerability Summary**

| | | |
|---|---|---|
| Total High Severity Issues | 5 | ■■■■■ |
| Total Medium Severity Issues | 4 | ■■■■ |
| Total Low Severity Issues | 3 | ■■■ |
| Total Informational Severity Issues | 0 | |
| Total | 12 | |

**Category Breakdown**

| | | |
|---|---|---|
| Business Logic | 1 | ■ |
| Configuration | 2 | ■■ |
| Data Validation | 2 | ■■ |
| Numerics | 4 | ■■■■ |
| Undefined Behavior | 1 | ■ |
| Timing | 2 | ■■ |
| Total | 12 | |

## 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 |
| Business Logic | Related to application business logic |
| 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 |
| Numerics | Related to numeric calculations |
| 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's information is at risk, exploitation would be bad for client's reputation, moderate financial impact, possible legal implications for client |
| High | Large numbers of users, very bad for client's reputation, or serious legal or financial implications |

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

# Recommendations Summary

## Short Term

**Ensure that all the rounding issues are mitigated.** These issues may lead to free tokens or trapped users and have to be be prevented.

**Remediate all the other identified vulnerabilities.** Remove `DS-Warp`, add the proper checks for the functions parameters. Except the rounding issues, all the identified issues have straightforward solutions.

**Update and document the deployment scripts.** The Sai deployment is a key step of the system. It has to be updated and documented in sync with the code.

## Long Term

**Verify the corner cases of mathematical operations.** Ensure that mathematical corner cases, such as rounding, integer overflow/underflow, division by zero, modulo by zero, are properly handled. Consider formal methods for verification of correctness.

**Assume that mistakes will come from users and owners.** Smart contracts have a history of costly errors due to small mistakes made by users of the contracts or its owner. Assume that everyone will incorrectly use the API or the function parameters.

**Improve the documentation scope.** The documentation has to cover all the underlying assumptions, such as all the actions to perform to properly configure the system.

**Add robustness for numerical imprecision.** Numerical errors such as rounding errors are inevitable when performing fixed precision computations. Care must be taken to carefully consider when issues are likely arise, and that they are mitigated as much as possible. Mathematical code using DSMath must be written with awareness of the strengths and weaknesses of a fixed point numerical representation.

## Findings Summary

| # | Title | Type | Severity |
|---|---|---|---|
| 1 | Race condition in the ERC20 approve function may lead to token theft | Timing | High |
| 2 | Unprotected function and integer overflow may lead to system destabilization | Data Validation | High |
| 3 | Reliance on undefined behavior could unexpected behavior | Undefined Behavior | Low |
| 4 | Rounding strategy in DSMath fixed-point multiplication/division may lead to errors | Numerics | Medium |
| 5 | Misconfigured deploy may lead to unusable system | Configuration | Low |
| 6 | Inconsistent documentation on SaiTub.join() may lead to unexpected system behavior for users | Business Logic | Low |
| 7 | Race conditions during contracts deployment may lead to system compromisation | Timing | High |
| 8 | Multiple division by zero may lead to unusable system | Data Validation | Medium |
| 9 | Lack of validation on tax may lead to unusable system | Configuration | High |
| 10 | Inconsistent debt bookkeeping may lead to trapped tokens | Numerics | Medium |
| 11 | Loss of decimal precision leads to free tokens | Numerics | High |
| 12 | Loss of decimal precision leads to incomplete global settlement | Numerics | Medium |

## 1. Race condition in the ERC20 approve function may lead to token theft

Severity: High
Difficulty: High
Type: Timing
Finding ID: TOB-Sai-001
Target: DSToken and DSTokenBase

**Description**
There is a known race condition in the ERC20 standard, on the `approve` function, leading to the possible theft of tokens.

The ERC20 standard describes how to create generic token contracts. Among other, a ERC20 contract has to define these two functions:

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

The goal of these functions is to give the 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 `value` tokens of the user by calling `transferFrom(user, to, value).`

This schema is vulnerable to a race condition when the user calls `approve` a second time on an already allowed spender. If the spender sees the transaction containing the call before it has been mined, they 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, 500)`. This allows Bob to spend 500 tokens.
2. Alice changes her mind and calls `approve(Bob, 1000).` This changes the number of tokens that Bob can spend to 1000.
3. Bob sees the transaction and calls `transferFrom(Alice, X, 500)` before it has been mined.
4. If the transaction of Bob is mined before the one of Alice, `500` tokens have been transferred by Bob. But, once the transaction of Alice is mined, Bob can call `transferFrom(Alice, X, 1000)`.

Bob has transferred 1500 tokens even though this was not Alice's intention.

**Recommendation**
While this issue is known and can have a severe impact, there is no straightforward solution.

One solution is to forbid a call to `approve` if all the previous tokens are not spent by adding a `require` to approve. This solution prevents the race condition but it may result in unexpected behavior for a third party.

```
require(_approvals[msg.sender][guy] == 0)
```

Another solution is the use of a temporal mutex. Once `transferFrom` has been called for a user, it needs to prevent a call to `approve` during the limited time. The user can then verify if someone transferred the tokens. However, this solution adds complexity and may also result in unexpected behavior for a third party.

This issue is a flaw in the ERC20 design. It cannot be easily fixed without modifications to the standard and it must to be considered by developers while writing code.

## 2. Unprotected function and integer overflow may lead to system destabilization

Severity: High
Difficulty: Medium
Type: Data Validation
Finding ID: TOB-Sai-002
Target: DSWarp

**Description**
If the contracts are not properly initialized, due to the unprotected access to the `DSWarp.warp` function and an integer overflow, anyone can control the time involved in the computing the token price, computing the stability fee, and enforcing the cooldown period.

`DSWarp` is used to control the time and is inherited by many contracts. The function `DSWarp.era()` returns either:

- The value of the variable `_era`
- Or the current time (`now`)

The function `DSWarp.warp(age)` increases `_era` by `age`.
Once `DSWarp.warp(0)` is called, `DSWarp.era()` will only return the current time.

The first vulnerability is that `DSWarp.warp` is public: anyone can call it, and add any value to `_era`. Moreover, there is a possible integer overflow in `DSWarp.warp` (warp.sol:29):

```
_era = age == 0 ? 0 : _era + age;
```

As a result, the value of `_era`, can be set to any arbitrary value by anyone. This leads an attacker to be able to influence anywhere `era()` is used, including the token price computation, the stability fee computation, and global settlement.

The purpose of `DSWarp` is not clear from the documentation, we suspect the contract to be used only for testing and debugging purposes. During testing, `warp(age)` is used to increase the time, and during the deployment of the contracts in the blockchain, the function `warp(0)` should be called, leading `DSWarp.era()` to returning only the current time.

**Exploit Scenario**
The tokens are deployed, but `warp(0)` is not called. As a result, Alice can change the time value used to compute the token price. As a result, Alice is able to influence the token price valuation.

**Recommendation**
Authorization checks should be added to DSWarp, using DSAuth, to prevent `warp` from being called by untrusted users, in case of a misconfigured deployment.

Alternatively, if DSWarp solely exists to be used by tests, it should be removed, as it adds unnecessary complexity to the contracts. If the testing system through Dapp do not provide any proper way to manipulate the time, other testing frameworks, such as truffle, or pyethereum, should be considered.

