Smart Contract Security

Security is one of the most important considerations when writing smart contracts. In the field of smart contract programming, mistakes are costly and easily exploited. In this chapter we will look at security best practices and design patterns, as well as "security antipatterns," which are practices and patterns that can introduce vulnerabilities in our smart contracts.

As with other programs, a smart contract will execute exactly what is written, which is not always what the programmer intended. Furthermore, all smart contracts are public, and any user can interact with them simply by creating a transaction. Any vulnerability can be exploited, and losses are almost always impossible to recover. It is therefore critical to follow best practices and use well-tested design patterns.

Security Best Practices

Defensive programming is a style of programming that is particularly well suited to smart contracts. It emphasizes the following, all of which are best practices:

Minimalism/simplicity

Complexity is the enemy of security. The simpler the code, and the less it does, the lower the chances are of a bug or unforeseen effect occurring. When first engaging in smart contract programming, developers are often tempted to try to write a lot of code. Instead, you should look through your smart contract code and try to find ways to do less, with fewer lines of code, less complexity, and fewer "features." If someone tells you that their project has produced "thousands of lines of code" for their smart contracts, you should question the security of that project. Simpler is more secure.

Code reuse

Try not to reinvent the wheel. If a library or contract already exists that does most of what you need, reuse it. Within your own code, follow the DRY principle: Don’t Repeat Yourself. If you see any snippet of code repeated more than once, ask yourself whether it could be written as a function or library and reused. Code that has been extensively used and tested is likely more secure than any new code you write. Beware of “Not Invented Here” syndrome, where you are tempted to "improve" a feature or component by building it from scratch. The security risk is often greater than the improvement value.

Code quality

Smart contract code is unforgiving. Every bug can lead to monetary loss. You should not treat smart contract programming the same way as general-purpose programming. Writing DApps in Solidity is not like creating a web widget in JavaScript. Rather, you should apply rigorous engineering and software development methodologies, as you would in aerospace engineering or any similarly unforgiving discipline. Once you "launch" your code, there’s little you can do to fix any problems.

Readability/auditability

Your code should be clear and easy to comprehend. The easier it is to read, the easier it is to audit. Smart contracts are public, as everyone can read the bytecode and anyone can reverse-

engineer it. Therefore, it is beneficial to develop your work in public, using collaborative and open source methodologies, to draw upon the collective wisdom of the developer community and benefit from the highest common denominator of open source development. You should write code that is well documented and easy to read, following the style and naming conventions that are part of the Ethereum community.

Test coverage

Test everything that you can. Smart contracts run in a public execution environment, where anyone can execute them with whatever input they want. You should never assume that input, such as function arguments, is well formed, properly bounded, or has a benign purpose. Test all arguments to make sure they are within expected ranges and properly formatted before allowing execution of your code to continue.

Security Risks and Antipatterns

As a smart contract programmer, you should be familiar with the most common security risks, so as to be able to detect and avoid the programming patterns that leave your contracts exposed to these risks. In the next several sections we will look at different security risks, examples of how vulnerabilities can arise, and countermeasures or preventative solutions that can be used to address them.

Reentrancy

One of the features of Ethereum smart contracts is their ability to call and utilize code from other external contracts. Contracts also typically handle ether, and as such often send ether to various external user addresses. These operations require the contracts to submit external calls. These external calls can be hijacked by attackers, who can force the contracts to execute further code (through a fallback function), including calls back into themselves. Attacks of this kind were used in the infamous DAO hack.

For further reading on reentrancy attacks, see Gus Guimareas's blog post on the subject and the Ethereum Smart Contract Best Practices.

The Vulnerability

This type of attack can occur when a contract sends ether to an unknown address. An attacker can carefully construct a contract at an external address that contains malicious code in the fallback function. Thus, when a contract sends ether to this address, it will invoke the malicious code. Typically the malicious code executes a function on the vulnerable contract, performing operations not expected by the developer. The term "reentrancy" comes from the fact that the external malicious contract calls a function on the vulnerable contract and the path of code execution "*reenters*" it.

To clarify this, consider the simple vulnerable contract in EtherStore.sol, which acts as an Ethereum vault that allows depositors to withdraw only 1 ether per week.

Example 1. EtherStore.sol

```
contract EtherStore {

    uint256 public withdrawalLimit = 1 ether;
    mapping(address => uint256) public lastWithdrawTime;
    mapping(address => uint256) public balances;

    function depositFunds() external payable {
        balances[msg.sender] += msg.value;
    }

    function withdrawFunds (uint256 _weiToWithdraw) public {
        require(balances[msg.sender] >= _weiToWithdraw);
        // limit the withdrawal
        require(_weiToWithdraw <= withdrawalLimit);
        // limit the time allowed to withdraw
        require(now >= lastWithdrawTime[msg.sender] + 1 weeks);
        require(msg.sender.call.value(_weiToWithdraw)());
        balances[msg.sender] -= _weiToWithdraw;
        lastWithdrawTime[msg.sender] = now;
    }
 }
```

This contract has two public functions, `depositFunds` and `withdrawFunds`. The `depositFunds` function simply increments the sender's balance. The `withdrawFunds` function allows the sender to specify the amount of wei to withdraw. This function is intended to succeed only if the requested amount to withdraw is less than 1 ether and a withdrawal has not occurred in the last week.

The vulnerability is in line 17, where the contract sends the user their requested amount of ether. Consider an attacker who has created the contract in Attack.sol.

Example 2. Attack.sol

```
import "EtherStore.sol";

contract Attack {
  EtherStore public etherStore;

  // intialize the etherStore variable with the contract address
  constructor(address _etherStoreAddress) {
      etherStore = EtherStore(_etherStoreAddress);
  }

  function attackEtherStore() external payable {
      // attack to the nearest ether
      require(msg.value >= 1 ether);
      // send eth to the depositFunds() function
```

```
        etherStore.depositFunds.value(1 ether)();
        // start the magic
        etherStore.withdrawFunds(1 ether);
    }

    function collectEther() public {
        msg.sender.transfer(this.balance);
    }

    // fallback function - where the magic happens
    function () payable {
        if (etherStore.balance > 1 ether) {
            etherStore.withdrawFunds(1 ether);
        }
    }
}
```

How might the exploit occur? First, the attacker would create the malicious contract (let's say at the address `0x0···123`) with the `EtherStore`'s contract address as the sole constructor parameter. This would initialize and point the public variable `etherStore` to the contract to be attacked.

The attacker would then call the `attackEtherStore` function, with some amount of ether greater than or equal to 1—let's assume `1 ether` for the time being. In this example, we will also assume a number of other users have deposited ether into this contract, such that its current balance is `10 ether`. The following will then occur:

1. *Attack.sol*, line 15: The `depositFunds` function of the `EtherStore` contract will be called with a `msg.value` of `1 ether` (and a lot of gas). The sender (`msg.sender`) will be the malicious contract (`0x0···123`). Thus, `balances[0x0..123] = 1 ether`.
2. *Attack.sol*, line 17: The malicious contract will then call the `withdrawFunds` function of the `EtherStore` contract with a parameter of `1 ether`. This will pass all the requirements (lines 12–16 of the `EtherStore` contract) as no previous withdrawals have been made.
3. *EtherStore.sol*, line 17: The contract will send `1 ether` back to the malicious contract.
4. *Attack.sol*, line 25: The payment to the malicious contract will then execute the fallback function.
5. *Attack.sol*, line 26: The total balance of the EtherStore contract was `10 ether` and is now `9 ether`, so this if statement passes.
6. *Attack.sol*, line 27: The fallback function calls the `EtherStore withdrawFunds` function again and '*reenters*' the `EtherStore` contract.
7. *EtherStore.sol*, line 11: In this second call to `withdrawFunds`, the attacking contract's balance is still `1 ether` as line 18 has not yet been executed. Thus, we still have `balances[0x0..123] = 1 ether`. This is also the case for the `lastWithdrawTime` variable. Again, we pass all the requirements.
8. *EtherStore.sol*, line 17: The attacking contract withdraws another `1 ether`.
9. Steps 4–8 repeat until it is no longer the case that `EtherStore.balance > 1`, as dictated by line 26

in *Attack.sol*.

10. *Attack.sol*, line 26: Once there is 1 (or less) ether left in the `EtherStore` contract, this `if` statement will fail. This will then allow lines 18 and 19 of the `EtherStore` contract to be executed (for each call to the `withdrawFunds` function).
11. *EtherStore.sol*, lines 18 and 19: The `balances` and `lastWithdrawTime` mappings will be set and the execution will end.

The final result is that the attacker has withdrawn all but 1 ether from the `EtherStore` contract in a single transaction.

Preventative Techniques

There are a number of common techniques that help avoid potential reentrancy vulnerabilities in smart contracts.The first is to (whenever possible) use the built-in transfer function when sending ether to external contracts. The transfer function only sends 2300 gas with the external call, which is not enough for the destination address/contract to call another contract (i.e., reenter the sending contract).

The second technique is to ensure that all logic that changes state variables happens before ether is sent out of the contract (or any external call). In the `EtherStore` example, lines 18 and 19 of *EtherStore.sol* should be put before line 17. It is good practice for any code that performs external calls to unknown addresses to be the last operation in a localized function or piece of code execution. This is known as the checks-effects-interactions pattern.

A third technique is to introduce a mutex—that is, to add a state variable that locks the contract during code execution, preventing reentrant calls.

