This is the very first iteration of the ♦ **Decentralized Application Security Project** (or **DASP**) **Top 10** of **2018**

This project is an initiative of NCC Group. It is an open and collaborative project to join efforts in discovering smart contract vulnerabilities within the security community. To get involved, join the github page.

♻ 1. Reentrancy

also known as or related to **race to empty**, **recursive call vulnerability**, **call to the unknown**

> This exploit was missed in review so many times by so many different people: reviewers tend to review functions one at a time, and assume that calls to secure subroutines will operate securely and as intended.
>
> — Phil Daian

The Reentrancy attack, probably the most famous Ethereum vulnerability, surprised everyone when discovered for the first time. It was first unveiled during a multimillion dollar heist which led to a hard fork of Ethereum. Reentrancy occurs when external contract calls are allowed to make new calls to the calling contract before the initial execution is complete. For a function, this means that the contract state may change in the middle of its execution as a result of a call to an untrusted contract or the use of a low level function with an external address.

Loss: estimated at 3.5M ETH (~50M USD at the time)

Timeline of discovery:

Date	Event
Jun 5, 2016	Christian Reitwiessner discovers an antipattern in solidity
Jun 9, 2016	More Ethereum Attacks: Race-To-Empty is the Real Deal (vessenes.com)
Jun 12, 2016	No DAO funds at risk following the Ethereum smart contract 'recursive call' bug discovery (blog.slock.it)
Jun 17, 2016	I think TheDAO is getting drained right now (reddit.com)
Aug 24, 2016	The History of the DAO and Lessons Learned (blog.slock.it)

Real World Impact:

- The DAO

Example:

1. A **smart contract** tracks the balance of a number of external addresses and allows users to retrieve funds with its public `withdraw()` function.
2. A **malicious smart contract** uses the `withdraw()` function to retrieve its entire balance.
3. The **victim contract** executes the `call.value(amount)()` **low level function** to send the ether to the **malicious contract** **before updating the balance of the malicious contract**.
4. The **malicious contract** has a payable `fallback()` function that accepts the funds and then calls back into the **victim contract**'s `withdraw()` function.
5. This second execution triggers a transfer of funds: remember, the balance of the **malicious contract** still hasn't been updated from the first withdrawal. As a result, the **malicious contract** successfully withdraws its entire balance a second time.

Code Example:

The following function contains a function vulnerable to a reentrancy attack. When the low level `call()` function sends ether to the `msg.sender` address, it becomes vulnerable; if the address is a smart contract, the payment will trigger its fallback function with what's left of the transaction gas:

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

Additional Resources:

- The DAO smart contract
- Analysis of the DAO exploit
- Simple DAO code example
- Reentrancy code example
- How Someone Tried to Exploit a Flaw in Our Smart Contract and Steal All of Its Ether

2. Access Control

It was possible to turn the Parity Wallet library contract into a regular multi-sig wallet and become an owner of it by calling the initWallet function.

— Parity

Access Control issues are common in all programs, not just smart contracts. In fact, it's number 5 on the OWASP top 10. One usually accesses a contract's functionality through its public or external functions. While insecure **visibility** settings give attackers straightforward ways to access a contract's private values or logic, access control bypasses are sometimes more subtle. These vulnerabilities can occur when contracts use the deprecated `tx.origin` to validate callers, handle large authorization logic with lengthy `require` and make reckless use of `delegatecall` in proxy libraries or proxy contracts.

Loss: estimated at 150,000 ETH (~30M USD at the time)

Real World Impact:

- Parity Multi-sig bug 1
- Parity Multi-sig bug 2
- Rubixi

Example:

1. A **smart contract** designates the address which initializes it as the contract's owner. This is a common pattern for granting special privileges such as the ability to withdraw the contract's funds.
2. Unfortunately, the initialization function can be called by anyone — even after it has already been called. Allowing anyone to become the owner of the contract and take its funds.