The design of the smart contracts should not be influenced by the choice of the testing framework.

The documentation needs to mention all the instructions that have to be followed to properly deploy the contracts.

## 3. Reliance on undefined behavior may lead to unexpected behavior

Severity: Low
Difficulty: Low
Type: Undefined Behavior
Finding ID: TOB-Sai-003
Target: SaiTap, SaiTub and DaiVox

**Description**

Due to the use of an undefined Solidity behavior, a future version of the compiler could lead to uncompilable code or unexpected behavior.

*Undefined behavior*

Several functions are declared as `constant` functions, but they change the state of the contract; this is an undefined behavior of solidity. These functions are:

- In SaiTap: bid and ask (tap.sol:71,75)
- In SaiTub: safe and tab (tub.sol:60,175)
- In DaiVox: par and way (lib.sol:26,30)

The Solidity documentation specifies that:

```
Functions can be declared constant in which case they promise not
to modify the state.
```

A warning in the documentation points out that:

```
The compiler does not enforce yet that a constant method is not
modifying state.
```

In the current compiler version, `constant` functions can change the state of the contract; it is up to the interpretation of the ethereum client to change or not the state. For example, the solidity browser does not change the state of contracts when calling constants functions, while pyethereum does.

A future version of the compiler could enforce this property. At that time, the code will not be compilable, or a call to a constant function will not change the state of the contract.

*Example of consequences on sai*

The function `SaiTub.tab` calls `SaiTub.chi`, which calls `SaiTub.drip`, which changes the state variable `_chi`. `_chi` indicates the internal debt price. As `SaiTub.tab` is declared as a constant function, a change of compiler could lead to modification of `_chi` to not be taken into account. `SaiTub.tab` may then indicate an incorrect CDP debt, since an unupdated debt price was used in the calculation.

**Exploit Scenario**
A new version of the solidity compiler is released before the launch of the smart contract that enforces the restrictions on constant functions. This version of Solidity is used to compile and deploy the token launch contracts. As a result, all constants functions do not change the state variables, leading several values to be not be updated once these functions are called.

**Recommendation**
Remove the `constant` attribute in `SaiTap.bid`, `SaiTap.ask`, `SaiTub.safe`, `SaiTub.tab`, `DaiVox.par` and `DaiVox.way`.

Carefully review the [Solidity documentation](). In particular, any section that contains a warning must be carefully understood since it may lead to unexpected or unintentional behavior.

## 4. Rounding strategy in DSMath fixed-point multiplication/division may lead to errors

Severity: Medium
Difficulty: Medium
Type: Numeric
Finding ID: TOB-Sai-004
Target: DSMath

**Description**

In specific cases where the precise result of a fixed-point multiplication or division is exactly halfway between the smallest degree of precision accounted for, DSMath will perform "round half up" rounding to fit the result into the available number of decimals tracked. If these cases occur frequently, this will tend to bias the calculation results in the positive direction, introducing error.

Example: Consider a smart contract using DSMath which divides numbers and computes the sum of the results of the divisions. For simplicity of illustration, assume DSMath is slightly modified so the WAD type has three digits of decimal precision (1000 = 1.000). Also assume the division operations may often result in a five in the ten-thousandths place, as the last digit in the number (x.xxx5). Let the input data that the contract processes be (0.015, 6) and (0.015, 10).

*Computed with no loss of precision:*
.015/6 = .0025
.015/10 = .0015
.0025 + .0015 = **.004**

*Computed by DSMath, using "round half up" rounding*
wdiv(15, 6000) = 3 // .0025 rounded up
wdiv(15, 10000) = 2 // .0015 rounded up
3 + 2 = 5 // **.005** in decimal

Note that DSMath computes .005, while the correct result is .004. An upward bias has been introduced, due to the fact that numbers exactly halfway between the smallest degree of precision are always rounded up. As further computations occur using this biased result, the bias will propagate, creating a larger and larger divergence from the ideal result.

The severity of this issue is directly related to the likelihood of the result of a multiplication or division being exactly halfway between the smallest degree of precision. For example, for DSMath WAD, this issue will only manifest if results of multiplication/division precisely should end with a 5 in the 1e-19-th decimal place, followed by no other digits (e.g. x.xxxxxxxxxxxxxxxxxx5).

**Exploit Scenario**
Alice is the user of a smart contract which does fixed-point multiplication with DSMath for exchange rate computations and various analyses on the converted values. The contract returns erroneous results randomly, based on current exchange rates and the particular analysis in question. Alice is reliant on these analysis results for making investment decisions, and loses a significant amount of money when an incorrect analysis result led her to make a bad investment.

**Recommendation**
Instead of the “round half up” strategy, the “round half to even” strategy, also known as “banker’s rounding”, may be used. This approach eliminates bias by rounding halfway numbers up or down based on whichever results in an even value in the last digit of precision. This approach assumes an even distribution of numbers that will be rounded up and down.

Referring to the above example from the description, the computation using “round half to even” is:

*Computed using “round half to even” rounding, with three decimals of precision:*
.015/6  = .002 // .0025 rounded down
.015/10 = .002 // .0015 rounded up
.002 + .002 = **.004**

Even though precision is still lost due to rounding, the end result retains accuracy.

It is possible that the risk of this error occuring with WAD and RAY types does not justify the additional implementation complexity involved with “round half to even” rounding. Consideration should be given to the possibility of a new fixed-point type being added in the future with less decimals of precision, increasing likelihood of error.

## 5. Misconfigured deploy may lead to unusable system

Severity: Low | Difficulty: Low
Type: Configuration | Finding ID: TOB-Sai-005
Target: SaiTub, bin/deploy-live-public

**Description**

The `hat` SaiTub state variable is the system parameter controlling the Sai debt ceiling. It is of type `uint256` and never explicitly initialized, thus taking an initial value of zero. This variable is used to enforce the debt ceiling in `SaiTub.draw` (tub.sol:228), which mints Sai.

```
require(sin.totalSupply() <= hat);
```

If `hat` is not initialized, this `require` will always fail, since `sin.totalSupply()` will always be greater than zero at this point. While `hat` is uninitialized, it will be impossible for CDP users to generate Sai.

The `hat` variable can only be set via `SaiTub.mold`, which serves as the administration interface for configuring the various `SaiTub` parameters. This interface should be used in the deploy scripts to ensure that a debt ceiling is always set for the system, however it is never referenced in any of the deploy scripts in `bin/`. In `bin/deploy-live-public` there is code to configure system parameters which uses `sai cork` with the intention of setting `hat`. The `sai cork` command, however, calls the `SaiTub.cork` interface (sai-cork:8), which does not exist, so this will have no effect.

```
(set -x; seth send "${SAI_TUB?}" "cork(uint256)" "$wad")
```

Additionally, in `bin/deploy-live-public` there appear to be two other uses of non-existent configuration interfaces: `sai cuff` and `sai chop`, which call `SaiTub.cuff` and `SaiTub.chop` respectively. These will also have no effect.