Applying all of these techniques (using all three is unnecessary, but we do it for demonstrative purposes) to *EtherStore.sol*, gives the reentrancy-free contract:

```
contract EtherStore {

    // initialize the mutex
    bool reEntrancyMutex = false;
    uint256 public withdrawalLimit = 1 ether;
    mapping(address => uint256) public lastWithdrawTime;
    mapping(address => uint256) public balances;

    function depositFunds() external payable {
        balances[msg.sender] += msg.value;
    }

    function withdrawFunds (uint256 _weiToWithdraw) public {
        require(!reEntrancyMutex);
        require(balances[msg.sender] >= _weiToWithdraw);
        // limit the withdrawal
        require(_weiToWithdraw <= withdrawalLimit);
        // limit the time allowed to withdraw
        require(now >= lastWithdrawTime[msg.sender] + 1 weeks);
```

```
        balances[msg.sender] -= _weiToWithdraw;
        lastWithdrawTime[msg.sender] = now;
        // set the reEntrancy mutex before the external call
        reEntrancyMutex = true;
        msg.sender.transfer(_weiToWithdraw);
        // release the mutex after the external call
        reEntrancyMutex = false;
    }
}
```

Real-World Example: The DAO

The DAO (Decentralized Autonomous Organization) attack was one of the major hacks that occurred in the early development of Ethereum. At the time, the contract held over $150 million. Reentrancy played a major role in the attack, which ultimately led to the hard fork that created Ethereum Classic (ETC). For a good analysis of the DAO exploit, see http://bit.ly/2EQaLCI. More information on Ethereum's fork history, the DAO hack timeline, and the birth of ETC in a hard fork can be found in [ethereum_standards].

Arithmetic Over/Underflows

The Ethereum Virtual Machine specifies fixed-size data types for integers. This means that an integer variable can represent only a certain range of numbers. A `uint8`, for example, can only store numbers in the range [0,255]. Trying to store `256` into a `uint8` will result in `0`. If care is not taken, variables in Solidity can be exploited if user input is unchecked and calculations are performed that result in numbers that lie outside the range of the data type that stores them.

For further reading on arithmetic over/underflows, see "How to Secure Your Smart Contracts", Ethereum Smart Contract Best Practices, and "Ethereum, Solidity and integer overflows: programming blockchains like 1970".

The Vulnerability

An over/underflow occurs when an operation is performed that requires a fixed-size variable to store a number (or piece of data) that is outside the range of the variable's data type.

For example, subtracting `1` from a `uint8` (unsigned integer of 8 bits; i.e., nonnegative) variable whose value is `0` will result in the number `255`. This is an *underflow*. We have assigned a number below the range of the `uint8`, so the result *wraps around* and gives the largest number a `uint8` can store. Similarly, adding `2^8=256` to a `uint8` will leave the variable unchanged, as we have wrapped around the entire length of the `uint`. Two simple analogies of this behavior are odometers in cars, which measure distance traveled (they reset to 000000, after the largest number, i.e., 999999, is surpassed) and periodic mathematical functions (adding 2π to the argument of sin leaves the value unchanged).

Adding numbers larger than the data type's range is called an *overflow*. For clarity, adding `257` to a `uint8` that currently has a value of `0` will result in the number `1`. It is sometimes instructive to think of fixed-size variables as being cyclic, where we start again from zero if we add numbers above the

largest possible stored number, and start counting down from the largest number if we subtract from zero. In the case of signed `int` types, which *can* represent negative numbers, we start again once we reach the largest negative value; for example, if we try to subtract `1` from a `int8` whose value is `-128`, we will get `127`.

These kinds of numerical gotchas allow attackers to misuse code and create unexpected logic flows. For example, consider the TimeLock contract in TimeLock.sol.

Example 3. TimeLock.sol

```
contract TimeLock {

    mapping(address => uint) public balances;
    mapping(address => uint) public lockTime;

    function deposit() external payable {
        balances[msg.sender] += msg.value;
        lockTime[msg.sender] = now + 1 weeks;
    }

    function increaseLockTime(uint _secondsToIncrease) public {
        lockTime[msg.sender] += _secondsToIncrease;
    }

    function withdraw() public {
        require(balances[msg.sender] > 0);
        require(now > lockTime[msg.sender]);
        uint transferValue = balances[msg.sender];
        balances[msg.sender] = 0;
        msg.sender.transfer(transferValue);
    }
}
```

This contract is designed to act like a time vault: users can deposit ether into the contract and it will be locked there for at least a week. The user may extend the wait time to longer than 1 week if they choose, but once deposited, the user can be sure their ether is locked in safely for at least a week—or so this contract intends.

In the event that a user is forced to hand over their private key, a contract such as this might be handy to ensure their ether is unobtainable for a short period of time. But if a user had locked in `100 ether` in this contract and handed their keys over to an attacker, the attacker could use an overflow to receive the ether, regardless of the `lockTime`.

The attacker could determine the current `lockTime` for the address they now hold the key for (it's a public variable). Let's call this `userLockTime`. They could then call the `increaseLockTime` function and pass as an argument the number `2^256 - userLockTime`. This number would be added to the current `userLockTime` and cause an overflow, resetting `lockTime[msg.sender]` to `0`. The attacker could then simply call the `withdraw` function to obtain their reward.

Let's look at another example (Underflow vulnerability example from Ethernaut challenge), this one from the Ethernaut challenges.

SPOILER ALERT: *If you have not yet done the Ethernaut challenges, this gives a solution to one of the levels.*

Example 4. Underflow vulnerability example from Ethernaut challenge

```
pragma solidity ^0.4.18;

contract Token {

  mapping(address => uint) balances;
  uint public totalSupply;

  function Token(uint _initialSupply) {
    balances[msg.sender] = totalSupply = _initialSupply;
  }

  function transfer(address _to, uint _value) public returns (bool) {
    require(balances[msg.sender] - _value >= 0);
    balances[msg.sender] -= _value;
    balances[_to] += _value;
    return true;
  }

  function balanceOf(address _owner) public constant returns (uint balance) {
    return balances[_owner];
  }
}
```

This is a simple token contract that employs a `transfer` function, allowing participants to move their tokens around. Can you see the error in this contract?

The flaw comes in the `transfer` function. The require statement on line 13 can be bypassed using an underflow. Consider a user with a zero balance. They could call the `transfer` function with any nonzero `_value` and pass the require statement on line 13. This is because `balances[msg.sender]` is 0 (and a `uint256`), so subtracting any positive amount (excluding `2^256`) will result in a positive number, as described previously. This is also true for line 14, where the balance will be credited with a positive number. Thus, in this example, an attacker can achieve free tokens due to an underflow vulnerability.

Preventative Techniques

The current conventional technique to guard against under/overflow vulnerabilities is to use or build mathematical libraries that replace the standard math operators addition, subtraction, and multiplication (division is excluded as it does not cause over/underflows and the EVM reverts on division by 0).

OpenZeppelin has done a great job of building and auditing secure libraries for the Ethereum community. In particular, its SafeMath library can be used to avoid under/overflow vulnerabilities.

To demonstrate how these libraries are used in Solidity, let's correct the `TimeLock` contract using the `SafeMath` library. The overflow-free version of the contract is:

```
library SafeMath {

  function mul(uint256 a, uint256 b) internal pure returns (uint256) {
    if (a == 0) {
      return 0;
    }
    uint256 c = a * b;
    assert(c / a == b);
    return c;
  }

  function div(uint256 a, uint256 b) internal pure returns (uint256) {
    // assert(b > 0); // Solidity automatically throws when dividing by 0
    uint256 c = a / b;
    // assert(a == b * c + a % b); // This holds in all cases
    return c;
  }

  function sub(uint256 a, uint256 b) internal pure returns (uint256) {
    assert(b <= a);
    return a - b;
  }

  function add(uint256 a, uint256 b) internal pure returns (uint256) {
    uint256 c = a + b;
    assert(c >= a);
    return c;
  }
}

contract TimeLock {
    using SafeMath for uint; // use the library for uint type
    mapping(address => uint256) public balances;
    mapping(address => uint256) public lockTime;

    function deposit() external payable {
        balances[msg.sender] = balances[msg.sender].add(msg.value);
        lockTime[msg.sender] = now.add(1 weeks);
    }

    function increaseLockTime(uint256 _secondsToIncrease) public {
        lockTime[msg.sender] = lockTime[msg.sender].add(_secondsToIncrease);
    }
```

```
    function withdraw() public {
        require(balances[msg.sender] > 0);
        require(now > lockTime[msg.sender]);
        uint256 transferValue = balances[msg.sender];
        balances[msg.sender] = 0;
        msg.sender.transfer(transferValue);
    }
}
```

Notice that all standard math operations have been replaced by those defined in the `SafeMath` library. The `TimeLock` contract no longer performs any operation that is capable of under/overflow.

Real-World Examples: PoWHC and Batch Transfer Overflow (CVE-2018–10299)

Proof of Weak Hands Coin (PoWHC), originally devised as a joke of sorts, was a Ponzi scheme written by an internet collective. Unfortunately it seems that the author(s) of the contract had not seen over/underflows before, and consequently 866 ether were liberated from its contract. Eric Banisadr gives a good overview of how the underflow occurred (which is not too dissimilar to the Ethernaut challenge described earlier) in his blog post on the event.

Another example comes from the implementation of a `batchTransfer()` function into a group of ERC20 token contracts. The implementation contained an overflow vulnerability; you can read about the details in PeckShield's account.

