

Smart Contract Security

Module 4

Ok, so you've finally built your dapp or smart contract.
But how do you know it was set up correctly and is safe from hackers?

The security tools and best practices will help ensure that your code is safe and follows all Ethereum development best practices— *the objective of this module.*

Content

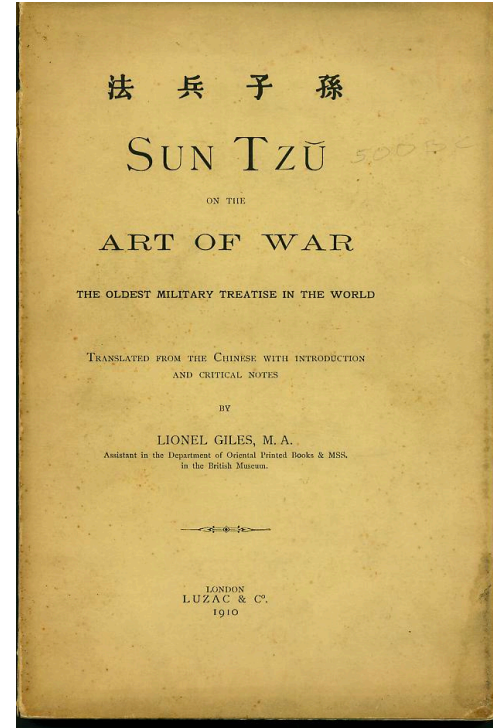
1. Stories of hacks/disasters from Ethereum
2. Known vulnerabilities/attacks
3. Security tools

Sun Tzu old rules still working!!

**One who knows the enemy and knows himself
will not be endangered in a hundred
engagements**

**One who does not know the enemy but knows
himself will sometimes be victorious**

**One who knows neither the enemy nor himself will
invariably be defeated in every engagement**



Thinking like an attacker, “The Security Mindset”

What's the worst thing that can happen? Can you make it break?

“The lack of a security mindset explains a lot of bad security out there”

https://www.schneier.com/blog/archives/2008/03/the_security_mi_1.html

Smart contracts are “**immutable**”. Once they are *deployed*, their code is impossible to change, making it impossible to fix any discovered bugs.

Code is law



1. Stories of disasters/hacks from Ethereum's first 2 years

Ethereum's timeline has been pocked by failures caused by **buggy** and **insecure** smart contracts

'\$300m in cryptocurrency' accidentally lost forever due to bug

User mistakenly takes control of hundreds of wallets containing cryptocurrency Ether, destroying them in a panic while trying to give them back

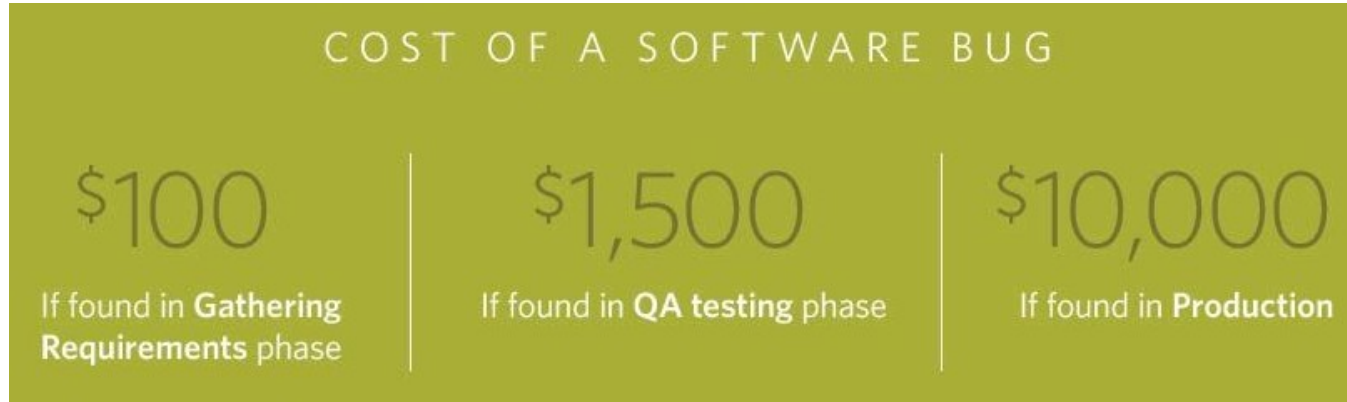
KLINT FINLEY BUSINESS 06.18.16 04:30 AM

**A \$50 MILLION HACK JUST
SHOWED THAT THE DAO WAS
ALL TOO HUMAN**

CRYPTO ENTOMOLOGY

**A coding error led to \$30 million in
ethereum being stolen**

Bugs occur in all software



A 2003 study commissioned by the Department of Commerce's National Institute of Standards and Technology found that **software bugs cost the US economy \$59.5 billion annually.**

In Smart Contracts, they're easily monetized

BOEING PLANS TO FIX THE 737 MAX JET WITH A SOFTWARE UPDATE



King of the Ether Throne hack

Post-Mortem Investigation (Feb 2016)

During the '[Turbulent Age](#)' (06 Feb 2016 to 08 Feb 2016) of the [King of the Ether Throne](#), a serious issue caused some monarch compensation payments and over/under payment refunds to fail to be sent. This web page explains the issue, the causes, the response, and the recommended solutions. It is currently in FINAL form.