The `bin/validate-deployment` script is an effective way to verify the state of a newly deployed Sai system, however the specific `hat` value it checks for (5000000) appears to be inconsistent with the value attempted to be set in `bin/deploy-live-public` (100000000).

validate-deployment:14
```
test $(sai hat)     = $(sai wad -h 5000000.0)
```

deploy-live-public:32
```
sai cork 100000000.00
```

**Exploit Scenario**
Sai is deployed using flawed deployment scripts which cause the debt ceiling parameter to never be specified. Sai users immediately begin to interact with the system, converting Ether to SKR, opening CDPs, and locking SKR into them as collateral. They attempt to draw Sai from the system, but find that they cannot and lose trust in the Sai.

**Recommendation**
In the short term, ensure that the configuration interfaces used by deployment code are consistent with those that exist in Sai.

For long term confidence in the correctness of the deployment code, use automated means of checking a deployment; the existing `validate-deployment` script is an excellent start towards this. Consider automatically invoking it at the end of the the deployment process to be aware of faulty deployment as soon as possible. Going further, it should be possible to express the parameters that the deploy system will set, in a format that can be checked for invariants (e.g. `hat` is set, and is nonzero) prior to deployment of the system.

## 6. Inconsistent documentation on SaiTub.join() may lead to unexpected system behavior for users

Severity: Low
Difficulty: Low
Type: Business Logic
Finding ID: TOB-Sai-006
Target: SaiTub

**Description**

The `SaiTub.join` function is the user interface for exchanging Ether for SKR. It takes a single parameter: the amount of SKR to receive expressed as a DSMath WAD type. It directly mints the input parameter amount of SKR to the sender, and converts that amount of SKR into Ether (GEM) and transfers that amount from the sender.

tub.sol:133

```
    function join(uint wad) note {
        require(!off);
        gem.transferFrom(msg.sender, this, ask(wad));
        skr.mint(msg.sender, wad);
    }
```

The documentation for `SaiTub.join` in the Sai README.md file, and `sai join` command line utility are inconsistent with this behavior. They document that the `join` function takes an input of the amount of GEM to use to buy SKR.

sai-join:2-3, 7

```
### sai-join -- buy SKR for gems
### Usage: sai join <amount-in-gem>
...
echo >&2 "Sending $jam GEM to TUB..."
```

Here, the comments at the top, and the tool's output document the behavior as sending an input amount of GEM to SaiTub.

README.md:167-173

```
# We need to have some GEM (W-ETH) balance to start with
$ token balance $(sai gem) $ETH_FROM
2.467935274974511817

# Join the system by exchanging some GEM (W-ETH) to SKR
```

```
$ sai join 2.2
Sending 2.200000000000000000 GEM to TUB...
$ token balance $(sai gem) $ETH_FROM
0.267935274974511817
$ token balance $(sai skr) $ETH_FROM
2.200000000000000000
```

Here, the example usage of `sai join` shows using `join` to exchange GEM for SKR, deducting the parameter to `sai join` from the initial GEM balance. This example is misleading because the user actually specified for Sai to spend 2.2 SKR *worth* of the user's GEM rather than for Sai to spend 2.2 GEM worth of SKR. The math here happens to work out such that the GEM account balance post-join is equal to the initial balance minus the input parameter to `sai join` because the GEM/SKR ratio in this example appears to be 1:1, and the bid/ask gap is also 1. This will result in a 1:1 GEM:SKR conversion because the `SaiTub.ask` converts a SKR amount to GEM by multiplying the GEM/SKR ratio (`per()`), the bid/ask gap (`gap`), and the SKR amount together.

tub.sol:126-128

```
    function ask(uint wad) constant returns (uint) {
        return rmul(wad, wmul(per(), gap));
    }
```

If the GEM/SKR ratio was 2:1, calling `join(2 ether)` would be documented as asking for 2 GEM worth of SKR (spending two of the user's GEM), but in actuality would be telling Sai to spend 2 SKR worth of GEM. Since the GEM/SKR ratio is always maintained, the user would actually spend twice as many GEM as intended.

**Exploit Scenario**

The current Sai GEM supply is 4, and the SKR supply is 2. Alice is a new Sai user wishes to exchange 2 GEM for SKR. After reading the Sai README.md file, she calls `sai join 2`, expecting to spend 2 GEM. She finds that 4 GEM has been transferred from her GEM account, and becomes confused, losing trust in Sai.

**Recommendation**

Verify interface documentation is correct for all external Sai interfaces. During development, make documentation modifications in the same commit or pull request as interface changes.

## 7. Race conditions during contracts deployment may lead to system compromise

Severity: High
Difficulty: High
Type: Timing
Finding ID: TOB-Sai-007
Target: All the contracts

**Description**

The deployment validation lacks checks, which can be used by an attacker to compromise the system.

The system relies heavily on the correct initialization of the contracts. To ensure these initializations, the script `bin/validate-deployment` performs several verifications. However, some checks are not implemented. An attacker could call the initialization functions before the deployment scripts do, to compromise the system.

For example, the authentication system (`DSAuth`) of each contract relies on the fact that the owner has their privileges removed by the script `bin/deploy-drop-auth`, however no check is performed to ensure that the privileges are dropped. An attacker could thus change the owner of the contract to another controlled address, before the call to `bin/deploy-drop-auth`, to covertly prevent loss of privileges.

*Missing checks:*

- The actions performed in `bin/deploy-drop-auth`
- The call to `tap.turn (bin/deploy:33)`
- All the calls to `setAuthority (bin/deploy:33-50)`

**Exploit Scenario**

The owner of a Sai token contract changes the owner of the contract to another controlled address, before the call to the `deploy-drop-auth` script. As the result, `deploy-drop-auth` on Sai token fails but `validate-deployment` does not emit a warning. The attacker will be authorized to call any function of the Sai token contract. As a result, the attacker can create new Sai tokens for free by calling the function `mint`.

**Recommendation**

Add the following checks to `bin/validate-deployment:`

- Check the actions performed in `bin/deploy-drop-auth`
- Check the outcome of the call to `tap.turn (bin/deploy:33)`
- Checks the authority setting `(bin/deploy:33-50)`

Sai relies on a complex initialization process. Ensure that the documentation maintains a clear description of each step to reduce future issues.

## 8. Multiple divisions by zero may lead to unusable system

Severity: Medium
Difficulty: High
Type: Data Validation
Finding ID: TOB-Sai-008
Target: SaiTub, SaiTap, SaiTop, DaiVox

**Description**
Multiple divisions by zero are possible due to a wrongly parameterized system. A division by zero leads to throwing. As a result, the system may become temporary or fully blocked.

We detail these issues in Appendix B, and we provide test cases for some of them in Appendix C.

**Exploit Scenario**
The Sai team decides to stop the tokens and call SaiTub.cage. But the function is called with 0 as `fit_` parameter. Such a mistake can easily occur due to the short address issue. As a result, any further calls to `bite` will `throw`. As a result, the users will not be able to liquidate their CDP.

**Recommendation**

- Add require(val != 0) in SaiTub.mold (tub.sol:113)
- Add require(fit_ != 0) SaiTub.cage (tub.sol:269)
- Consider the case where per() and tag() return 0 in bite (tub.sol:254)
- Consider the case where sin.totalSupply() returns 0 in cage (top.sol: 52)
- Add require(par != 0) in DaiVox (lib.sol:19)

All the functions parametrizing the contracts should check that the parameters have reasonable value. This is particularly true for the parameters that can be set only one time, as the parameters of the cage system.