Unexpected Ether

Typically, when ether is sent to a contract it must execute either the fallback function or another function defined in the contract. There are two exceptions to this, where ether can exist in a contract without having executed any code. Contracts that rely on code execution for all ether sent to them can be vulnerable to attacks where ether is forcibly sent.

For further reading on this, see "How to Secure Your Smart Contracts" and "Solidity Security Patterns - Forcing Ether to a Contract".

The Vulnerability

Acommon defensive programming technique that is useful in enforcing correct state transitions or validating operations is *invariant checking*. This technique involves defining a set of invariants (metrics or parameters that should not change) and checking that they remain unchanged after a single (or many) operation(s). This is typically good design, provided the invariants being checked are in fact invariants. One example of an invariant is the `totalSupply` of a fixed-issuance ERC20 token. As no function should modify this invariant, one could add a check to the `transfer` function that ensures the `totalSupply` remains unmodified, to guarantee the function is working as expected.

In particular, there is one apparent invariant that it may be tempting to use but that can in fact be manipulated by external users (regardless of the rules put in place in the smart contract). This is the current ether stored in the contract. Often when developers first learn Solidity they have the

misconception that a contract can only accept or obtain ether via payable functions. This misconception can lead to contracts that have false assumptions about the ether balance within them, which can lead to a range of vulnerabilities. The smoking gun for this vulnerability is the (incorrect) use of `this.balance`.

There are two ways in which ether can (forcibly) be sent to a contract without using a payable function or executing any code on the contract:

Self-destruct/suicide

Any contract is able to implement the `selfdestruct` function, which removes all bytecode from the contract address and sends all ether stored there to the parameter-specified address. If this specified address is also a contract, no functions (including the fallback) get called. Therefore, the `selfdestruct` function can be used to forcibly send ether to any contract regardless of any code that may exist in the contract, even contracts with no payable functions. This means any attacker can create a contract with a `selfdestruct` function, send ether to it, call `selfdestruct(target)` and force ether to be sent to a `target` contract. Martin Swende has an excellent blog post describing some quirks of the self-destruct opcode (Quirk #2) along with an account of how client nodes were checking incorrect invariants, which could have led to a rather catastrophic crash of the Ethereum network.

Pre-sent ether

Another way to get ether into a contract is to preload the contract address with ether. Contract addresses are deterministic—in fact, the address is calculated from the Keccak-256 (commonly synonymous with SHA-3) hash of the address creating the contract and the transaction nonce that creates the contract. Specifically, it is of the form `address = sha3(rlp.encode([account_address,transaction_nonce]))` (see Adrian Manning's discussion of "Keyless Ether" for some fun use cases of this). This means anyone can calculate what a contract's address will be before it is created and send ether to that address. When the contract is created it will have a nonzero ether balance.

Let's explore some pitfalls that can arise given this knowledge. Consider the overly simple contract in EtherGame.sol.

Example 5. EtherGame.sol

```
contract EtherGame {

    uint public payoutMileStone1 = 3 ether;
    uint public mileStone1Reward = 2 ether;
    uint public payoutMileStone2 = 5 ether;
    uint public mileStone2Reward = 3 ether;
    uint public finalMileStone = 10 ether;
    uint public finalReward = 5 ether;

    mapping(address => uint) redeemableEther;
    // Users pay 0.5 ether. At specific milestones, credit their accounts.
    function play() external payable {
        require(msg.value == 0.5 ether); // each play is 0.5 ether
        uint currentBalance = this.balance + msg.value;
```

```
        // ensure no players after the game has finished
        require(currentBalance <= finalMileStone);
        // if at a milestone, credit the player's account
        if (currentBalance == payoutMileStone1) {
            redeemableEther[msg.sender] += mileStone1Reward;
        }
        else if (currentBalance == payoutMileStone2) {
            redeemableEther[msg.sender] += mileStone2Reward;
        }
        else if (currentBalance == finalMileStone ) {
            redeemableEther[msg.sender] += finalReward;
        }
        return;
    }

    function claimReward() public {
        // ensure the game is complete
        require(this.balance == finalMileStone);
        // ensure there is a reward to give
        require(redeemableEther[msg.sender] > 0);
        uint transferValue = redeemableEther[msg.sender];
        redeemableEther[msg.sender] = 0;
        msg.sender.transfer(transferValue);
    }
}
```

This contract represents a simple game (which would naturally involve race conditions) where players send 0.5 ether to the contract in the hopes of being the player that reaches one of three milestones first. Milestones are denominated in ether. The first to reach the milestone may claim a portion of the ether when the game has ended. The game ends when the final milestone (10 ether) is reached; users can then claim their rewards.

The issues with the `EtherGame` contract come from the poor use of `this.balance` in both lines 14 (and by association 16) and 32. A mischievous attacker could forcibly send a small amount of ether—say, 0.1 ether—via the `selfdestruct` function (discussed earlier) to prevent any future players from reaching a milestone. `this.balance` will never be a multiple of 0.5 ether thanks to this 0.1 ether contribution, because all legitimate players can only send 0.5-ether increments. This prevents all the if conditions on lines 18, 21, and 24 from being true.

Even worse, a vengeful attacker who missed a milestone could forcibly send 10 ether (or an equivalent amount of ether that pushes the contract's balance above the `finalMileStone`), which would lock all rewards in the contract forever. This is because the `claimReward` function will always revert, due to the require on line 32 (i.e., because `this.balance` is greater than `finalMileStone`).

Preventative Techniques

This sort of vulnerability typically arises from the misuse of `this.balance`. Contract logic, when possible, should avoid being dependent on exact values of the balance of the contract, because it can be artificially manipulated. If applying logic based on `this.balance`, you have to cope with

unexpected balances.

If exact values of deposited ether are required, a self-defined variable should be used that is incremented in payable functions, to safely track the deposited ether. This variable will not be influenced by the forced ether sent via a `selfdestruct` call.

With this in mind, a corrected version of the `EtherGame` contract could look like:

```
contract EtherGame {

    uint public payoutMileStone1 = 3 ether;
    uint public mileStone1Reward = 2 ether;
    uint public payoutMileStone2 = 5 ether;
    uint public mileStone2Reward = 3 ether;
    uint public finalMileStone = 10 ether;
    uint public finalReward = 5 ether;
    uint public depositedWei;

    mapping (address => uint) redeemableEther;

    function play() external payable {
        require(msg.value == 0.5 ether);
        uint currentBalance = depositedWei + msg.value;
        // ensure no players after the game has finished
        require(currentBalance <= finalMileStone);
        if (currentBalance == payoutMileStone1) {
            redeemableEther[msg.sender] += mileStone1Reward;
        }
        else if (currentBalance == payoutMileStone2) {
            redeemableEther[msg.sender] += mileStone2Reward;
        }
        else if (currentBalance == finalMileStone ) {
            redeemableEther[msg.sender] += finalReward;
        }
        depositedWei += msg.value;
        return;
    }

    function claimReward() public {
        // ensure the game is complete
        require(depositedWei == finalMileStone);
        // ensure there is a reward to give
        require(redeemableEther[msg.sender] > 0);
        uint transferValue = redeemableEther[msg.sender];
        redeemableEther[msg.sender] = 0;
        msg.sender.transfer(transferValue);
    }
 }
```

Here, we have created a new variable, `depositedWei`, which keeps track of the known ether

deposited, and it is this variable that we use for our tests. Note that we no longer have any reference to `this.balance`.

Further Examples

A few examples of exploitable contracts were given in the Underhanded Solidity Coding Contest, which also provides extended examples of a number of the pitfalls raised in this section.

DELEGATECALL

The `CALL` and `DELEGATECALL` opcodes are useful in allowing Ethereum developers to modularize their code.Standard external message calls to contracts are handled by the `CALL` opcode, whereby code is run in the context of the external contract/function. The `DELEGATECALL` opcode is almost identical, except that the code executed at the targeted address is run in the context of the calling contract, and `msg.sender` and `msg.value` remain unchanged. This feature enables the implementation of *libraries*, allowing developers to deploy reusable code once and call it from future contracts.

Although the differences between these two opcodes are simple and intuitive, the use of `DELEGATECALL` can lead to unexpected code execution.

For further reading, see Loi.Luu's Ethereum Stack Exchange question on this topic and the Solidity docs.

The Vulnerability

As a result of the context-preserving nature of `DELEGATECALL`, building vulnerability-free custom libraries is not as easy as one might think. The code in libraries themselves can be secure and vulnerability-free; however, when run in the context of another application new vulnerabilities can arise.Let's see a fairly complex example of this, using Fibonacci numbers.

Consider the library in FibonacciLib.sol, which can generate the Fibonacci sequence and sequences of similar form. (Note: this code was modified from https://bit.ly/2MReuii.)

Example 6. FibonacciLib.sol

```
// library contract - calculates Fibonacci-like numbers
contract FibonacciLib {
    // initializing the standard Fibonacci sequence
    uint public start;
    uint public calculatedFibNumber;

    // modify the zeroth number in the sequence
    function setStart(uint _start) public {
        start = _start;
    }

    function setFibonacci(uint n) public {
        calculatedFibNumber = fibonacci(n);
    }
```

```
    function fibonacci(uint n) internal returns (uint) {
        if (n == 0) return start;
        else if (n == 1) return start + 1;
        else return fibonacci(n - 1) + fibonacci(n - 2);
    }
}
```

This library provides a function that can generate the *n*-th Fibonacci number in the sequence. It allows users to change the starting number of the sequence (`start`) and calculate the *n*-th Fibonacci-like numbers in this new sequence.