Code Example:

In the following example, the contract's **initialization function** sets the caller of the function as its owner. However, the logic is detached from the contract's constructor, and it does not keep track of the fact that it has already been called.

```
function initContract() public {
        owner = msg.sender;
}
```

In the Parity multi-sig wallet, this initialization function was detached from the wallets themselves and defined in a "library" contract. Users were expected to initialize their own wallet by calling the library's function via a `delegateCall`. Unfortunately, as in our example, the function did not check if the wallet had already been initialized. Worse, since the library was a smart contract, anyone could initialize the library itself and call for its destruction.

Additional Resources:

- Fix for Parity multi-sig wallet bug 1
- Parity security alert 2
- On the Parity wallet multi-sig hack
- Unprotected function
- Rubixi's smart contract

3. Arithmetic Issues

also known as **integer overflow** and **integer underflow**

An overflow condition gives incorrect results and, particularly if the possibility has not been anticipated, can compromise a program's reliability and security.

— Jules Dourlens

Integer overflows and underflows are not a new class of vulnerability, but they are especially dangerous in smart contracts, where unsigned integers are prevalent and most developers are used to simple `int` types (which are often just signed integers). If overflows occur, many benign-seeming codepaths become vectors for theft or denial of service.

Real World Impact:

- The DAO
- BatchOverflow (multiple tokens)
- ProxyOverflow (multiple tokens)

Example:

1. A **smart contract**'s `withdraw()` function allows you to retrieve ether donated to the contract as long as your balance remains positive after the operation.
2. An **attacker** attempts to withdraw more than his or her current balance.
3. The `withdraw()` function check's result is always a positive amount, allowing the attacker to withdraw more than allowed. The resulting balance underflows and becomes an order of magnitude larger than it should be.

Code Example:

The most straightforward example is a function that does not check for integer underflow, allowing you to withdraw an infinite amount of tokens:

```
function withdraw(uint _amount) {
        require(balances[msg.sender] - _amount > 0);
        msg.sender.transfer(_amount);
        balances[msg.sender] -= _amount;
}
```

The second example (spotted during the Underhanded Solidity Coding Contest) is an off-by-one error facilitated by the fact that an array's length is represented by an unsigned integer:

```
function popArrayOfThings() {
        require(arrayOfThings.length >= 0);
        arrayOfThings.length--;
}
```

The third example is a variant of the first example, where the result of arithmetic on two unsigned integers is an unsigned integer:

```
function votes(uint postId, uint upvote, uint downvotes) {
        if (upvote - downvote < 0) {
                deletePost(postId)
        }
}
```

The fourth example features the soon-to-be-deprecated `var` keyword. Because `var` will change itself to the smallest type needed to contain the assigned value, it will become an `uint8` to hold the value 0. If the loop is meant to iterate more than 255 times, it will never reach that number and will stop when the execution runs out of gas:

```
for (var i = 0; i < somethingLarge; i ++) {
        // ...
}
```

Additional Resources:

- SafeMath to protect from overflows
- Integer overflow code example

✔ 4. Unchecked Return Values For Low Level Calls

also known as or related to **silent failing sends**, **unchecked-send**

The use of low level "call" should be avoided whenever possible. It can lead to unexpected behavior if return values are not handled properly.

— Remix

One of the deeper features of Solidity are the low level functions `call()`, `callcode()`, `delegatecall()` and `send()`. Their behavior in accounting for errors is quite different from other Solidity functions, as they will not propagate (or bubble up) and will not lead to a total reversion of the current execution. Instead, they will return a boolean value set to `false`, and the code will continue to run. This can surprise developers and, if the return value of such low-level calls are not checked, can lead to fail-opens and other unwanted outcomes. Remember, **send can fail!**

Real World Impact:

- King of the Ether
- Etherpot

Code Example:

The following code is an example of what can go wrong when one forgets to check the return value of `send()`. If the call is used to send ether to a smart contract that does not accept them (e.g. because it does not have a **payable** fallback function), the EVM will replace its return value with `false`. Since the return value is not checked in our example, the function's changes to the contract state will not be reverted, and the `etherLeft` variable will end up tracking an incorrect value:

```
function withdraw(uint256 _amount) public {
        require(balances[msg.sender] >= _amount);
        balances[msg.sender] -= _amount;
        etherLeft -= _amount;
        msg.sender.send(_amount);
}
```

Additional Resources:

- Unchecked external call
- Scanning Live Ethereum Contracts for the "Unchecked-Send" Bug

🚫 5. Denial of Service

including **gas limit reached**, **unexpected throw**, **unexpected kill**, **access control breached**

I accidentally killed it.

— devops199 on the Parity multi-sig wallet

Denial of service is deadly in the world of Ethereum: while other types of applications can eventually recover, smart contracts can be taken offline forever by just one of these attacks. Many ways lead to denials of service, including maliciously behaving when being the recipient of a transaction, artificially increasing the gas necessary to compute a function, abusing access controls to access private components of smart contracts, taking advantage of mixups and negligence, etc. This class of attack includes many different variants and will probably see a lot of development in the years to come.

Loss: estimated at 514,874 ETH (~300M USD at the time)

Real World Impact:

- GovernMental
- Parity Multi-sig wallet

Example:

1. An **auction contract** allows its users to bid on different assets.
2. To bid, a user must call a `bid(uint object)` function with the desired amount of ether. The auction contract will store the ether in **escrow** until the object's owner accepts the bid or the initial bidder cancels it. This means that the auction contract must hold the full value of any unresolved bid in its balance.
3. The **auction contract** also contains a `withdraw(uint amount)` function which allows admins to retrieve funds from the contract. As the function sends the `amount` to a hardcoded address, the developers have decided to make the function public.
4. An **attacker** sees a potential attack and calls the function, directing all the **contract**'s funds to its admins. This destroys the promise of escrow and blocks all the pending bids.
5. While the admins might return the escrowed money to the contract, the **attacker** can continue the attack by simply withdrawing the funds again.

Code Example:

In the following example (inspired by King of the Ether) a function of a game contract allows you to become the president if you publicly bribe the previous one. Unfortunately, if the previous president is a smart contract and causes reversion on payment, the transfer of power will fail and the malicious smart contract will remain president forever. Sounds like a dictatorship to me:

```
function becomePresident() payable {
    require(msg.value >= price); // must pay the price to become president
    president.transfer(price);   // we pay the previous president
    president = msg.sender;      // we crown the new president
    price = price * 2;           // we double the price to become president
}
```

In this second example, a caller can decide who the next function call will reward. Because of the expensive instructions in the `for` loop, an attacker can introduce a number too large to iterate on (due to gas block limitations in Ethereum) which will effectively block the function from functioning.

```
function selectNextWinners(uint256 _largestWinner) {
        for(uint256 i = 0; i < largestWinner, i++) {
                // heavy code
        }
        largestWinner = _largestWinner;
}
```

Additional Resources:

- Parity Multisig Hacked. Again
- Statement on the Parity multi-sig wallet vulnerability and the Cappasity token crowdsale

👆 6. Bad Randomness

also known as **nothing is secret**

The contract had insufficient validation of the block.number age, which resulted in 400 ETH being lost to an unknown player who waited for 256 blocks before revealing the predictable winning number.

— Arseny Reutov

Randomness is hard to get right in Ethereum. While Solidity offers functions and variables that can access apparently hard-to-predict values, they are generally either more public than they seem or subject to miners' influence. Because these sources of randomness are to an extent predictable, malicious users can generally replicate it and attack the function relying on its unpredictablility.