## 9. Lack of validation on tax may lead to unusable system

Severity: High
Difficulty: High
Type: Configuration
Finding ID: TOB-Sai-009
Target: SaiTub, SaiTop

**Description**

`SaiTub` contains many system configuration parameters for Sai, including `tax`, which controls the stability fees Sai collects from CDPs. The `tax` value should, in practice, should always be greater than, or equal to 1, however there is no code to enforce this. This may allow `tax` to be accidentally set to a value less than 1 at some point, which can have dire consequences for the Sai system.

`SaiTub.drip` is one of the primary users of `tax`, using it for the stability fee calculation. First, `tax` is used to compute a value, `inc`, representing the increase factor of the system's `sin` debt.

tub.sol:160

```
var inc = rpow(tax, age);
```

That variable, `inc`, is then used to compute the actual amount of fees that the system collects.

tub.sol:163

```
var dew = sub(rmul(ice(), inc), ice());
```

If `tax` is less than 1, `inc` will correspondingly be less than 1. This will cause the above subtraction to trigger an exception, as there will be an integer underflow, so with this configuration, the `SaiTub.drip` function will always fail. This has several implications. First, many of the main SaiTub interfaces for CDP interaction will fail, as `SaiTub.drip` is indirectly called via the `chi` function which is called often by these interfaces. Next, the `tax` parameter will effectively become immutable after such an erroneous configuration because `SaiTub.drip` is called because setting the `tax` in `SaiTub.mold`. Lastly, the `cage` function in the `SaiTop` contract for global settlement will fail, as it calls `SaiTub.drip`. Depending on the authorization configurations of `SaiTub.cage` and `SaiTap.cage`, this may result in Sai being unable to be settled.

**Exploit Scenario**
The `SaiTub.mold` function is accidentally used to set `tax` to the RAY equivalent of 0.99. SaiTub CDP operations like `draw`, `wipe`, `free`, and `bite` immediately begin failing. A reset of the `tax` is attempted, but this fails, as does an attempt to call `SaiTop.cage` to settle the system.

**Recommendation**
In the short term, a `require` statement should be used to ensure that `tax` is never set to a value below 1. In the long term, for all sensitive system parameters with ranges of intended values, code which enforces these ranges should be added.

## 10. Inconsistent debt bookkeeping may lead to trapped tokens

Severity: Medium
Difficulty: Low
Type: Numerics
Finding ID: TOB-Sai-010
Target: SaiTub

**Description**
Note: This issue is the result of our investigation into Sai [Issue #87](#).

The Sai system uses the `sin` token to track the total CDP debt to the Sai system, as well as the individual debts for each CDP. At any point, the sum of the debts of all CDPs should be consistent with the total sin count of the system, however rounding operations can violate this invariant. This can lead to tokens being trapped in the system.

A user can generate `sai` tokens, by generating a debt in a CDP, through `SaiTub.draw.` The debt is expressed through `sin` tokens. The functions involved in the `sin` tokens manipulation are:

- `SaiTub.tab:` compute the current debt of a CDP
- `SaiTub.ice:` give the number of `sin` tokens of `SaiTub`
- `SaiTub.draw`: increase the debt of a CDP
- `SaiTub.wipe`: decrease the debt of a CDP
- `SaiTub.bite`: remove the debt of a CDP
- `SaiTub.drip:` increase `SaiTub.ice`, and influence `SaiTub.tab`

`SaiTub.ice` is expected to contain at least the sum of all the `SaiTub.tab(cup).`

`SaiTub.ice` is increased through an addition in `SaiTub.draw` and `SaiTub.drip`. In `SaiTub.drip` the addition is proportional to the debt increasing `inc`.

- `draw(wad) -> SaiTub.ice += wad`
- `drip() -> SaiTub.ice += (SaiTub.ice * inc) - SaiTub.ice`

`SaiTub.tab(cup)` is computed through the multiplication of the current debt (`cup.art`) and the price of the internal debt `chi`. The current debt is increased in `draw`. `chi` is the result of the multiplication of its previous value to the debt increasing `inc`.

- `draw(wad) -> cup.art += wad / chi`
- `drip() -> chi = chi * inc`

`SaiTub.ice` is not computed using the same mathematical logic as `SaiTub.tab`, resulting in slightly different rounding operations for the two calculations. This can cause `SaiTub.ice` to be less than the sum of all `SaiTub.tab`, which affects repayment of CDPs. It may be not possible to cancel all the debts in this situation. Indeed, the last owner of a CDP may not able `wipe` their CDP’s balance, as the subtraction of `SaiTub.tab(cup)` from `SaiTub.ice` (In `sin.burn`) will underflow. This affects the `SaiTub` `shut` and `bite` functions.

Appendix C contains a test case for this issue.

**Exploit Scenario**
The cage system is activated. Bob has a CDP with an active debt. He is the last user to call `bite` to cancel the debt of his CDP. Due to the difference in the rounding operation, there are not enough `sin` tokens in `SaiTub` for him to cancel the debt. As a result, he is not able to retrieve his remaining `skr` tokens and to cash out his money.

**Recommendation**
A quick workaround would be to:

- Change SaiTub.`mend(src, wad)` to burn the minimum between `wad` and sin.balanceOf(this)
- Change `SaiTub.bite` to push to `SaiTap` the minimum between `tab(cup)` and sin.balanceOf(this)

However, this solution does not fix the root of the issue.

To fix the possible difference, the logic used to compute the number of sin tokens mined in `SaiTub` and affected to a CDP should be changed. The same mathematical operations have to be used to prevent a difference to appear during rounding.

If two values are entangled, it is preferable to store only one and compute the second from it. If it is not possible, the relation between these values has to be proved, or at least tested.

## 11. Loss of decimal precision leads to free tokens

Severity: High
Type: Numerics
Target: SaiTub, SaiTap, Ds-math

Difficulty: Low
Finding ID: TOB-Sai-011

**Description**
The Sai system uses the fixed point decimal representation to handle fractional values. Fixed point arithmetic is known to lack precision when dealing with multiplication or division, and these operations are used to compute token prices. The resulting loss of precision allows an attacker to receive tokens for free.

Exploitation of these issues require a specific state of the system (e.g., a specific value for `chi`). Appendix C contains test cases along with each required beginning state. Note that these test cases exercise the vulnerabilities but do not exploit the full capabilities afforded by this flaw to an attacker. To demonstrate the severity of the problem, we provide a test case in Appendix D where an attacker is able to generate 0x28000000 free `skr` tokens.