Let us now consider a contract that utilizes this library, shown in FibonacciBalance.sol.

Example 7. FibonacciBalance.sol

```
contract FibonacciBalance {

    address public fibonacciLibrary;
    // the current Fibonacci number to withdraw
    uint public calculatedFibNumber;
    // the starting Fibonacci sequence number
    uint public start = 3;
    uint public withdrawalCounter;
    // the Fibonancci function selector
    bytes4 constant fibSig = bytes4(sha3("setFibonacci(uint256)"));

    // constructor - loads the contract with ether
    constructor(address _fibonacciLibrary) external payable {
        fibonacciLibrary = _fibonacciLibrary;
    }

    function withdraw() {
        withdrawalCounter += 1;
        // calculate the Fibonacci number for the current withdrawal user-
        // this sets calculatedFibNumber
        require(fibonacciLibrary.delegatecall(fibSig, withdrawalCounter));
        msg.sender.transfer(calculatedFibNumber * 1 ether);
    }

    // allow users to call Fibonacci library functions
    function() public {
        require(fibonacciLibrary.delegatecall(msg.data));
    }
}
```

This contract allows a participant to withdraw ether from the contract, with the amount of ether

being equal to the Fibonacci number corresponding to the participant's withdrawal order; i.e., the first participant gets 1 ether, the second also gets 1, the third gets 2, the fourth gets 3, the fifth 5, and so on (until the balance of the contract is less than the Fibonacci number being withdrawn).

There are a number of elements in this contract that may require some explanation. Firstly, there is an interesting-looking variable, `fibSig`. This holds the first 4 bytes of the Keccak-256 (SHA-3) hash of the string `'setFibonacci(uint256)'`. This is known as the function selector and is put into `calldata` to specify which function of a smart contract will be called. It is used in the `delegatecall` function on line 21 to specify that we wish to run the `fibonacci(uint256)` function. The second argument in `delegatecall` is the parameter we are passing to the function. Secondly, we assume that the address for the `FibonacciLib` library is correctly referenced in the constructor (External Contract Referencing discusses some potential vulnerabilities relating to this kind of contract reference initialization).

Can you spot any errors in this contract? If one were to deploy this contract, fill it with ether, and call `withdraw`, it would likely revert.

You may have noticed that the state variable `start` is used in both the library and the main calling contract. In the library contract, `start` is used to specify the beginning of the Fibonacci sequence and is set to `0`, whereas it is set to `3` in the calling contract. You may also have noticed that the fallback function in the `FibonacciBalance` contract allows all calls to be passed to the library contract, which allows for the `setStart` function of the library contract to be called. Recalling that we preserve the state of the contract, it may seem that this function would allow you to change the state of the `start` variable in the local `FibonnacciBalance` contract. If so, this would allow one to withdraw more ether, as the resulting `calculatedFibNumber` is dependent on the `start` variable (as seen in the library contract). In actual fact, the `setStart` function does not (and cannot) modify the `start` variable in the `FibonacciBalance` contract. The underlying vulnerability in this contract is significantly worse than just modifying the `start` variable.

Before discussing the actual issue, let's take a quick detour to understand how state variables actually get stored in contracts. State or storage variables (variables that persist over individual transactions) are placed into *slots* sequentially as they are introduced in the contract. (There are some complexities here; consult the Solidity docs for a more thorough understanding.)

As an example, let's look at the library contract. It has two state variables, `start` and `calculatedFibNumber`. The first variable, `start`, is stored in the contract's storage at `slot[0]` (i.e., the first slot). The second variable, `calculatedFibNumber`, is placed in the next available storage slot, `slot[1]`. The function `setStart` takes an input and sets `start` to whatever the input was. This function therefore sets `slot[0]` to whatever input we provide in the `setStart` function. Similarly, the `setFibonacci` function sets `calculatedFibNumber` to the result of `fibonacci(n)`. Again, this is simply setting storage `slot[1]` to the value of `fibonacci(n)`.

Now let's look at the `FibonacciBalance` contract. Storage `slot[0]` now corresponds to the `fibonacciLibrary` address, and `slot[1]` corresponds to `calculatedFibNumber`. It is in this incorrect mapping that the vulnerability occurs. `delegatecall` *preserves contract context*. This means that code that is executed via `delegatecall` will act on the state (i.e., storage) of the calling contract.

Now notice that in `withdraw` on line 21 we execute `fibonacciLibrary.delegatecall(fibSig,withdrawalCounter)`. This calls the `setFibonacci` function,

which, as we discussed, modifies storage `slot[1]`, which in our current context is `calculatedFibNumber`. This is as expected (i.e., after execution, `calculatedFibNumber` is modified). However, recall that the `start` variable in the `FibonacciLib` contract is located in storage `slot[0]`, which is the `fibonacciLibrary` address in the current contract. This means that the function `fibonacci` will give an unexpected result. This is because it references `start` (`slot[0]`), which in the current calling context is the `fibonacciLibrary` address (which will often be quite large, when interpreted as a `uint`). Thus it is likely that the `withdraw` function will revert, as it will not contain `uint(fibonacciLibrary)` amount of ether, which is what `calculatedFibNumber` will return.

Even worse, the `FibonacciBalance` contract allows users to call all of the `fibonacciLibrary` functions via the fallback function at line 26. As we discussed earlier, this includes the `setStart` function. We discussed that this function allows anyone to modify or set storage `slot[0]`. In this case, storage `slot[0]` is the `fibonacciLibrary` address. Therefore, an attacker could create a malicious contract, convert the address to a `uint` (this can be done in Python easily using `int('<address>',16)`), and then call `setStart(<attack_contract_address_as_uint>)`. This will change `fibonacciLibrary` to the address of the attack contract. Then, whenever a user calls `withdraw` or the fallback function, the malicious contract will run (which can steal the entire balance of the contract) because we've modified the actual address for `fibonacciLibrary`. An example of such an attack contract would be:

```
contract Attack {
    uint storageSlot0; // corresponds to fibonacciLibrary
    uint storageSlot1; // corresponds to calculatedFibNumber

    // fallback - this will run if a specified function is not found
    function() public {
        storageSlot1 = 0; // we set calculatedFibNumber to 0, so if withdraw
        // is called we don't send out any ether
        <attacker_address>.transfer(this.balance); // we take all the ether
    }
 }
```

Notice that this attack contract modifies the `calculatedFibNumber` by changing storage `slot[1]`. In principle, an attacker could modify any other storage slots they choose, to perform all kinds of attacks on this contract. We encourage you to put these contracts into Remix and experiment with different attack contracts and state changes through these `delegatecall` functions.

It is also important to notice that when we say that `delegatecall` is state-preserving, we are not talking about the variable names of the contract, but rather the actual storage slots to which those names point. As you can see from this example, a simple mistake can lead to an attacker hijacking the entire contract and its ether.

Preventative Techniques

Solidity provides the `library` keyword for implementing library contracts (see the docs for further details). This ensures the library contract is stateless and non-self-destructable. Forcing libraries to be stateless mitigates the complexities of storage context demonstrated in this section. Stateless libraries also prevent attacks wherein attackers modify the state of the library directly in order to affect the contracts that depend on the library's code. As a general rule of thumb, when using

DELEGATECALL pay careful attention to the possible calling context of both the library contract and the calling contract, and whenever possible build stateless libraries.

Real-World Example: Parity Multisig Wallet (Second Hack)

The Second Parity Multisig Wallet hack is an example of how well-written library code can be exploited if run outside its intended context. There are a number of good explanations of this hack, such as "Parity Multisig Hacked. Again" and "An In-Depth Look at the Parity Multisig Bug".

To add to these references, let's explore the contracts that were exploited. The library and wallet contracts can be found on GitHub.

The library contract is as follows:

```
contract WalletLibrary is WalletEvents {

  ...

  // throw unless the contract is not yet initialized.
  modifier only_uninitialized { if (m_numOwners > 0) throw; _; }

  // constructor - just pass on the owner array to multiowned and
  // the limit to daylimit
  function initWallet(address[] _owners, uint _required, uint _daylimit)
      only_uninitialized {
    initDaylimit(_daylimit);
    initMultiowned(_owners, _required);
  }

  // kills the contract sending everything to `_to`.
  function kill(address _to) onlymanyowners(sha3(msg.data)) external {
    suicide(_to);
  }

  ...

}
```

And here's the wallet contract:

```
contract Wallet is WalletEvents {

  ...

  // METHODS

  // gets called when no other function matches
  function() payable {
    // just being sent some cash?
```

```
    if (msg.value > 0)
      Deposit(msg.sender, msg.value);
    else if (msg.data.length > 0)
      _walletLibrary.delegatecall(msg.data);
  }

  ...

  // FIELDS
  address constant _walletLibrary =
    0xcafecafecafecafecafecafecafecafecafecafe;
}
```

Notice that the `Wallet` contract essentially passes all calls to the `WalletLibrary` contract via a delegate call. The constant `_walletLibrary` address in this code snippet acts as a placeholder for the actually deployed `WalletLibrary` contract (which was at `0x863DF6BFa4469f3ead0bE8f9F2AAE51c91A907b4`).