<https://www.kingoftheether.com/thrones/kingoftheether/index.html>

For address.send(), Exceptions are not propagated (for default function)

Example: an exception caused by Out of Gas

Contract A:

```
function payWinnings() {  
    Winner.send();  
    selfdestruct;  
}
```

Minimum amount of gas (2300)

Contract Wallet:

```
function() { // handle payment  
    // Do more than 2300  
    // gas worth of work  
}
```

Out of gas

Result: If you played King of Ether Throne using a “Wallet Contract”, your winnings would be destroyed forever.



Causes

As with most defects, there were a number of underlying causes: (c.f. the [5 Whys](#))

1. The stipend of 2300 gas included with a payment from the KotET contract to an Ethereum Mist wallet contract was insufficient for the payment to be accepted by the wallet contract.
2. KotET contract developer was unaware that only 2300 gas included when sending payment to an address in Solidity.
3. KotET contract developer was unaware that part of a transaction could fail and roll-back without the whole transaction "chain" failing and rolling-back.
4. Insufficient real-world beta testing by KotET contract developer; testing was performed prior to launch but this did not include use of wallet-contracts to interact with the KotET contract.
5. Many Solidity example contracts (e.g. [Simple Open Auction](#), [30_endowment_retriever.sol](#)) use Solidity `<address> . send ( <amount> )` (where `<address>` is a `msg . sender`) to send payment to an address without checking return value, adding extra gas, or otherwise highlighting this issue. There is a note in the Solidity [Address](#) section that mentions the possibility of `send ( )` failing - but the example code above does not check the return value.

<https://www.kingoftheether.com/postmortem.html>

TheDAO hack

slock.it a Blockchain + IoT company



Example use case:

1. AirBnB user submits payment to the Ethereum blockchain
2. Slock Home Server (Ethereum client) receives the transaction
3. Power switch connected to Home Server receives “unlock” command, unlocks the door

Slock Home Server



Slock Power Switch



In Progress
(with partners)

- Slock Door Lock
- Slock Bike Lock
- Slock Pad Lock
- Slock Car Lock

slock.it built The DAO as a custom fundraising tool

“DAO”: Decentralized Autonomous Organization (coined by Vitalik in 2013)

Built by slock.it to raise funds for their company

Main idea: A decentralized hedge fund

Investors contribute funds, receive ownership “tokens”

Investors jointly decide how to spend funds, by voting in proportion to tokens

Many additional mechanisms:

“Splitting” to prevent hostile takeover

Reward disbursing

DAOs, Democracy and Governance

by Ralph C. Merkle, merkle@merkle.com

Version 1.9, May 31st 2016

DAO



tasks & orders



Service provider



Reward



- % fee of every transaction
- one time deployment fee

Tasks:

- Produce Slocks
- Marketing
- Partnerships



Tasks:

- Fund the development
- Vote on major decisions
- Control the funds (!)
- Profitable

Slock Home Server



supports:

- Z-Wave
- Zigbee
- Bluetooth LE

Slock Power Switch



In Progress (with partners)

- Slock Door Lock
- Slock Bike Lock
- Slock Pad Lock
- Slock Car Lock

THE DAO IS AUTONOMOUS. |

1071.36 M

DAO TOKENS CREATED

10.73 M

TOTAL ETH

116.81 M

USD EQUIVALENT



1.10

CURRENT RATE
ETH / 100 DAO TOKENS

15 hours

NEXT PRICE PHASE

11 days

LEFT
ENDS 28 MAY 09:00 GMT

Raised ~150 million dollars in ~ 1 month

Understanding The DAO Attack



David Siegel  

🕒 Jun 25, 2016 at 16:00 UTC | Updated Jun 27, 2016 at 17:52 UTC

The Hack

Unfortunately, while programmers were working on fixing this and other problems, an unknown attacker began using this approach to start draining The DAO of ether collected from the sale of its tokens.

By Saturday, 18th June, the attacker managed to drain more than [3.6m ether](#) into a "child DAO" that has the same structure as The DAO. The price of ether dropped from over \$20 to under \$13.