**Pattern 1: Division rounding to zero**
This pattern represents `draw`, and can be exploited to generate free Sai tokens (`15 wei` worth in our example in Appendix C):

```
f(input):
 a += input / x
 b += input
```

If an attacker calls `f(user)` where the following condition is met, then `a` is not increased, while `b` is increased by `user`.

```
user / x == 0                                                    (1)
```

**Pattern 2: Division roundings**
This pattern represents `draw/wipe` and be exploited to generate free Sai tokens (`1 wei` worth in our example in Appendix C):

```
f1(input):
 a += input / x
 b += input

f2(input):
 a -= input / x
 b -= input
```

If an attacker calls `f1(user1)` followed by `f2(user2)` where the following conditions are met, then `a` ends with its initial value, while `b` is increased by `y`.

```
user2     == user1 - y                                     (1)
user1 / x == user2 / x                                     (2)
```

**Pattern 3: Multiplication rounding to zero**
This pattern represents `join` and can be exploited to generate free `skr` tokens (`1 wei` in our example in see Appendix C):

```
f(input):
 a += input * x
 b += input
```

If an attacker calls `f(user)` where the following condition is met, then `a` is not increased, while `b` is increased by `user`.

```
user * x == 0                                              (1)
```

`SaiTap.bust` may also be vulnerable to this issue.

Appendix D shows an example where a user can generate 0x28000000 free skr tokens by abusing this pattern.

**Pattern 4: Multiplication roundings**
This pattern represents `join/exit` and can be exploited to steal `gem` tokens (`1 wei` in our example in Appendix C).

```
f1(input):
 a += input * x
 b += input

f2(input):
 a -= input * x
 b -= input
```

If an attacker calls `f1(user1)` followed by `f2(user2_0)`, ..., `f2(user2_n)` where the following conditions are met, then `a` ends with its initial value, while `b` is increased by the difference in (2).

```
user1     = user2_0 + ... + user2_n                        (1)
user1 * x < user2_0 * x + ... + user2_n * x                (2)
```

`SaiTap.bust`/`SaiTap.boom` may also be vulnerable to this issue.

**Exploit Scenario**
Bob exploits certain token ratio conditions in Sai, using `join` to generate 0x28000000 free `skr` tokens. This allows Bob to do several things, including maliciously manipulate the SKR/GEM ratio, and effectively draw SAI without spending any GEM.

Alice discovers the attack and shows it publicly. As a result, users lose trust in Sai.

**Recommendation**
To prevent the pattern 1, add in `SaiTub.draw`:

```
require(div(wad, chi()) > 0)
```

To prevent the pattern 3, add in `SaiTub.join` and `SaiTap.bust`:

```
require(ask(wad) > 0)
```

A solution to prevent the pattern 2 could be to add in `SaiTub.draw`:

```
wad = rmul(rdiv(wad, chi()), chi())
```

Note that all of these recommendations require additional, thorough testing to validate the work properly. Further, we could not easily find a solution to mitigate the 4th pattern.

Fixed point computation is not well suited for multiplication and division, and requires careful consideration of corner cases. Consider using a tool based on formal methods, such as Manticore, to ensure that these issues are properly mitigated.

Recommended References:

- What causes floating point round errors? (as answered by Mark Booth)

## 12. Loss of decimal precision leads to incomplete global settlement

Severity: Medium
Type: Numerics
Target: SaiTub, SaiTap, SaiTop

Difficulty: Low
Finding ID: TOB-Sai-012

**Description**

Rounding errors can cause Sai to be unable to convert tokens in certain situations. This has been examined for converting SKR to GEM via `exit`, but we believe this also applies to converting SAI to GEM post-cage, and potentially other conversions.

As an example, consider a Sai system whose state is comprised of a single SKR holder wishing to convert to GEM. It has the following initial state:

- SKR balance (holder/total): 57571046195508407 0.048793674274572 68
- GEM balance (tub): 105983668016838502 0.599124280851040344
- gap parameter: 1.0

When the holder converts their entire SKR balance to GEM, then they should receive the entire remaining GEM balance since they are the only SKR holder. In reality, `exit` reverts due to integer underflow in the GEM DSToken and the token conversion fails. This is demonstrated in [Appendix C](#).

The `exit` operation converts a SKR balance to GEM, according to a conversion rate. This conversion process, and all conversions done by Sai, fundamentally involves the introduction of numerical rounding errors, due to the nature of fixed precision multiplication and division.

The below predicate expresses a failure scenario for `exit`, when a single holder owns all remaining SKR. *holder_skr* represents the holder's SKR balance, equal to the SKR total supply. *tub_gem* represents the GEM balance owned by SaiTub.

*holder_skr * (tub_gem / holder_skr) > sai_gem*

The value computed on the left side of the > operator is ultimately subtracted from the value on the right side. If this predicate is true, `exit` will fail because there will be an integer underflow in the subtraction. According to pure math, this predicate is false, however we found that due to aforementioned rounding errors, it can actually evaluate to true. We tested this predicate's satisfiability through targeted use of symbolic execution with [Manticore](#).

We found that ratios that induce this failure are not rare. Given SKR and GEM balances requiring the full range of precision offered by DSMath WAD types, it is easy to produce failing test cases. We found this through a simple fuzzer designed to generate large, random SKR/GEM balances and test the specific above `exit` scenario.

This particular `exit` failure scenario is mitigated by the user's control over the amount of SKR to convert; if converting an entire balance fails, it would likely be possible for them to incrementally convert nearly their entire balance. Though not thoroughly investigated, we believe this issue also applies to conversions of SAI to GEM in a post-cage system state, due to the same fundamental conversion (via error-introducing multiplication/division), and subtraction. This scenario is more severe, not only because it affects SAI holders directly, but because SAI holders do not control the amount of SAI to convert in the `cash` function. If the holder's SAI balance happens to convert to a GEM balance greater than the SaiTap's GEM balance, it will be impossible to convert any amount of their SAI to GEM.

**Exploit Scenario**
Alice is the only SKR holder, and becomes confused when she finds herself unable to convert her SKR balance to GEM in one transaction, despite calling `exit` with her exactly SKR balance. She discovers she is able to incrementally convert most of her balance with incremental conversions, but becomes distrustful of Sai.

**Recommendations**
Similar to the recommendation for TOB-Sai-010, this issue may be mitigated at the surface level by modifying the logic for `exit` (and other affected functions) to transfer the minimum between the converted value and the total available tokens. For example, in `exit`, transfer the minimum between `bid(wad)` and `gem.balanceOf(this)`.

According to our analysis, there is no single, comprehension solution that eliminates the need to carefully consider introduced numerical errors, and their potential effects. Care should be taken to avoid integer underflows which can occur easily in token conversion and transfer scenarios. Be aware of the strengths and weaknesses of fixed point computation, and if possible, prefer computational strategies favoring addition/subtraction over multiplication/division.

# A. Code Quality Recommendations

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

## General Recommendation

- Define explicitly the visibility of all functions. This would prevent mistakes in the understanding of their scope.
- Note: It is possible for a user to burn `skr` tokens for free: if a user sends some `sai` tokens to `SaiTap`, he generates fake stability fees. He can then call `SaiTap.boom` to burn its own `skr` tokens and receive back the `sai` tokens.

## SaiTub

- Use modifiers for repetitive pattern checks, such as `require(msg.sender == cups[cup].lad);` and `require(!off);`. Such design is less error prone and facilitates the review of the code
- Add to `shut` its own validation. `shut` relies on the check performed by its inside calls (`wipe` and `free`). This would prevent bugs introduced in a future refactoring of the code and improves the consistency of the functions.
- Use `DSMath.add` when incrementing `cupi` in `SaiTub.open`. Even though integer overflow is impractical in this case, using `DSMath.add` improves code uniformity, and adds safety with few drawbacks.