The intended operation of these contracts was to have a simple low-cost deployable `Wallet` contract whose codebase and main functionality were in the `WalletLibrary` contract. Unfortunately, the `WalletLibrary` contract is itself a contract and maintains its own state. Can you see why this might be an issue?

It is possible to send calls to the `WalletLibrary` contract itself. Specifically, the `WalletLibrary` contract could be initialized and become owned. In fact, a user did this, calling the `initWallet` function on the `WalletLibrary` contract and becoming an owner of the library contract. The same user subsequently called the `kill` function. Because the user was an owner of the library contract, the modifier passed and the library contract self-destructed. As all `Wallet` contracts in existence refer to this library contract and contain no method to change this reference, all of their functionality, including the ability to withdraw ether, was lost along with the `WalletLibrary` contract. As a result, all ether in all Parity multisig wallets of this type instantly became lost or permanently unrecoverable.

Default Visibilities

Functions in Solidity have visibility specifiers that dictate how they can be called. The visibility determines whether a function can be called externally by users, by other derived contracts, only internally, or only externally. There are four visibility specifiers, which are described in detail in the Solidity docs. Functions default to `public`, allowing users to call them externally. We shall now see how incorrect use of visibility specifiers can lead to some devastating vulnerabilities in smart contracts.

The Vulnerability

The default visibility for functions is `public`, so functions that do not specify their visibility will be callable by external users. The issue arises when developers mistakenly omit visibility specifiers on functions that should be private (or only callable within the contract itself).

Let's quickly explore a trivial example:

```
contract HashForEther {

    function withdrawWinnings() {
        // Winner if the last 8 hex characters of the address are 0
        require(uint32(msg.sender) == 0);
        _sendWinnings();
     }

     function _sendWinnings() {
         msg.sender.transfer(this.balance);
     }
}
```

This simple contract is designed to act as an address-guessing bounty game. To win the balance of the contract, a user must generate an Ethereum address whose last 8 hex characters are 0. Once achieved, they can call the `withdrawWinnings` function to obtain their bounty.

Unfortunately, the visibility of the functions has not been specified. In particular, the `_sendWinnings` function is `public` (the default), and thus any address can call this function to steal the bounty.

Preventative Techniques

It is good practice to always specify the visibility of all functions in a contract, even if they are intentionally `public`. Recent versions of solc show a warning for functions that have no explicit visibility set, to encourage this practice.

Real-World Example: Parity Multisig Wallet (First Hack)

In the first Parity multisig hack, about $31M worth of Ether was stolen, mostly from three wallets. A good recap of exactly how this was done is given by Haseeb Qureshi.

Essentially, the multisig wallet is constructed from a base `Wallet` contract, which calls a library contract containing the core functionality (as described in Real-World Example: Parity Multisig Wallet (Second Hack)). The library contract contains the code to initialize the wallet, as can be seen from the following snippet:

```
contract WalletLibrary is WalletEvents {

  ...

  // METHODS

  ...

  // constructor is given number of sigs required to do protected
  // "onlymanyowners" transactions as well as the selection of addresses
```

```
  // capable of confirming them
  function initMultiowned(address[] _owners, uint _required) {
    m_numOwners = _owners.length + 1;
    m_owners[1] = uint(msg.sender);
    m_ownerIndex[uint(msg.sender)] = 1;
    for (uint i = 0; i < _owners.length; ++i)
    {
      m_owners[2 + i] = uint(_owners[i]);
      m_ownerIndex[uint(_owners[i])] = 2 + i;
    }
    m_required = _required;
  }

  ...

  // constructor - just pass on the owner array to multiowned and
  // the limit to daylimit
  function initWallet(address[] _owners, uint _required, uint _daylimit) {
    initDaylimit(_daylimit);
    initMultiowned(_owners, _required);
  }
}
```

Note that neither of the functions specifies their visibility, so both default to `public`. The `initWallet` function is called in the wallet's constructor, and sets the owners for the multisig wallet as can be seen in the `initMultiowned` function. Because these functions were accidentally left `public`, an attacker was able to call these functions on deployed contracts, resetting the ownership to the attacker's address. Being the owner, the attacker then drained the wallets of all their ether.

Entropy Illusion

All transactions on the Ethereum blockchain are deterministic state transition operations. This means that every transaction modifies the global state of the Ethereum ecosystem in a calculable way, with no uncertainty. This has the fundamental implication that there is no source of entropy or randomness in Ethereum. Achieving decentralized entropy (randomness) is a well-known problem for which many solutions have been proposed, including RANDAO, or using a chain of hashes, as described by Vitalik Buterin in the blog post "Validator Ordering and Randomness in PoS".

The Vulnerability

Some of the first contracts built on the Ethereum platform were based around gambling. Fundamentally, gambling requires uncertainty (something to bet on), which makes building a gambling system on the blockchain (a deterministic system) rather difficult. It is clear that the uncertainty must come from a source external to the blockchain. This is possible for bets between players (see for example the commit–reveal technique); however, it is significantly more difficult if you want to implement a contract to act as "the house" (like in blackjack or roulette). A common pitfall is to use future block variables—that is, variables containing information about the

transaction block whose values are not yet known, such as hashes, timestamps, block numbers, or gas limits. The issue with these are that they are controlled by the miner who mines the block, and as such are not truly random. Consider, for example, a roulette smart contract with logic that returns a black number if the next block hash ends in an even number. A miner (or miner pool) could bet $1M on black. If they solve the next block and find the hash ends in an odd number, they could happily not publish their block and mine another, until they find a solution with the block hash being an even number (assuming the block reward and fees are less than $1M). Using past or present variables can be even more devastating, as Martin Swende demonstrates in his excellent blog post. Furthermore, using solely block variables means that the pseudorandom number will be the same for all transactions in a block, so an attacker can multiply their wins by doing many transactions within a block (should there be a maximum bet).

Preventative Techniques

The source of entropy (randomness) must be external to the blockchain. This can be done among peers with systems such as commit–reveal, or via changing the trust model to a group of participants (as in RandDAO). This can also be done via a centralized entity that acts as a randomness oracle. Block variables (in general, there are some exceptions) should not be used to source entropy, as they can be manipulated by miners.

Real-World Example: PRNG Contracts

In February 2018 Arseny Reutov blogged about his analysis of 3,649 live smart contracts that were using some sort of pseudorandom number generator (PRNG); he found 43 contracts that could be exploited.

External Contract Referencing

One of the benefits of the Ethereum “world computer” is the ability to reuse code and interact with contracts already deployed on the network. As a result, a large number of contracts reference external contracts, usually via external message calls. These external message calls can mask malicious actors' intentions in some nonobvious ways, which we’ll now examine.

The Vulnerability

In Solidity, any address can be cast to a contract, regardless of whether the code at the address represents the contract type being cast. This can cause problems, especially when the author of the contract is trying to hide malicious code. Let’s illustrate this with an example.

Consider a piece of code like Rot13Encryption.sol, which rudimentarily implements the ROT13 cipher.

Example 8. Rot13Encryption.sol

```
// encryption contract
contract Rot13Encryption {

   event Result(string convertedString);
```

```
    // rot13-encrypt a string
    function rot13Encrypt (string text) public {
        uint256 length = bytes(text).length;
        for (var i = 0; i < length; i++) {
            byte char = bytes(text)[i];
            // inline assembly to modify the string
            assembly {
                // get the first byte
                char := byte(0,char)
                // if the character is in [n,z], i.e. wrapping
                if and(gt(char,0x6D), lt(char,0x7B))
                // subtract from the ASCII number 'a',
                // the difference between character <char> and 'z'
                { char:= sub(0x60, sub(0x7A,char)) }
                if iszero(eq(char, 0x20)) // ignore spaces
                // add 13 to char
                {mstore8(add(add(text,0x20), mul(i,1)), add(char,13))}
            }
        }
        emit Result(text);
    }

    // rot13-decrypt a string
    function rot13Decrypt (string text) public {
        uint256 length = bytes(text).length;
        for (var i = 0; i < length; i++) {
            byte char = bytes(text)[i];
            assembly {
                char := byte(0,char)
                if and(gt(char,0x60), lt(char,0x6E))
                { char:= add(0x7B, sub(char,0x61)) }
                if iszero(eq(char, 0x20))
                {mstore8(add(add(text,0x20), mul(i,1)), sub(char,13))}
            }
        }
        emit Result(text);
    }
}
```

This code simply takes a string (letters a–z, without validation) and *encrypts* it by shifting each character 13 places to the right (wrapping around `z`); i.e., `a` shifts to `n` and `x` shifts to `k`. The assembly in the preceding contract does not need to be understood to appreciate the issue being discussed, so readers unfamiliar with assembly can safely ignore it.