Excerpt from DAO contract (simplified)

Contract DAO:

```
mapping (address => int64) balances;

function withdraw(uint x) {
    if (balances[msg.sender] >= x) {
        msg.sender.call.value(balance) ();
        balances[msg.sender] -= x;
    }
}
```

Re-entrancy vulnerability in Ethereum

Ether Balance	200
balances[user]	0

Contract DAO:

```
mapping (address => int64) balances;  
  
function withdraw(uint x) {  
    if (balances[msg.sender] >= x) {  
        callee.call.value(x)();  
        balances[msg.sender] -= x;  
    }  
}
```

Balance	100
---------	-----

Wallet Contract

```
function doWithdraw() {  
    A.withdraw(100);  
}  
  
function() {  
    EventMoneyReceived(msg.value);  
}
```

Re-entrancy vulnerability in Ethereum

Ether Balance	0
balances[attacker]	-300

Contract DAO:

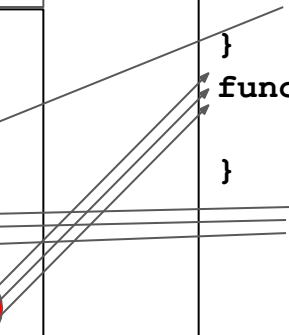
```
mapping (address => int64) balances;  
  
function withdraw(uint x) {  
    if (balances[msg.sender] >= x) {  
        callee.call.value(x) ();  
        balances[msg.sender] -= x;  
    }  
}
```



Balance	300
---------	-----

Attacker Contract

```
function startAttack() {  
    A.withdraw(100);  
}  
function() {  
    A.withdraw(100);  
}
```



Ethereum Developers Launch White Hat Counter-Attack on The DAO



Stan Higgins



🕒 Jun 21, 2016 at 20:14 UTC | Updated Jun 22, 2016 at 14:21 UTC

<https://etherscan.io/address/0xb136707642a4ea12fb4bae820f03d2562ebff487>

NEWS

Reports are emerging that members of the ethereum development community are moving funds from The DAO in attempt to stifle a new alleged attack.

Developer Alex Van de Sande, lead designer for the Ethereum Foundation, took to [Twitter](#) to announce the move, which came shortly after word emerged on social media that more funds were being [siphoned](#) from contracts associated with The DAO. He said that the action is a response to a new exploitation of The DAO's smart contract code, which comes days after millions of dollars worth of ether [were taken](#) from contracts associated with project.

The address being used by Ethereum's developers can be found [here](#), and at press time, it had amassed more than 4m ethers, worth approximately \$48m.

Timeline and Aftermath of The DAO Attack

- June 12: slock.it developers announce that a bug is found, but no funds at risk
- June 17 (Morning): attacker drains $\frac{1}{3}$ of the DAO's Ether (\$50M) over 24 hrs
Attacker's funds were trapped in a subcontract for **40 days** (July 27)
- June 17 (Evening): Eth Foundation proposes a "Soft Fork" to freeze the funds
- June 28: researchers publish a flaw in the Soft Fork Proposal
- July 15 (Morning): Eth Foundation proposes a "Hard Fork" to recover funds
- July 15 (Evening): "Ethereum Classic" manifesto published on github
- July 19: "Hard Fork" moves funds from attacker's contract to recovery contract
Ethereum Classic blockchain survives and is traded on exchanges

Both Ethereum and Ethereum Classic are both around, reached new peaks

The Parity wallet hack: Two massive failures in a row

- In July 2017, an exploitable vulnerability was exploited and \$30M

153,037 Ether (worth \$30+ million) was stolen from three wallet contracts

A remaining \$78 million worth of tokens (half the value being BAT and ICONOMI) plus 377,105+ ETH (around \$72 million) were recovered by a daring whitehat rescue, and returned to their *rightful* owners

- In November 2017, a second bug was triggered, leading to \$150M frozen funds.

Parity - The Rust Ethereum node

Parity is the Rust-Ethereum client.

Separate core development team to Ethereum Foundation.

Currently around 20% of the nodes (we think actually 6%)

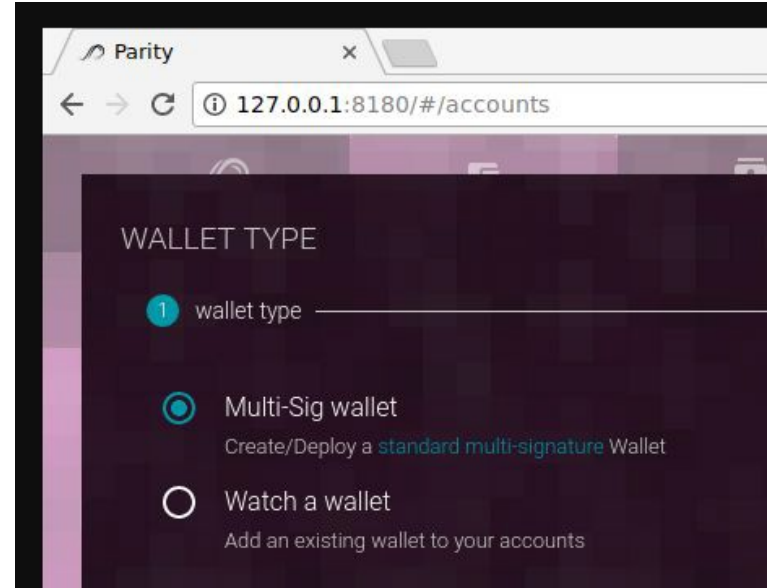
What is the Parity Wallet?