## SaiTap

- Add require(!off) in SaiTap.cage (tap.sol:107), to forbid an owner to set `fix` a second time

## SaiTop

- In the `cage` function, `tub.vox().par()` can be replaced with `vox.par()`.

## DSToken

- Note: the variable `decimals` is never used (DSToken.sol:23).

## DSMath

- Refactor `mul()` to not return an uninitialized uint value if the function is called with `y == 0`. Though correct behavior, this function is difficult to read due to Solidity boolean expression short circuiting, which makes it unclear what will be returned, given that `z` is not at all related to the path that executes.

```
function mul(uint x, uint y) internal returns (uint z) {
    require(y == 0 || (z = x * y) / y == x);
}
```

## Cage System

- Note: an out-of-order calls in the cage function (calling SaiTub.cage or SaiTap.cage before SaiTop.cage) leads to block the call to SaiTop.cage. In the current configuration, this attack vector is only feasible by the authority, of the contracts, which prevents it, so we do not classify it as a vulnerability.

## Deployment scripts

- Change the address 0x0 used by `deploy-drop-auth` script, as 0x0 Is a valid address. The address of the contract itself can be used instead.
- Note: SaiTap.calk does not exist anymore (deploy-config-multisig:20, validate-deployment:33). It was replaced by mold in c42d6006.

## DS-chief

- Correct the mismatches between the code and the documentation:
    - `LogLockFree` / `LogEtch` / `LogVote` / `LogLift` are not implemented
    - `vote(address[] yays)` in the doc versus `vote(address[] guys)` in the code
    - `vote(address[] yays, address lift_whom)` in the doc versus `vote(address[] guys, address lift_whom)` in the code
    - `DSChief.isUserRoot` and `DSChief.setRootUser` do not match the documentation (they do not call up `DSRoles`)
    - `DSChief.getUserRoles` and `DSChief.setUserRole` are not implemented (but they are present in `DSRoles`)
    - `vote(address[] yays, address lift_whom)` fails if `lift_whom` is not elected. As a result the vote is not taken into account. It is not the expected behavior, according the documentation.
    - Similary `vote(bytes32 slate, address lift_whom)` fails if `lift_whom` is not elected. The documentation is not clear on the expected outcome

## B. Analysis on the feasibility to call rdiv(x, 0)

The following details the analysis of the issue TOB-Sai-008. Note that we consider here only the divisions by zero that are due to a wrong parameterization of the system, we do not consider the divisions by zero that are triggered due to a direct input of the call.

**In SaiTub #1**

The call to `rdiv` in `draw` (tub.sol:224) :

```
cups[cup].art = add(cups[cup].art, rdiv(wad, chi()));
```

And to `wipe` (tub.sol:234):

```
cups[cup].art = sub(cups[cup].art, rdiv(wad, chi()));
```

Can lead to a division by zero, if `chi()` (tub.sol:147) returns 0. `chi()` can return 0 if `inc` is 0 (tub.sol:165):

```
_chi = rmul(_chi, inc);
```

`inc` can return 0 if `tax` is 0 (tub.sol:160):

```
var inc = rpow(tax, age);
```

`tax` is initialized at `RAY` and is modified by `mold` (tub.sol:113):

```
else if (param == 'tax') { drip(); tax = val; }
```

| |
|---|
| `val` should be different to 0 in `mold` (tub.sol:113) |

A test case is provided in Appendix C.

**In SaiTub #2**

The call to `rdiv` in `bite` (tub.sol:254):

```
var owe = rdiv(rmul(rmul(rue, axe), vox.par()), tag());
```

Can lead to a division by zero, if `tag()` (tub.sol:171) returns 0.

```
function tag() constant returns (uint wad) {
  return off ? fit : wmul(per(), uint(pip.read()));
```

**2a**

If `fit` is 0, `tag()` can return 0. `fit` is assigned in SaiTub.`cage` (tub.sol:269):

```
fit = fit_; // ref per skr
```

| |
|---|
| `fit` should be different to 0 in SaiTub.`cage` (tub.sol:269) |

A test case is provided in Appendix C.

**2b**

If `per()` (tub.sol:122) is 0, `tag()` can return 0. `per()` is 0 if `pie()` (tub.sol:73) is 0. `pie()` is 0 if `gem.balanceof(SaiTub)` is 0.

```
function pie() constant returns (uint) {
  return gem.balanceOf(this);
```

> *It is not clear how to avoid the case where `gem.balanceOf(this)` is zero, neither it's feasibility. Note that this would only block temporary the contract as it is possible to send a gem to SaiTub to avoid the division by zero.*

**In SaiTop #1**

The first call to `rdiv` in `cage` (top.sol: 52)

```
fix = min(rdiv(WAD, price), rdiv(tub.pie(), sin.totalSupply()));
```

Can lead to a division by zero if `sin.totalSupply()` is 0.

> *It is not clear if the case* `sin.totalSupply()` is zero *is realistic. Note that this would only block temporary the contract as it is possible to send a token to sin to avoid the division by zero.*

**In SaiTop #2**

The second call to `rdiv` in `cage` (top.sol:61):

```
cage(rdiv(uint(tub.pip().read()), vox.par()));
```

Can lead to a division by zero, if `vox.par()` (lib.sol:28) returns 0. `vox.par()` returns if `DaiVox` is called with 0:

```
function DaiVox(uint256 par) {
  _par = fix = par;
```

> `par` should be different to 0 in `DaiVox` (lib.sol:20)

Note that `SaiVox` initializes `DaiVox` with `RAY` (vox.sol:12).

```
function SaiVox() DaiVox(RAY) {
```

This prevents the division by zero in the current configuration.

**In SaiTap #1**

The call to `rdiv` in `s2s` (tap.sol:68):

```
var par = vox.par(); // ref per sai
return rdiv(tag, par); // sai per skr
```

Can lead to a division by zero, if `vox.par()` (lib.sol:28) returns 0. `vox.par()` returns if `DaiVox` is called with 0:

```
function DaiVox(uint256 par) {
```

```
        _par = fix = par;
```

| |
|---|
| par should be different to 0 in DaiVox (lib.sol:20) |

This case is similar to the division by zero in SaiTop.