Now consider the following contract, which uses this code for its encryption:

```
import "Rot13Encryption.sol";
```

```
// encrypt your top-secret info
contract EncryptionContract {
    // library for encryption
    Rot13Encryption encryptionLibrary;

    // constructor - initialize the library
    constructor(Rot13Encryption _encryptionLibrary) {
        encryptionLibrary = _encryptionLibrary;
    }

    function encryptPrivateData(string privateInfo) {
        // potentially do some operations here
        encryptionLibrary.rot13Encrypt(privateInfo);
     }
 }
```

The issue with this contract is that the `encryptionLibrary` address is not public or constant. Thus, the deployer of the contract could give an address in the constructor that points to this contract:

```
// encryption contract
contract Rot26Encryption {

   event Result(string convertedString);

    // rot13-encrypt a string
    function rot13Encrypt (string text) public {
        uint256 length = bytes(text).length;
        for (var i = 0; i < length; i++) {
            byte char = bytes(text)[i];
            // inline assembly to modify the string
            assembly {
                // get the first byte
                char := byte(0,char)
                // if the character is in [n,z], i.e. wrapping
                if and(gt(char,0x6D), lt(char,0x7B))
                // subtract from the ASCII number 'a',
                // the difference between character <char> and 'z'
                { char:= sub(0x60, sub(0x7A,char)) }
                // ignore spaces
                if iszero(eq(char, 0x20))
                // add 26 to char!
                {mstore8(add(add(text,0x20), mul(i,1)), add(char,26))}
            }
        }
        emit Result(text);
    }

    // rot13-decrypt a string
    function rot13Decrypt (string text) public {
        uint256 length = bytes(text).length;
```

```
        for (var i = 0; i < length; i++) {
            byte char = bytes(text)[i];
            assembly {
                char := byte(0,char)
                if and(gt(char,0x60), lt(char,0x6E))
                { char:= add(0x7B, sub(char,0x61)) }
                if iszero(eq(char, 0x20))
                {mstore8(add(add(text,0x20), mul(i,1)), sub(char,26))}
            }
        }
        emit Result(text);
    }
}
```

This contract implements the ROT26 cipher, which shifts each character by 26 places (i.e., does nothing). Again, there is no need to understand the assembly in this contract. More simply, the attacker could have linked the following contract to the same effect:

```
contract Print{
    event Print(string text);

    function rot13Encrypt(string text) public {
        emit Print(text);
    }
 }
```

If the address of either of these contracts were given in the constructor, the `encryptPrivateData` function would simply produce an event that prints the unencrypted private data.

Although in this example a library-like contract was set in the constructor, it is often the case that a privileged user (such as an owner) can change library contract addresses. If a linked contract doesn't contain the function being called, the fallback function will execute. For example, with the line `encryptionLibrary.rot13Encrypt()`, if the contract specified by `encryptionLibrary` was:

```
 contract Blank {
     event Print(string text);
     function () {
         emit Print("Here");
         // put malicious code here and it will run
     }
 }
```

then an event with the text `Here` would be emitted. Thus, if users can alter contract libraries, they can in principle get other users to unknowingly run arbitrary code.

WARNING	The contracts represented here are for demonstrative purposes only and do not represent proper encryption. They should not be used for encryption.

Preventative Techniques

As demonstrated previously, safe contracts can (in some cases) be deployed in such a way that they behave maliciously. An auditor could publicly verify a contract and have its owner deploy it in a malicious way, resulting in a publicly audited contract that has vulnerabilities or malicious intent.

There are a number of techniques that prevent these scenarios.

One technique is to use the `new` keyword to create contracts. In the preceding example, the constructor could be written as:

```
constructor() {
    encryptionLibrary = new Rot13Encryption();
}
```

This way an instance of the referenced contract is created at deployment time, and the deployer cannot replace the `Rot13Encryption` contract without changing it.

Another solution is to hardcode external contract addresses.

In general, code that calls external contracts should always be audited carefully. As a developer, when defining external contracts, it can be a good idea to make the contract addresses public (which is not the case in the honey-pot example in the following section) to allow users to easily examine code referenced by the contract. Conversely, if a contract has a private variable contract address it can be a sign of someone behaving maliciously (as shown in the real-world example). If a user can change a contract address that is used to call external functions, it can be important (in a decentralized system context) to implement a time-lock and/or voting mechanism to allow users to see what code is being changed, or to give participants a chance to opt in/out with the new contract address.

Real-World Example: Reentrancy Honey Pot

A number of recent honey pots have been released on the mainnet. These contracts try to outsmart Ethereum hackers who try to exploit the contracts, but who in turn end up losing ether to the contract they expect to exploit. One example employs this attack by replacing an expected contract with a malicious one in the constructor. The code can be found here:

```
pragma solidity ^0.4.19;

contract Private_Bank
{
    mapping (address => uint) public balances;
    uint public MinDeposit = 1 ether;
    Log TransferLog;

    function Private_Bank(address _log)
    {
        TransferLog = Log(_log);
    }
```

```
    function Deposit()
    public
    payable
    {
        if(msg.value >= MinDeposit)
        {
            balances[msg.sender]+=msg.value;
            TransferLog.AddMessage(msg.sender,msg.value,"Deposit");
        }
    }

    function CashOut(uint _am)
    {
        if(_am<=balances[msg.sender])
        {
            if(msg.sender.call.value(_am)())
            {
                balances[msg.sender]-=_am;
                TransferLog.AddMessage(msg.sender,_am,"CashOut");
            }
        }
    }

    function() external payable{}

}

contract Log
{
    struct Message
    {
        address Sender;
        string  Data;
        uint Val;
        uint  Time;
    }

    Message[] public History;
    Message LastMsg;

    function AddMessage(address _adr,uint _val,string _data)
    public
    {
        LastMsg.Sender = _adr;
        LastMsg.Time = now;
        LastMsg.Val = _val;
        LastMsg.Data = _data;
        History.push(LastMsg);
    }
```

```
}
```

This post by one reddit user explains how they lost 1 ether to this contract by trying to exploit the reentrancy bug they expected to be present in the contract.

Short Address/Parameter Attack

This attack is not performed on Solidity contracts themselves, but on third-party applications that may interact with them. This section is added for completeness and to give the reader an awareness of how parameters can be manipulated in contracts.

For further reading, see "The ERC20 Short Address Attack Explained", "ICO Smart Contract Vulnerability: Short Address Attack", or this Reddit post.

The Vulnerability

When passing parameters to a smart contract, the parameters are encoded according to the ABI specification. It is possible to send encoded parameters that are shorter than the expected parameter length (for example, sending an address that is only 38 hex chars (19 bytes) instead of the standard 40 hex chars (20 bytes)). In such a scenario, the EVM will add zeros to the end of the encoded parameters to make up the expected length.

This becomes an issue when third-party applications do not validate inputs. The clearest example is an exchange that doesn't verify the address of an ERC20 token when a user requests a withdrawal. This example is covered in more detail in Peter Vessenes's post, "The ERC20 Short Address Attack Explained".

Consider the standard ERC20 transfer function interface, noting the order of the parameters:

```
function transfer(address to, uint tokens) public returns (bool success);
```

Now consider an exchange holding a large amount of a token (let's say `REP`) and a user who wishes to withdraw their share of 100 tokens. The user would submit their address, `0xdeaddeaddeaddeaddeaddeaddeaddeaddeaddead`, and the number of tokens, `100`. The exchange would encode these parameters in the order specified by the `transfer` function; that is, `address` then `tokens`. The encoded result would be:

```
a9059cbb000000000000000000000000deaddeaddea \
ddeaddeaddeaddeaddeaddeaddead0000000000000
0000000000000000000000000000000056bc75e2d63100000
```

The first 4 bytes (`a9059cbb`) are the `transfer` function signature/selector, the next 32 bytes are the address, and the final 32 bytes represent the `uint256` number of tokens. Notice that the hex `56bc75e2d63100000` at the end corresponds to 100 tokens (with 18 decimal places, as specified by the `REP` token contract).

Let us now look at what would happen if one were to send an address that was missing 1 byte (2 hex digits). Specifically, let's say an attacker sends `0xdeaddeaddeaddeaddeaddeaddeaddeaddeadde` as an address (missing the last two digits) and the same `100` tokens to withdraw. If the exchange does not validate this input, it will get encoded as:

```
a9059cbb000000000000000000000000deaddeaddea \
ddeaddeaddeaddeaddeaddeadde0000000000000000
0000000000000000000000000000000000000000000056bc75e2d6310000000
```

The difference is subtle. Note that `00` has been added to the end of the encoding, to make up for the short address that was sent. When this gets sent to the smart contract, the `address` parameters will be read as `0xdeaddeaddeaddeaddeaddeaddeaddeaddeadde00` and the value will be read as `56bc75e2d6310000000` (notice the two extra 0s). This value is now `25600` tokens (the value has been multiplied by `256`). In this example, if the exchange held this many tokens, the user would withdraw `25600` tokens (while the exchange thinks the user is only withdrawing `100`) to the modified address. Obviously the attacker won't possess the modified address in this example, but if the attacker were to generate any address that ended in 0s (which can be easily brute-forced) and used this generated address, they could steal tokens from the unsuspecting exchange.

Preventative Techniques

All input parameters in external applications should be validated before sending them to the blockchain. It should also be noted that parameter ordering plays an important role here. As padding only occurs at the end, careful ordering of parameters in the smart contract can mitigate some forms of this attack.

Unchecked CALL Return Values

There are a number of ways of performing external calls in Solidity. Sending ether to external accounts is commonly performed via the `transfer` method. However, the `send` function can also be used, and for more versatile external calls the `CALL` opcode can be directly employed in Solidity. The `call` and `send` functions return a Boolean indicating whether the call succeeded or failed. Thus, these functions have a simple caveat, in that the transaction that executes these functions will not revert if the external call (intialized by `call` or `send`) fails; rather, the functions will simply return `false`. A common error is that the developer expects a revert to occur if the external call fails, and does not check the return value.

For further reading, see #4 on the DASP Top 10 of 2018 and “Scanning Live Ethereum Contracts for the ‘Unchecked-Send’ Bug”.