“Parity Wallet” is a Solidity smart contract that Parity can create as part of its wallet management feature.

It is a popular feature of Parity, though separate its “full node” function

<https://github.com/paritytech/parity-ethereum>

<https://paritytech.io/blog/security-alert.html>



Parity Wallet uses the “Prototype Inheritance” pattern

* Toy version of code

```
contract Wallet {  
    address walletLibrary;  
    address owner;  
  
    function Wallet(address _owner) {  
        _walletLibrary = ${hardcoded address};  
        _walletLibrary.delegatecall(  
            "initWallet(address)",  
            _owner);  
    }  
    function withdraw(uint amount)  
        returns (bool success) {  
        return _walletLibrary.delegatecall(  
            "withdraw(uint)",  
            amount);  
    }  
    // fallback function gets called if no  
    // other function matches  
    function () payable {  
        _walletLibrary.delegatecall(msg.data);  
    }  
}
```

Wallet Library contract:

```
contract WalletLibrary {  
    address owner;  
    // called by constructor  
    function initWallet(address _owner) {  
        owner = _owner;  
        // ... more setup ...  
    }  
    function changeOwner(address _new_owner)  
    function () payable {  
        // ... receive money, log events, ...  
    }  
    function withdraw(uint amount);  
}
```

[0xa657491c1e7f16adb39b9b60e87bbb8d93988bc3](#)

Multiple Wallet contracts, all refer to Wallet Library
Why? Reduces fees from duplicate data!

1. Constructor is called, invokes `initWallet`

```
contract Wallet {  
    address _walletLibrary;  
    address owner;  
  
    function Wallet(address _owner) {  
        _walletLibrary = ${hardcoded address};  
        _walletLibrary.delegatecall(  
            "initWallet(address)",  
            _owner);  
    }  
  
    function withdraw(uint amount)  
        returns (bool success) {  
        return _walletLibrary.delegatecall(  
            "withdraw(uint)",  
            amount);  
    }  
    // fallback function gets called if no  
    // other function matches  
    function () payable {  
        _walletLibrary.delegatecall(msg.data);  
    }  
}
```

Wallet Library contract:

```
contract WalletLibrary {  
    address owner;  
    // called by constructor  
    function initWallet(address _owner) {  
        owner = _owner;  
        // ... more setup ...  
    }  
  
    function changeOwner(address _new_owner)  
    function () payable {  
        // ... receive money, log events, ...  
    }  
    function withdraw(uint amount);  
}
```

[0xa657491c1e7f16adb39b9b60e87bbb8d93988bc3](#)

2. The withdraw function invokes `withdraw`

```
contract Wallet {
    address _walletLibrary;
    address owner;

    function Wallet(address _owner) {
        _walletLibrary = ${hardcoded address};
        _walletLibrary.delegatecall(
            "initWallet(address)",
            _owner);
    }

    function withdraw(uint amount)
        returns (bool success) {
        return _walletLibrary.delegatecall(
            "withdraw(uint)",
            amount);
    }

    // fallback function gets called if no
    // other function matches
    function () payable {
        _walletLibrary.delegatecall(msg.data);
    }
}
```

Wallet Library contract:

```
contract WalletLibrary {
    address owner;
    // called by constructor
    function initWallet(address _owner) {
        owner = _owner;
        // ... more setup ...
    }

    function changeOwner(address _new_owner)
    function () payable {
        // ... receive money, log events, ...
    }

    function withdraw(uint amount);
}
```

[0xa657491c1e7f16adb39b9b60e87bbb8d93988bc3](#)

3. The fallback function can invoke *any method*

```
contract Wallet {  
    address _walletLibrary;  
    address owner;  
  
    function Wallet(address _owner) {  
        _walletLibrary = ${hardcoded address};  
        _walletLibrary.delegatecall(  
            "initWallet(address)",  
            _owner);  
    }  
  
    function withdraw(uint amount)  
        returns (bool success) {  
        return _walletLibrary.delegatecall(  
            "withdraw(uint)",  
            amount);  
    }  
  
    // fallback function gets called if no  
    // other function matches  
    function () payable {  
        _walletLibrary.delegatecall(msg.data);  
    }  
}
```

Wallet Library contract:

```
contract WalletLibrary {  
    address owner;  
    // called by constructor  
    function initWallet(address _owner) {  
        owner = _owner;  
        // ... more setup ...  
    }  
  
    function changeOwner(address _new_owner)  
    function () payable {  
        // ... receive money, log events, ...  
    }  
  
    function withdraw(uint amount);  
}
```

Attacker calls:

```
victim.call("initWallet(address)", attacker)
```


Fixing the July 2017 Parity wallet bug

Old Wallet Library contract:

```
contract WalletLibrary {
    address owner;
    // called by constructor
    function initWallet(address _owner) {
        owner = _owner;
        // ... more setup ...
    }
    function changeOwner(address _new_owner)
    function () payable {
        // ... receive money, log events, ...
    }
    function withdraw(uint amount);
}
```