## C. Test cases

### TOB-Sai-008

```
// same import as sai.t.sol

// Copy of original SaiAdmin
// the function setTaxUnprotected is added
contract SaiAdmin is DSThing {
   [..]

    // Stability fee
    // copy of setTax, without the checks
    function setTaxUnprotected(uint ray) note auth {
        tub.mold('tax', ray);
    }
}

// Copy of original SaiTestBase, except that
// tub.setOwner(0); is ignored in configureAuth
contract SaiTestAudit is DSTest, DSMath {
      [...]
    function configureAuth() {
        [...]

        // removed, to allow to call tub.cage easily
        // tub.setOwner(0);
        [..]
    }

  [...]
}

contract Audit is SaiTestAudit {
    function testDrawDiv0() {
        admin.setMat(ray(1 ether));
        tub.join(10 ether);
        var cup = tub.open();
        tub.lock(cup, 10 ether);
```

```
        // set tax (and chi()) to 0
        admin.setTaxUnprotected(0);
        warp(1 days);

        require(tub.chi() == 0);

        // trigger the division by zero
        tub.draw(cup, 1 ether);
    }
      function testCageDiv0() {
        admin.setMat(ray(1 ether));
        tub.join(10 ether);
        var cup = tub.open();
        tub.lock(cup, 10 ether);

        // set fit to 0
        tub.cage(0,0 ether);

        require(tub.fit() ==0);


        // trigger the division by zero
        tub.draw(cup, 1 ether);

      }
}
```

## TOB-Sai-010

```
function testTOBSai010(){
    gem.mint(1000 ether);
    sai.mint(100 ether); // so it can pay back stability fee
    tub.mold('hat', 1000 ether);
    tub.mold('tax', 1000000400000000000000000000);
    var cup = tub.open();
    tub.join(100 ether);
    tub.lock(cup, 100 ether);
    // draw initial amount
    tub.draw(cup, 10 ether);
    // increase chi
    warp(1 days);
    tub.drip();

    // initial values
    // _chi                            = 1.035164129205985238932488761
    // cup.art                      = 10.000000000000000000
    // sin.balanceOf(tub)  = 10.351641292059852389
    // tab(cup)                    = 10.351641292059852389
    tub.draw(cup, 4 wei);
    // cup.art                      = 10.000000000000000004
    // sin.balanceOf(tub)  = 10.351641292059852393
    // tab(cup)                    = 10.351641292059852393
    tub.draw(cup, 1 wei);
    // cup.art                      = 10.000000000000000005
    // sin.balanceOf(tub)  = 10.351641292059852394
    // tab(cup)                    = 10.351641292059852395
    // the last digit for sin(tub) is 4, and for tab(cup) is 5

    // Details of tub.draw(cup, 1 wei)
    // cup.art    = cup.art + 1/ chi = cup.art + 1
    // sin(tub)   = sin(tub) + 1
    // tab(cup)  = cup.art * chi
    // Due to the rounding, tab(cup) is added by two
    // while sin(tub) is added by one

    // this should be true
    assert(tub.sin().balanceOf(tub) >= tub.tab(cup));
}
```

TOB-Sai-011: Pattern 1

```
function testTOBSai011Pattern1(){
        gem.mint(1000 ether);

        tub.mold('tax', 1000040100000000000000000000);

        var cup = tub.open();
        tub.join(100 ether);

        // increase chi
        warp(1 days);
        tub.drip();

        assert(sai.balanceOf(this) == 0);

        tub.draw(cup, 15 wei); // create 15 sai, and 0 art
        assert(sai.balanceOf(this) >0);
        assert(tub.art(cup) > 0);
}
```

### TOB-Sai-011: Pattern 2

Note: for testing purpose, we use created the function `tub.art(cup)` which returns the `art` of a `cup`.

```
function testTOBSai011Pattern2(){
        tub.mold('tax', 1000040100000000000000000000);

        var cup = tub.open();
        tub.join(100 ether);
        // increase chi
        warp(1 days);
        tub.drip();

        tub.lock(cup, 1 ether);

        assert(sai.balanceOf(this) == 0);

        // create 21 sai, for 20 art
        tub.draw(cup, 21 wei);
        // remove 20 sai, for 20 art
        tub.wipe(cup, 20 wei);

        // no more art
        assert(tub.art(cup) == 0);
        // 1 sai left
        assert(sai.balanceOf(this) == 0);
}
```

### TOB-Sai-011: Pattern 3

```
function testTOBSai011Pattern3(){
        sin.mint(tap, 1 ether); // so the bust/flop will work

        // Get the per ratio less than .5
        var cup = tub.open();
        tub.join(1 ether);
        tub.lock(cup, 1 ether);
        tub.draw(cup, 1 ether);
        tap.bust(1.1 ether); // this mints skr and modifies per
        assert(tub.per() < ray(1 ether / 2));

        assert(gem.balanceOf(tub) == 1 ether);
        assert(skr.balanceOf(this) == 1.1 ether);

        tub.join(1 wei); // create 1 skr for 0 gem

        assert(skr.balanceOf(this) > 1.1 ether);
        assert(gem.balanceOf(tub) > 1 ether);
}
```

## TOB-Sai-011: Pattern 4

```
function testTOBSai011Pattern4(){
        // put some initial fee
        sai.mint(tap, 1 ether);

        MyFakePerson person = new MyFakePerson(tap, tub, gem, skr);
        gem.mint(person, 100 ether);
        bytes32 cup_1 = person.open();
        person.join(10 ether);
        person.lock(cup_1, 0.5 ether);
        person.draw(cup_1, 0.5 ether);
        // pay the fee
        // as a result pie() != skr.totalSupply (in per())
        person.boom(0.5 ether);

        assert(gem.balanceOf(this) == 100 ether);

        tub.join(28 wei); // cost 29 gem

        tub.exit(10 wei); // return 11 gem
        tub.exit(10 wei); // return 11 gem
        tub.exit(8 wei); // return 8 gem

        // cost 29 gem for 30 gem
        assert(gem.balanceOf(this) <= 100 ether);
}
```

### TOB-Sai-012

```
function testTOBSai012() {
        uint user0 = 575710461955084070048793674274572680;
        uint gems = 1059836680168385020599124280851040344;
        skr.mint(this, user0);
        gem.mint(tub, gems);
        tub.exit(user0); // fails
        assertEq(gem.balanceOf(tub), 0);
        assertEq(skr.balanceOf(this), 0);
}
```

## D. Manticore test case

Manticore is a dynamic binary analysis tool supporting symbolic execution of EVM bytecode. The following code triggers the third pattern from issue TOB-Sai-011 with Manticore.

Figure 1 contains the Python script to find values triggering the issue. We use a proxy function (`test_join_pattern_3(uint wad, uint wad_min, uint pie, uint skrTotalSupply)` to simulate the behavior of pattern #3 on the `join` function.

```
# import https://github.com/trailofbits/manticore/blob/0.1.5/examples/evm/seth.py
from seth import ManticoreEVM

seth = ManticoreEVM()

# Make the contract account to analyze
source_code = '''
pragma solidity ^0.4.15;

contract DSMath {
    event Log(string);
    function add(uint x, uint y) internal returns (uint z) {
        require((z = x + y) >= x);
    }
    function mul(uint x, uint y) internal returns (uint z) {
        require(y == 0 || (z = x * y) / y == x);
    }

    uint constant WAD = 10 ** 18;
    uint constant RAY = 10 ** 27;
    function rmul(uint x, uint y) internal returns (uint z) {
        z = add(mul(x, y), RAY / 2) / RAY;
    }
    function rdiv(uint x, uint y) internal returns (uint z) {
        z = add(mul(x, RAY), y / 2) / y;
    }

    // This function simulates the join function for the pattern 3
    function test_join_pattern_3(uint wad, uint wad_min, uint pie, uint skrTotalSupply){
        // limit the range of values
        require(pie > 0);
```

```
      require(pie < 10 ether);
      require(skrTotalSupply > 0);
      require(skrTotalSupply < 10 ether);

     // require the wad to be greater than a user provided value
     require(wad >= wad_min);

      // simulate join
      uint per = rdiv(pie, skrTotalSupply);
      uint ask = rmul(wad, per);

      // simulate  art += 0
      require(ask == 0);
      return ;
   }
}
'''

#Initialize user and contract
user_account = seth.create_account(balance=1000)
bytecode = seth.compile(source_code)
contract_account = seth.create_contract(owner=user_account,
                                  balance=0,
                                  init=bytecode)

# Decide what is symbolic or concrete in test_joint_pattern_3
wad = seth.SValue
wad_min = 0x28000000
pie = seth.SValue
skrTotalSupply = seth.SValue

# Generate the symbolic data
symbolic_data =
seth.make_function_call('test_join_pattern_3(uint256,uint256,uint256,uint256)',
                                  wad,
                                  wad_min,
                                  pie,
                                  skrTotalSupply)

# Generate one transaction
seth.transaction(caller=user_account,
            address=contract_account,
```

```
            value=0,
            data=symbolic_data,
           )


print "[+] There are %d alive states now"% len(seth.running_state_ids)
for state_id in seth.running_state_ids:
   seth.report(state_id)
```