The Vulnerability

Consider the following example:

```
contract Lotto {
```

```
    bool public payedOut = false;
    address public winner;
    uint public winAmount;

    // ... extra functionality here

    function sendToWinner() public {
        require(!payedOut);
        winner.send(winAmount);
        payedOut = true;
    }

    function withdrawLeftOver() public {
        require(payedOut);
        msg.sender.send(this.balance);
    }
}
```

This represents a Lotto-like contract, where a `winner` receives `winAmount` of ether, which typically leaves a little left over for anyone to withdraw.

The vulnerability exists on line 11, where a `send` is used without checking the response. In this trivial example, a `winner` whose transaction fails (either by running out of gas or by being a contract that intentionally throws in the fallback function) allows `payedOut` to be set to `true` regardless of whether ether was sent or not. In this case, anyone can withdraw the `winner`'s winnings via the `withdrawLeftOver` function.

Preventative Techniques

Whenever possible, use the `transfer` function rather than `send`, as `transfer` will revert if the external transaction reverts. If `send` is required, always check the return value.

A more robust recommendation is to adopt a *withdrawal pattern*. In this solution, each user must call an isolated withdraw function that handles the sending of ether out of the contract and deals with the consequences of failed send transactions. The idea is to logically isolate the external send functionality from the rest of the codebase, and place the burden of a potentially failed transaction on the end user calling the withdraw function.

Real-World Example: Etherpot and King of the Ether

Etherpot wasa smart contract lottery, not too dissimilar to the example contract mentioned earlier. The downfall of this contract was primarily due to incorrect use of block hashes (only the last 256 block hashes are usable; see Aakil Fernandes's post about how Etherpot failed to take account of this correctly). However, this contract also suffered from an unchecked call value. Consider the function `cash` in lotto.sol: Code snippet.

Example 9. lotto.sol: Code snippet

```
...
```

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

         var subpotsCount = getSubpotsCount(roundIndex);

         if(subpotIndex>=subpotsCount)
             return;

         var decisionBlockNumber = getDecisionBlockNumber(roundIndex,subpotIndex);

         if(decisionBlockNumber>block.number)
             return;

         if(rounds[roundIndex].isCashed[subpotIndex])
             return;
         //Subpots can only be cashed once. This is to prevent double payouts

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

         winner.send(subpot);

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

Notice that on line 21 the `send` function's return value is not checked, and the following line then sets a Boolean indicating that the winner has been sent their funds. This bug can allow a state where the winner does not receive their ether, but the state of the contract can indicate that the winner has already been paid.

A more serious version of this bug occurred in the King of the Ether. An excellent post-mortem of this contract has been written that details how an unchecked failed `send` could be used to attack the contract.

Race Conditions/Front Running

Thecombination of external calls to other contracts and the multiuser nature of the underlying blockchain gives rise to a variety of potential Solidity pitfalls whereby users *race* code execution to obtain unexpected states. Reentrancy (discussed earlier in this chapter) is one example of such a race condition. In this section we will discuss other kinds of race conditions that can occur on the Ethereum blockchain. There are a variety of good posts on this subject, including "Race Conditions" on the Ethereum Wiki, #7 on the DASP Top10 of 2018, and the Ethereum Smart Contract Best Practices.

The Vulnerability

As with most blockchains, Ethereum nodes pool transactions and form them into blocks. The

transactions are only considered valid once a miner has solved a consensus mechanism (currently Ethash PoW for Ethereum). The miner who solves the block also chooses which transactions from the pool will be included in the block, typically ordered by the `gasPrice` of each transaction. Here is a potential attack vector. An attacker can watch the transaction pool for transactions that may contain solutions to problems, and modify or revoke the solver's permissions or change state in a contract detrimentally to the solver. The attacker can then get the data from this transaction and create a transaction of their own with a higher `gasPrice` so their transaction is included in a block before the original.

Let's see how this could work with a simple example. Consider the contract shown in FindThisHash.sol.

Example 10. FindThisHash.sol

```
contract FindThisHash {
    bytes32 constant public hash =
      0xb5b5b97fafd9855eec9b41f74dfb6c38f5951141f9a3ecd7f44d5479b630ee0a;

    constructor() external payable {} // load with ether

    function solve(string solution) public {
        // If you can find the pre-image of the hash, receive 1000 ether
        require(hash == sha3(solution));
        msg.sender.transfer(1000 ether);
    }
}
```

Say this contract contains 1,000 ether. The user who can find the preimage of the following SHA-3 hash:

```
0xb5b5b97fafd9855eec9b41f74dfb6c38f5951141f9a3ecd7f44d5479b630ee0a
```

can submit the solution and retrieve the 1,000 ether. Let's say one user figures out the solution is `Ethereum!`. They call `solve` with `Ethereum!` as the parameter. Unfortunately, an attacker has been clever enough to watch the transaction pool for anyone submitting a solution. They see this solution, check its validity, and then submit an equivalent transaction with a much higher `gasPrice` than the original transaction. The miner who solves the block will likely give the attacker preference due to the higher `gasPrice`, and mine their transaction before the original solver's. The attacker will take the 1,000 ether, and the user who solved the problem will get nothing. Keep in mind that in this type of "front-running" vulnerability, miners are uniquely incentivized to run the attacks themselves (or can be bribed to run these attacks with extravagant fees). The possibility of the attacker being a miner themselves should not be underestimated.

Preventative Techniques

There are two classes of actor who can perform these kinds of front-running attacks: users (who

modify the `gasPrice` of their transactions) and miners themselves (who can reorder the transactions in a block how they see fit). A contract that is vulnerable to the first class (users) is significantly worse off than one vulnerable to the second (miners), as miners can only perform the attack when they solve a block, which is unlikely for any individual miner targeting a specific block. Here we'll list a few mitigation measures relative to both classes of attackers.

One method is to place an upper bound on the `gasPrice`. This prevents users from increasing the `gasPrice` and getting preferential transaction ordering beyond the upper bound. This measure only guards against the first class of attackers (arbitrary users). Miners in this scenario can still attack the contract, as they can order the transactions in their block however they like, regardless of gas price.

A more robust method is to use a commit–reveal scheme. Such a scheme dictates that users send transactions with hidden information (typically a hash). After the transaction has been included in a block, the user sends a transaction revealing the data that was sent (the reveal phase). This method prevents both miners and users from front-running transactions, as they cannot determine the contents of the transaction. This method, however, cannot conceal the transaction value (which in some cases is the valuable information that needs to be hidden). The ENS smart contract allowed users to send transactions whose committed data included the amount of ether they were willing to spend. Users could then send transactions of arbitrary value. During the reveal phase, users were refunded the difference between the amount sent in the transaction and the amount they were willing to spend.

A further suggestion byLorenz Breidenbach, Phil Daian, Ari Juels, and Florian Tramèr is to use "submarine sends". An efficient implementation of this idea requires the `CREATE2` opcode, which currently hasn't been adopted but seems likely to be in upcoming hard forks.

Real-World Examples: ERC20 and Bancor

The ERC20 standard is quite well-known for building tokens on Ethereum. This standard has a potential front-running vulnerability that comes about due to the `approve` function. Mikhail Vladimirov and Dmitry Khovratovich have written a good explanation of this vulnerability (and ways to mitigate the attack).

The standard specifies the `approve` function as:

```
function approve(address _spender, uint256 _value) returns (bool success)
```

This function allows a user to permit other users to transfer tokens on their behalf. The front-running vulnerability occurs in the scenario where a user Alice *approves* her friend Bob to spend 100 tokens. Alice later decides that she wants to revoke Bob's approval to spend, say, 100 tokens, so she creates a transaction that sets Bob's allocation to 50 tokens. Bob, who has been carefully watching the chain, sees this transaction and builds a transaction of his own spending the 100 tokens. He puts a higher `gasPrice` on his transaction than Alice's, so gets his transaction prioritized over hers. Some implementations of `approve` would allow Bob to transfer his 100 tokens and then, when Alice's transaction is committed, reset Bob's approval to 50 tokens, in effect giving Bob access to 150 tokens.

Another prominent real-world example is Bancor. Ivan Bogatyy and his team documented a profitable attack on the initial Bancor implementation. His blog post and DevCon3 talk discuss in detail how this was done. Essentially, prices of tokens are determined based on transaction value; users can watch the transaction pool for Bancor transactions and front-run them to profit from the price differences. This attack has been addressed by the Bancor team.

Denial of Service (DoS)

This category is very broad, but fundamentally consists of attacks where users can render a contract inoperable for a period of time, or in some cases permanently. This can trap ether in these contracts forever, as was the case in Real-World Example: Parity Multisig Wallet (Second Hack).

The Vulnerability

There are various ways a contract can become inoperable. Here we highlight just a few less-obvious Solidity coding patterns that can lead to DoS vulnerabilities:

Looping through externally manipulated mappings or arrays

This pattern typically appears when an owner wishes to distribute tokens to investors with a `distribute`-like function, as in this example contract:

```
contract DistributeTokens {
    address public owner; // gets set somewhere
    address[] investors; // array of investors
    uint[] investorTokens; // the amount of tokens each investor gets

    // ... extra functionality, including transfertoken()

    function invest() external payable {
        investors.push(msg.sender);
        investorTokens.push(msg.value * 5); // 5 times the wei sent
        }

    function distribute() public {
        require(msg.sender == owner); // only owner
        for(uint i = 0; i < investors.length; i++) {
            // here transferToken(to,amount) transfers "amount" of
            // tokens to the address "to"
            transferToken(investors[i],investorTokens[i]);
        }
    }
}
```

Notice that the loop in this contract runs over an array that can be artificially inflated. An attacker can create many user accounts, making the `investor` array large. In principle this can be done such that the gas required to execute the for loop exceeds the block gas limit, essentially making the `distribute` function inoperable.