[0xa657491c1e7f16adb39b9b60e87bbb8d93988bc3](#)

New Library contract:

```
contract WalletLibrary {
    address owner;
    // called by constructor
    function initWallet(address _owner) {
        require(initialized == false);
        initialized = true;
        owner = _owner;
        // ... more setup ...
    }
    function changeOwner(address _new_owner)
    function () payable {
        // ... receive money, log events, ...
    }
    function withdraw(uint amount);
}
```

[0x863df6bfa4469f3ead0be8f9f2aae51c91a907b4](#)



devcon three

Ethereum Foundation Developers Conference
Nov 1-4, 2017 Cancún, Mexico



Bitcoin.com

omise

WANYANG
BLOCKCHAIN LABS

santiment

FUNFAIR



anyone can kill your contract #6995



devops199 opened this issue 22 hours ago · 12 comments



devops199 commented 22 hours ago • edited

I accidentally killed it.

<https://etherscan.io/address/0x863df6bfa4469f3ead0be8f9f2aae51c91a907b4>

Hello, first of all i'm not the owner of that contract. I was able to make myself the owner of that contract because its uninitialized.

These (<https://pastebin.com/ejakDR1f>) multi_sig wallets deployed using Parity were using the library located at "0x863df6bfa4469f3ead0be8f9f2aae51c91a907b4" address. I made myself the owner of "0x863df6bfa4469f3ead0be8f9f2aae51c91a907b4" contract and killed it and now when i query the dependent contracts "isowner(<any_addr>)" they all return TRUE because the delegate call made to a died contract.

I believe some one might exploit.

The WalletLibrary contract



anyone can kill your contract #6995

Parity wallet uses “Prototype Inheritance”

The WalletLibrary should only act as a template.

- It should not have any “state”
- Its constructor has never been called

But, the WalletLibrary is actually just a contract!

Attacker calls:

```
library.call("initWallet(address)",  
            attacker);  
library.kill();
```

Thereafter, any method call to any instance “Wallet” will fail, since the target of delegatecall is destroyed

Wallet Library contract:

```
contract WalletLibrary {  
    address owner;  
    // called by constructor  
    function initWallet(address _owner) {  
        require(initialized == false);  
        initialized = true;  
        owner = _owner;  
        // ... more setup ...  
    }  
    function changeOwner(address _new_owner)  
    function () payable {  
        // ... receive money, log events, ...  
    }  
    function withdraw(uint amount);  
}
```

[0x863df6bfa4469f3ead0be8f9f2aae51c91a907b4](#)

No one knows who devops199 was!

@devops199 you are the one that called the kill tx?

i'm eth newbie..just learning

you are famous now haha

you can see my history

 (((



Deleted user

ghost

Bug Bounty Programs

ETHEREUM Bounty Program

<https://bounty.ethereum.org/>

So, what should a good vulnerability submission look like?

Here is an example of a real issue which was previously present in the Go client:

Description: Remote Denial-of-service using non-validated blocks

Attack scenario: An attacker can send blocks that may require a high amount of computation (the maximum gasLimit) but has no proof-of-work. If the attacker sends blocks continuously, the attacker may force the victim node to 100% CPU utilization.

Impact: An attacker can abuse CPU utilization on remote nodes, possibly causing full DoS.

Components: Go client version v0.6.8

Reproduction: Send a block to a Go node that contains many txs but no valid PoW.

Details: Blocks are validated in the method `Process(Block, dontReact)`. This method performs expensive CPU-intensive tasks, such as executing transactions (`sm.ApplyDiff`) and afterward it verifies the proof-of-work (`sm.ValidateBlock()`). This allows an attacker to send blocks that may require a high amount of computation (the maximum `gasLimit`) but has no proof-of-work. If the attacker sends blocks continuously, the attacker may force the victim node to 100% CPU utilization.

Fix: Invert the order of the checks.

The EtherPot t hack

Y **Hacker News** new | threads | comments | show | ask | jobs | submit

Show HN: EtherPot – A decentralized, autonomous, provably fair lottery (etherpot.github.io)

61 points by aakilfernandes 12 days ago | flag | 25 comments

August 26, 2015

▲ aakilfernandes 12 days ago

Hey everyone, EtherPot is a smart contract on the Ethereum Blockchain. That means that no one can steal the funds or cheat to win. The lottery is provably fair.

100% of finds (except for transaction costs that go to miners) get returned to the users who play.