Loss: more than 400 ETH

Real World Impact:

- SmartBillions Lottery
- TheRun

Example:

1. A **smart contract** uses the block number as a source of randomness for a game.
2. An attacker creates a **malicious contract** that checks if the current block number is a winner. If so, it calls the first **smart contract** in order to win; since the call will be part of the same transaction, the block number will remain the same on both contracts.

3. The attacker only has to call her **malicious contract** until it wins.

Code Example:

In this first example, a `private seed` is used in combination with an `iteration` number and the `keccak256` hash function to determine if the caller wins. Even though the `seed` is `private`, it must have been set via a transaction at some point in time and thus is visible on the blockchain.

```
uint256 private seed;

function play() public payable {
        require(msg.value >= 1 ether);
        iteration++;
        uint randomNumber = uint(keccak256(seed + iteration));
        if (randomNumber % 2 == 0) {
                msg.sender.transfer(this.balance);
        }
}
```

In this second example, `block.blockhash` is being used to generate a random number. This hash is unknown if the `blockNumber` is set to the current `block.number` (for obvious reasons), and is thus set to `0`. In the case where the `blockNumber` is set to more than 256 blocks in the past, it will always be zero. Finally, if it is set to a previous block number that is not too old, another smart contract can access the same number and call the game contract as part of the same transaction.

```
function play() public payable {
        require(msg.value >= 1 ether);
        if (block.blockhash(blockNumber) % 2 == 0) {
                msg.sender.transfer(this.balance);
        }
}
```

Additional Resources:

- Predicting Random Numbers in Ethereum Smart Contracts
- Random in Ethereum

7. Front-Running

also known as **time-of-check vs time-of-use (TOCTOU)**, **race condition**, **transaction ordering dependence (TOD)**

Turns out, all it takes is about 150 lines of Python to get a working front-running algorithm.

— Ivan Bogatyy

Since miners always get rewarded via gas fees for running code on behalf of externally owned addresses (EOA), users can specify higher fees to have their transactions mined more quickly. Since the Ethereum blockchain is public, everyone can see the contents of others' pending transactions. This means if a given user is revealing the solution to a puzzle or other valuable secret, a malicious user can steal the solution and copy their transaction with higher fees to preempt the original solution. If developers of smart contracts are not careful, this situation can lead to practical and devastating front-running attacks.

Real World Impact:

- Bancor

- ERC-20
- TheRun

Example:

1. A **smart contract** publishes an RSA number (`N = prime1 x prime2`).
2. A call to its `submitSolution()` public function with the right `prime1` and `prime2` rewards the caller.
3. Alice successfuly factors the RSA number and submits a solution.
4. **Someone** on the network sees Alice's transaction (containing the solution) waiting to be mined and submits it with a higher gas price.
5. The second transaction gets picked up first by miners due to the higher paid fee. The **attacker** wins the prize.

Additional Resources:

- Predicting random numbers in Ethereum smart contracts
- Front-running, Griefing and the Perils of Virtual Settlement
- Frontrunning Bancor

8. Time manipulation

also known as **timestamp dependence**

If a miner holds a stake on a contract, he could gain an advantage by choosing a suitable timestamp for a block he is mining.

— Nicola Atzei, Massimo Bartoletti and Tiziana Cimoli

From locking a token sale to unlocking funds at a specific time for a game, contracts sometimes need to rely on the current time. This is usually done via `block.timestamp` or its alias `now` in Solidity. But where does that value come from? From the miners! Because a transaction's miner has leeway in reporting the time at which the mining occurred, good smart contracts will avoid relying strongly on the time advertised. Note that `block.timestamp` is also sometimes (mis)used in the generation of random numbers as is discussed in #6. Bad Randomness.

Real World Impact:

- GovernMental

Example:

1. A **game** pays out the very first player at midnight today.
2. A malicious **miner** includes his or her attempt to win the game and sets the timestamp to midnight.
3. A bit before midnight the miner ends up mining the block. The real current time is "close enough" to midnight (the currently set timestamp for the block), other nodes on the network decide to accept the block.