Figure 1: Manticore Script

Figure 2 is the output of the run of the script in Figure 1 (the solution may differ according the valuation of the solver).

```
REPORT: STOP

LOGS:
INPUT SYMBOLS
        data_1:
b8c71c9f0000000000000000000000000000000000000000000000000000000028000000000
000000000000000000000000000000000000000000000000000000000000100000000000000
000000000000000000000000000000000000000000060200000000000000000000000000000
0000000000000000000000009b4c5e41c115138
BALANCES
       0xd30a286ec6737b8b2a6a7b5fbb5d75b895f62956L 1000
       0x1bfa530d5d685155e98cd7d9dd23f7b6a801cfefL 0
```

Figure 2: Script output

The data of Figure 2 can be split as follow:

```
b8c71c9f000000000000000000000000000000000000000000000000000000002800000000000
0000000000000000000000000000000000000000000000000000000001000000000000000000
0000000000000000000000000000000000000006020000000000000000000000000000000000
0000000000000000009b4c5e41c115138
```

Where:

- `b8c71c9f` is the function signature
- `0x28000000` is wad
- `0x1` is `wad_min`
- `0x6020000` is `pie`
- `0x9b4c5e41c115138` is `skrTotalSupply`

The Figure 3 is the corresponding  solidity test case.

```
function testTOBSai011Pattern3Large(){
      gem.mint(tub, 0x6020000);
      skr.mint(tub, 0x9b4c5e41c115138);

      assert(gem.balanceOf(this) == 100 ether);
      //generate  0x28000000 skr tokens for 0 gem
      tub.join(0x28000000);

      assert(gem.balanceOf(this) == 100 ether);
      // it fails as 0x28000000 tokens were generated
      assert(skr.balanceOf(this) == 0);
   }
```

Figure 3: Solidity test case

As a result, the user is able to generate 0x28000000 free `skr` tokens in this scenario.

## E. Fix Log

DappHub made the following modifications to their codebase as a result of this report. Each of the fixes was verified by the audit team. The code analysis is 6a2b3ac5.

| **Finding #** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Fixed** | WF | F | F | WF | F | F | F | F | F | F | M | M |

*Figure 1: Fixes summary*

*F: Fixed*, *M: Mitigated*, *WF: Won't-fix*

### Finding 1: Race condition in the ERC20 approve function may lead to token theft

**Solution:** DappHub decided to not fix the vulnerability to maintain the excepted approval semantics.

### Finding 2: Unprotected function and integer overflow may lead to system destabilization

**Solution:** DSWarp was removed.

### Finding 3: Reliance on undefined behavior could unexpected behavior

**Solution:** The inappropriate constant modifiers were removed.

### Finding 4: Rounding strategy in DSMath fixed-point multiplication/division may lead to errors

**Solution:** The Banker's rounding strategy will be considered in long-term, but will not be integrated in short-term, due to the increase of the code complexity.

### Finding 5: Misconfigured deploy may lead to unusable system

**Solution:** Deployment scripts were updated (https://github.com/makerdao/sai/pull/105).

**Notes:**

- Ensure the correct values of the script before the deployment (such as `SAI_PIT`).
- The PR 105 added this commentary in the readme.md `<!-- The oracle updates the GEM:REF price feed. This is the only external real-time input to the system. REVIEW THIS!!! -->`, it is still present.
- The readme section on the deployment is incomplete.

## Finding 6: Inconsistent documentation on SaiTub.join() may lead to unexpected system behavior for users

**Solution:** Client updated (https://github.com/makerdao/sai/pull/105)

## Finding 7: Race conditions during contracts deployment may lead to system compromisation

**Solution:** New deployment architecture based on a smart contract (https://github.com/makerdao/sai/pull/105, https://github.com/makerdao/sai/pull/112)

**Note:** We recommend to create a documentation of the initialization, to prevent future mistakes and simplify future audits. For example, the authorization schema is crucial but could be misunderstood.

## Finding 8: Multiple division by zero may lead to unusable system

**Solution:** Additional argument checks, and hard limit on risk-parameters (https://github.com/makerdao/sai/pull/107)

## Finding 9: Lack of validation on tax may lead to unusable system

**Solution:** The hard limit `tax >= 1` was added (https://github.com/makerdao/sai/pull/107)

## Finding 10: Inconsistent debt bookkeeping may lead to trapped tokens

**Solution:** Modify and relax a system invariant.

**Notes:**

- the previous system invariant was:

  *tub.ice() + tap.woe() == sin.totalSupply() == sai.totalSupply()*
  *tub.ice() == sum(tab(cdp))*

  It is replaced by:

  *tap.woe() == sin.totalSupply()*
  *tub.din() + tap.woe() =~ sai.totalSupply()*
  *tub.din() == sum(tab(cdp))*

- cap is not anymore the debt ceiling as stated in the documentation and the code commentary.

## Finding 11: Loss of decimal precision leads to free tokens

**Solutions:**

- Pattern 1 is [mitigated against free tokens](#).
- Pattern 3 is [mitigated against free tokens](#).
- Pattern 2 and 4 do not seem exploitable due to their cost.
- Additionally, `flip` is now also [protected](#).

**Notes:**

- Pattern 2 and 4: In our experiment, we were only able to generate 1 free token using pattern 2 and 4. Generating one token is not enough for an attacker to take advantage of the attack, considering the gas price. While it is unlikely that more tokens could be generated by these patterns, we were not able to prove that it is not possible.
- It may still be possible for someone to take advantage of the rounding operations to compute a better price. But this may be viewed as an investor optimisation rather than an attack.
- We recommend keeping `tax = 1` to avoid any manipulation on pattern 1 and 2.
- The value of `per()` is to be watched during the use of Sai. An unexpected value of `per` may lead to price manipulation on pattern 4.

## Finding 12: Loss of decimal precision leads to incomplete global settlement

**Solution**: The protection is let to the fronted. `cash` was replaced by `cash(wad)` to allow fronted mitigation.

**Note**: We recommend to document how a front end can properly mitigate this issue