[reply](#)

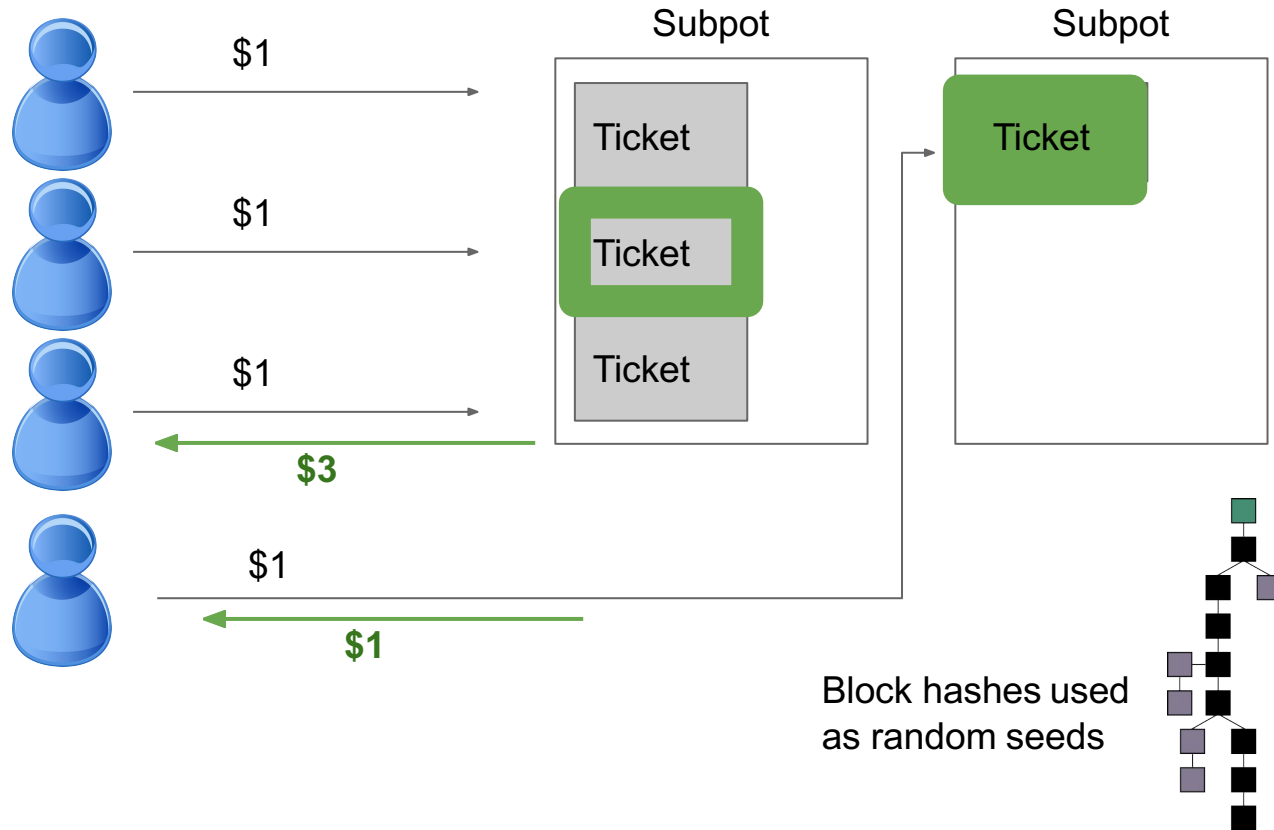
Bugs in EtherPot

Within days, hundreds of dollars of Eth paid out to the wrong recipient.

Warning: EtherPot is broken

Multiple more bugs were found.

How Etherpot Works



1. Each round lasts 1 day.
(Everything is reset after a round.)

2. Users deposit money to
purchase Tickets at a
fixed price.

Each “Subpot” holds a
fixed number of tickets.

3. After the round ends,
the next N block hashes
are used as random
seeds to determine the
winners of N subpots.
(1 block for each subpot)

```
function cash(uint roundIndex, uint subpotIndex){  
    ....  
  
    var winner = calculateWinner(roundIndex, subpotIndex);  
    var subpot = getSubpot(roundIndex);  
  
    winner.send(subpot);  
  
    rounds[roundIndex].isCashed[subpotIndex] = true;  
    //Mark the round as cashed  
}
```

Many idioms for calling functions

100 Amount in Ether
25 Amount in gas
"a1" Data argument

Solidity:

recipient.send(**100**)

recipient.call.value(**100**).gas(**25**())

recipient.foo.value(**100**)("a1")

recipient.foo("a1")

recipient.transfer(100)

Exception behavior

returns 0

returns 0

exception

exception

exception

Gas sent

minimum

25

all of it

all of it

minimum

Serpent:

send(recipient, **100**)

send(**25**, recipient, **100**)

recipient.foo(55)

recipient.foo(55, gas=**25**)

returns 0

returns 0

returns 0

returns 0

all of it

25

all of it

25

Call stack hazard:

Maximum call stack depth is 1023

Suppose we call A.recurse(0). Does Alice get 100?

Contract A:

```
function recurse(int i) {  
  if (i == 1022)  
    return B.b() + "World";  
  else recurse(i+1);  
  return OK;  
}
```

Stack depth = 1023

Contract B:

```
function b() {  
  C.send(100);  
  return "Hello ";  
}
```

Contract C:

```
function() {  
  Alice.send(100);  
}
```

A.recurse(0)

Stack depth = 0

A.recurse(1)

Stack depth = 1

....