Code Example:

The following function only accepts calls that come after a specific date. Since miners can influence their block's timestamp (to a certain extent), they can attempt to mine a block containing their transaction with a block timestamp set in the future. If it is close enough, it will be accepted on the network and the transaction will give the miner ether before any other player could have attempted to win the game:

```
function play() public {
        require(now > 1521763200 && neverPlayed == true);
        neverPlayed = false;
        msg.sender.transfer(1500 ether);
}
```

Additional Resources:

- A survey of attacks on Ethereum smart contracts
- Predicting Random Numbers in Ethereum Smart Contracts
- Making smart contracts smarter

9 Short Address Attack

also known as or related to **off-chain issues**, **client vulnerabilities**

The service preparing the data for token transfers assumed that users will input 20-byte long addresses, but the length of the addresses was not actually checked.

— Paweł Bylica

Short address attacks are a side-effect of the EVM itself accepting incorrectly padded arguments. Attackers can exploit this by using specially-crafted addresses to make poorly coded clients encode arguments incorrectly before including them in transactions. Is this an EVM issue or a client issue? Should it be fixed in smart contracts instead? While everyone has a different opinion, the fact is that a great deal of ether could be directly impacted by this issue. While this vulnerability has yet to be exploited in the wild, it is a good demonstration of problems arising from the interaction between clients and the Ethereum blockchain. Other off-chain issues exist: an important one is the Ethereum ecosystem's deep trust in specific Javascript front ends, browser plugins and public nodes. An infamous off-chain exploit was used in the hack of the Coindash ICO that modified the company's Ethereum address on their webpage to trick participants into sending ethers to the attacker's address.

Timeline of discovery:

Date	Event
April 6, 2017	How to Find $10M Just by Reading the Blockchain

Real World Impact:

- unknown exchange(s)

Example:

1. An exchange API has a trading function that takes a recipient address and an amount.
2. The API then interacts with the smart contract `transfer(address _to, uint256 _amount)` function with padded arguments: it prepends the address (of an expected 20-byte length) with 12 zero bytes to make it 32-byte long
3. Bob (`0x3bdde1e9fbaef2579dd63e2abbf0be445ab93f00`) asks Alice to transfer him 20 tokens. He maliciously gives her his address truncated to remove the trailing zeroes.
4. Alice uses the exchange API with the shorter 19-byte address of Bob (`0x3bdde1e9fbaef2579dd63e2abbf0be445ab93f`).
5. The API pads the address with 12 zero bytes, making it 31 bytes instead of the 32 bytes. Effectively stealing one byte from the following `_amount` argument.
6. Eventually, the EVM executing the smart contract's code will remark that the data is not properly padded and will add the missing byte at the end of the `_amount` argument. Effectively transfering 256 times more tokens than thought.

Additional Resources:

- The ERC20 Short Address Attack Explained
- Analyzing the ERC20 Short Address Attack
- Smart Contract Short Address Attack Mitigation Failure
- Remove short address attack checks from tokens

? 10. Unknown Unknowns

We believe more security audits or more tests would have made no difference. The main problem was that reviewers did not know what to look for.

— Christoph Jentzsch

Ethereum is still in its infancy. The main language used to develop smart contracts, Solidity, has yet to reach a stable version and the ecosystem's tools are still experimental. Some of the most damaging smart contract vulnerabilities surprised everyone, and there is no reason to believe there will not be another one that will be equally unexpected or equally destructive. As long as investors decide to place large amounts of money on complex but lightly-audited code, we will continue to see new discoveries leading to dire consequences. Methods of formally verifying smart contracts are not yet mature, but they seem to hold great promise as ways past today's shaky status quo. As new classes of vulnerabilities continue to be found, developers will need to stay on their feet, and new tools will need to be developed to find them before the bad guys do. This top 10 will likely evolve rapidly until smart contract development reaches a state of steadiness and maturity.