A.recurse(1022)

Stack depth = 1022

Returns "Hello World!"

The Callstack hazard in Etherpot

Attack Contract:

```
function recurse(int i) {  
    if (i == 1022)  
        Etherpot.cash(r,idx)  
    else recurse(i+1);  
    return OK;  
}
```

```
function cash(uint roundIndex, uint subpotIndex){
```

```
....
```

```
var winner = calculateWinner(roundIndex,subpotIndex);  
var subpot = getSubpot(roundIndex);
```

```
winner.send(subpot);
```

```
rounds[roundIndex].isCashed[subpotIndex] = true;  
//Mark the round as cashed
```

```
}
```

Result: *attacker can destroy all the funds in the contract*

Converting bytes32 to int always returning 0 #34



Closed aakilfernandes opened this issue on Aug 27, 2015 · 5 comments



aakilfernandes commented on Aug 27, 2015



Not sure if I this is a bug or I misunderstand how it should work, but the following function returns 0 for all blocks

```
function getHashOfBlock(uint blockIndex) constant returns(uint) {  
  
    return uint(block.blockhash(blockIndex));  
}
```



LianaHus commented on Aug 28, 2015

Contributor



<http://gavwood.com/paper.pdf> page 25.

"0 is left on the stack if the looked for block number is greater than the current block number or more than 256 blocks behind the current block"

Please check if this is your case. I think this is not because of conversion but because the blockhash returns 0.

2. Known vulnerabilities/attacks

Attacks which you should be aware of, and defend against when writing smart contracts.

List of known security vulnerabilities/pitfalls

Integer Overflow and Underflow

Reentrancy

Timestamp Dependence

DoS with (Unexpected) revert

DoS with Block Gas Limit

Insufficient gas griefing

Forcibly Sending Ether to a Contract

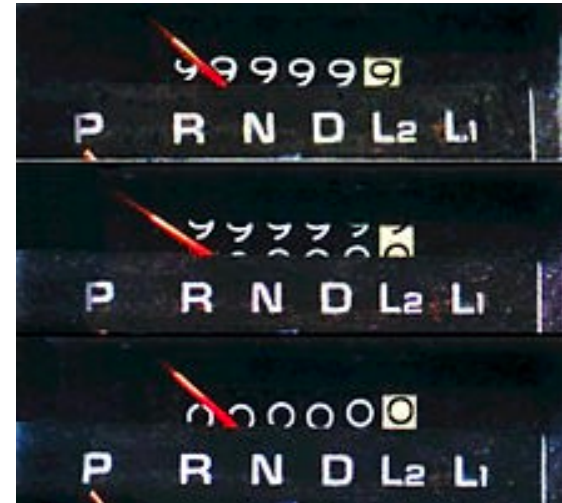
Other Vulnerabilities ([Smart Contract Weakness Classification](#)

[Registry](#) offers a complete and up-to-date catalogue of known smart contract vulnerabilities and anti-patterns along with real-world examples)

Overflows & Underflows

An **overflow** is when a number gets incremented above its maximum value. Solidity can handle up to 256 bit numbers (up to $2^{256}-1$), so incrementing by 1 would result into 0.

Likewise, in the inverse case, **when the number is unsigned**, decrementing will underflow the number, resulting in the maximum possible value.



After reaching the maximum reading, an odometer or trip meter restarts from zero, called odometer rollover.

```
pragma solidity 0.5.1;

// Contract to test unsigned integer underflows and overflows
// note: uint in solidity is an alias for uint256

// Guidelines: Press "deploy" to the right, then check the values of max and zero by clicking
// Then, call overflow and underflow and check the values of max and zero again

contract OverflowUnderFlow {
    uint public zero = 0;
    uint public max = 2**256-1;

    // zero will end up at 2**256-1
    function underflow() public {
        zero -= 1;
    }
    // max will end up at 0
    function overflow() public {
        max += 1;
    }
}
```



Download: <https://gist.github.com/rparizi1/0285d5b27a28a0898a19fc6f5e4e14b8>

Consider a simple token transfer:

```
mapping (address => uint256) public balanceOf;
```

```
// INSECURE
```

```
function transfer(address _to, uint256 _value) {  
    /* Check if sender has balance */  
    require(balanceOf[msg.sender] >= _value);  
    /* Add and subtract new balances */  
    balanceOf[msg.sender] -= _value;  
    balanceOf[_to] += _value;  
}
```

```
// SECURE
```

```
function transfer(address _to, uint256 _value) {  
    /* Check if sender has balance and for overflows */  
    require(balanceOf[msg.sender] >= _value && balanceOf[_to] + _value >= balanceOf[_to]);  
  
    /* Add and subtract new balances */  
    balanceOf[msg.sender] -= _value;  
    balanceOf[_to] += _value;  
}
```

- If a balance reaches the *maximum uint value* (2^{256}) it will circle back to zero which checks for the condition.
- Think about whether or not the uint value has an opportunity to approach such a large number. Think about how the uint variable changes state, and who has authority to make such changes.
- The same is true for underflow. If a uint is made to be less than zero, it will cause an underflow and get set to its maximum value.

→ Be careful with the smaller data-types like uint8, uint16, uint24...etc: they can even more easily hit their maximum value.

Mitigation?

One simple solution to mitigate the common mistakes for overflow and underflow is to use SafeMath.sol [library](#) for arithmetic functions (powered by OpenZeppelin)

You can test the fixed bug for first example here: [download](#)

Reentrancy

Race To Empty is the Real Deal

Use caution when making external calls



Reentrancy

One of the major dangers of calling external contracts is that they can take over the control flow, and make changes to your data that the calling function wasn't expecting.

This class of bug can take many forms, and both of the major bugs that led to the **DAO's** collapse were bugs of this sort.

Example: Reentrancy on a Single Function

```
mapping (address => uint) private userBalances;  
  
function withdrawBalance() public {  
    uint amountToWithdraw = userBalances[msg.sender];  
    require(msg.sender.call.value(amountToWithdraw()));  
    userBalances[msg.sender] = 0;  
}
```



Since the user's balance is not set to 0 until the very end of the function, the second (and later) invocations will still succeed, and will withdraw the balance over and over again.

At this point, the caller's code is executed, and can call withdrawBalance again

Mitigation?

the best mitigation method is to use `send()` instead of `call.value()`. This will limit any external code from being executed.

Be aware of the tradeoffs between `send()`, `transfer()`, and `call.value()`

```
mapping (address => uint) private userBalances;  
  
function withdrawBalance() public {  
    uint amountToWithdraw = userBalances[msg.sender];  
    userBalances[msg.sender] = 0;  
    require(msg.sender.call.value(amountToWithdraw)());  
}
```



The user's balance is already 0, so future invocations won't withdraw anything

Be aware of the tradeoffs between `send()`, `transfer()`, and `call.value()`

When sending ether be aware of the relative tradeoffs between the use of `someAddress.send()`, `someAddress.transfer()`, and `someAddress.call.value()`

- `someAddress.send()` and `someAddress.transfer()` are considered *safe* against [reentrancy](#). While these methods still trigger code execution, the called contract is only given a stipend of 2,300 gas which is currently only enough to log an event.
- `x.transfer(y)` is equivalent to `require(x.send(y))`; it will automatically revert if the send fails.
- `someAddress.call.value(y)()` will send the provided ether and trigger code execution. The executed code is given all available gas for execution making this type of value transfer *unsafe* against reentrancy.

Example: Cross-function Reentrancy



```
mapping (address => uint) private userBalances;

function transfer(address to, uint amount) {
    if (userBalances[msg.sender] >= amount) {
        userBalances[to] += amount;
        userBalances[msg.sender] -= amount;
    }
}

function withdrawBalance() public {
    uint amountToWithdraw = userBalances[msg.sender];
    require(msg.sender.call.value(amountToWithdraw)());
    userBalances[msg.sender] = 0;
}
```

In this case, the attacker calls transfer() when their code is executed on the external call in withdrawBalance. Since their balance has not yet been set to 0, they are able to transfer the tokens even though they already received the withdrawal. This vulnerability was also used in the DAO attack.

At this point, the caller's code is executed, and can call transfer()

2. Security tools

Static and Dynamic Analysis tools

[MythX Plugin for Truffle](#) - Security verification plugin for Truffle.

[Sabre](#) - Easy-to-use MythX security analyzer written in JavaScript.

[PythX](#) - MythX Python library and CLI tool.

[Mythril Classic](#) - Swiss army knife for smart contract security.

[Slither](#) - Static analysis framework with detectors for many common Solidity issues. It has taint and value tracking capabilities and is written in Python.

[Echidna](#) - The only available fuzzer for Ethereum software. Uses property testing to generate malicious inputs that break smart contracts.

[Manticore](#) - Dynamic binary analysis tool with [EVM support](#).

[Oyente](#) - Analyze Ethereum code to find common vulnerabilities, based on this [paper](#).

[Securify](#) - Fully automated online static analyzer for smart contracts, providing a security report based on vulnerability patterns.

[SmartCheck](#) - Static analysis of Solidity source code for security vulnerabilities and best practices.

[Octopus](#) - Security Analysis tool for Blockchain Smart Contracts with support of EVM and (e)WASM.

Visualization tools

[Sūrya](#) - Utility tool for smart contract systems, offering a number of visual outputs and information about the contracts' structure. Also supports querying the function call graph.

[Solgraph](#) - Generates a DOT graph that visualizes function control flow of a Solidity contract and highlights potential security vulnerabilities.

[EVM Lab](#) - Rich tool package to interact with the EVM. Includes a VM, Ethereum API, and a trace-viewer.

[ethereum-graph-debugger](#) - A graphical EVM debugger. Displays the entire program control flow graph.

Reading/resource:

Ethereum Smart Contract Security Best Practices

<https://consensys.net/developers/#security>