Owner operations

Another common pattern is where owners have specific privileges in contracts and must perform some task in order for the contract to proceed to the next state. One example would be an Initial Coin Offering (ICO) contract that requires the owner to `finalize` the contract, which then allows tokens to be transferable. For example:

```
bool public isFinalized = false;
address public owner; // gets set somewhere

function finalize() public {
    require(msg.sender == owner);
    isFinalized = true;
}

// ... extra ICO functionality

// overloaded transfer function
function transfer(address _to, uint _value) returns (bool) {
    require(isFinalized);
    super.transfer(_to,_value)
}

...
```

In such cases, if the privileged user loses their private keys or becomes inactive, the entire token contract becomes inoperable. In this case, if the owner cannot call `finalize` no tokens can be transferred; the entire operation of the token ecosystem hinges on a single address.

Progressing state based on external calls

Contracts are sometimes written such that progressing to a new state requires sending ether to an address, or waiting for some input from an external source. These patterns can lead to DoS attacks when the external call fails or is prevented for external reasons. In the example of sending ether, a user can create a contract that does not accept ether. If a contract requires ether to be withdrawn in order to progress to a new state (consider a time-locking contract that requires all ether to be withdrawn before being usable again), the contract will never achieve the new state, as ether can never be sent to the user's contract that does not accept ether.

Preventative Techniques

In the first example, contracts should not loop through data structures that can be artificially manipulated by external users. A withdrawal pattern is recommended, whereby each of the investors call a withdraw function to claim tokens independently.

In the second example, a privileged user was required to change the state of the contract. In such examples a failsafe can be used in the event that the owner becomes incapacitated. One solution is to make the owner a multisig contract. Another solution is to use a time-lock: in the example given the require on line 5 could include a time-based mechanism, such as `require(msg.sender == owner || now > unlockTime)`, that allows any user to finalize after a period of time specified by `unlockTime`.

This kind of mitigation technique can be used in the third example also. If external calls are required to progress to a new state, account for their possible failure and potentially add a time-based state progression in the event that the desired call never comes.

NOTE	Of course, there are centralized alternatives to these suggestions: one can add a `maintenanceUser` who can come along and fix problems with DoS-based attack vectors if need be. Typically these kinds of contracts have trust issues, because of the power of such an entity.

Real-World Examples: GovernMental

GovernMentalwas an old Ponzi scheme that accumulated quite a large amount of ether (1,100 ether, at one point). Unfortunately, it was susceptible to the DoS vulnerabilities mentioned in this section. A Reddit post by etherik describes how the contract required the deletion of a large mapping in order to withdraw the ether. The deletion of this mapping had a gas cost that exceeded the block gas limit at the time, and thus it was not possible to withdraw the 1,100 ether. The contract address is 0xF45717552f12Ef7cb65e95476F217Ea008167Ae3, and you can see from transaction 0x0d80d67202bd9cb6773df8dd2020e719 0a1b0793e8ec4fc105257e8128f0506b that the 1,100 ether were finally obtained with a transaction that used 2.5M gas (when the block gas limit had risen enough to allow such a transaction).

Block Timestamp Manipulation

Block timestamps have historically been used for a variety of applications, such as entropy for random numbers (see the Entropy Illusion for further details), locking funds for periods of time, and various state-changing conditional statements that are time-dependent. Miners have the ability to adjust timestamps slightly, which can prove to be dangerous if block timestamps are used incorrectly in smart contracts.

Useful references for this include the Solidity docs and Joris Bontje's Ethereum Stack Exchange question on the topic.

The Vulnerability

`block.timestamp` and its alias `now` can be manipulated by miners if they have some incentive to do so. Let's construct a simple game, shown in roulette.sol, that would be vulnerable to miner exploitation.

Example 11. roulette.sol

```
contract Roulette {
    uint public pastBlockTime; // forces one bet per block

    constructor() external payable {} // initially fund contract

    // fallback function used to make a bet
    function () external payable {
        require(msg.value == 10 ether); // must send 10 ether to play
```

```
        require(now != pastBlockTime); // only 1 transaction per block
        pastBlockTime = now;
        if(now % 15 == 0) { // winner
            msg.sender.transfer(this.balance);
        }
    }
}
```

This contract behaves like a simple lottery. One transaction per block can bet 10 ether for a chance to win the balance of the contract. The assumption here is that `block.timestamp's last two digits are uniformly distributed. If that were the case, there would be a 1 in 15 chance of winning this lottery.

However, as we know, miners can adjust the timestamp should they need to. In this particular case, if enough ether pools in the contract, a miner who solves a block is incentivized to choose a timestamp such that `block.timestamp` or `now` modulo 15 is `0`. In doing so they may win the ether locked in this contract along with the block reward. As there is only one person allowed to bet per block, this is also vulnerable to front-running attacks (see Race Conditions/Front Running for further details).

In practice, block timestamps are monotonically increasing and so miners cannot choose arbitrary block timestamps (they must be later than their predecessors). They are also limited to setting block times not too far in the future, as these blocks will likely be rejected by the network (nodes will not validate blocks whose timestamps are in the future).

Preventative Techniques

Block timestamps should not be used for entropy or generating random numbers—i.e., they should not be the deciding factor (either directly or through some derivation) for winning a game or changing an important state.

Time-sensitive logic is sometimes required; e.g., for unlocking contracts (time-locking), completing an ICO after a few weeks, or enforcing expiry dates. It is sometimes recommended to use `block.number` and an average block time to estimate times; with a `10 second` block time, `1 week` equates to approximately, `60480 blocks`. Thus, specifying a block number at which to change a contract state can be more secure, as miners are unable to easily manipulate the block number. The BAT ICO contract employed this strategy.

This can be unnecessary if contracts aren't particularly concerned with miner manipulations of the block timestamp, but it is something to be aware of when developing contracts.

Real-World Example: GovernMental

GovernMental,the old Ponzi scheme mentioned above, was also vulnerable to a timestamp-based attack. The contract paid out to the player who was the last player to join (for at least one minute) in a round. Thus, a miner who was a player could adjust the timestamp (to a future time, to make it look like a minute had elapsed) to make it appear that they were the last player to join for over a minute (even though this was not true in reality). More detail on this can be found in the "History

of Ethereum Security Vulnerabilities, Hacks and Their Fixes" post by Tanya Bahrynovska.

Constructors with Care

Constructors are special functions that often perform critical, privileged tasks when initializing contracts. Before Solidity v0.4.22, constructors were defined as functions that had the same name as the contract that contained them. In such cases, when the contract name is changed in development, if the constructor name isn't changed too it becomes a normal, callable function. As you can imagine, this can lead (and has) to some interesting contract hacks.

For further insight, the reader may be interested in attempting the Ethernaut challenges (in particular the Fallout level).

The Vulnerability

If the contract name is modified, or there is a typo in the constructor's name such that it does not match the name of the contract, the constructor will behave like a normal function. This can lead to dire consequences, especially if the constructor performs privileged operations. Consider the following contract:

```
contract OwnerWallet {
    address public owner;

    // constructor
    function ownerWallet(address _owner) public {
        owner = _owner;
    }

    // Fallback. Collect ether.
    function () payable {}

    function withdraw() public {
        require(msg.sender == owner);
        msg.sender.transfer(this.balance);
    }
}
```

This contract collects ether and allows only the owner to withdraw it, by calling the `withdraw` function. The issue arises because the constructor is not named exactly the same as the contract: the first letter is different! Thus, any user can call the `ownerWallet` function, set themselves as the owner, and then take all the ether in the contract by calling `withdraw`.

Preventative Techniques

This issue has been addressed in version 0.4.22 of the Solidity compiler. This version introduced a `constructor` keyword that specifies the constructor, rather than requiring the name of the function to match the contract name. Using this keyword to specify constructors is recommended to prevent naming issues.

Real-World Example: Rubixi

Rubixiwas another pyramid scheme that exhibited this kind of vulnerability. It was originally called `DynamicPyramid`, but the contract name was changed before deployment to `Rubixi`. The constructor's name wasn't changed, allowing any user to become the creator. Some interesting discussion related to this bug can be found on Bitcointalk. Ultimately, it allowed users to fight for creator status to claim the fees from the pyramid scheme. More detail on this particular bug can be found in "History of Ethereum Security Vulnerabilities, Hacks and Their Fixes".

Uninitialized Storage Pointers

The EVM stores data either as storage or as memory. Understanding exactly how this is done and the default types for local variables of functions is highly recommended when developing contracts. This is because it is possible to produce vulnerable contracts by inappropriately initializing variables.

To read more about storage and memory in the EVM, see the Solidity documentation on data location, layout of state variables in storage, and layout in memory.

NOTE	This section is based on an excellent post by Stefan Beyer. Further reading on this topic, inspired by Stefan, can be found in this Reddit thread.

The Vulnerability

Local variables within functions default to storage or memory depending on their type. Uninitialized local storage variables may contain the value of other storage variables in the contract; this fact can cause unintentional vulnerabilities, or be exploited deliberately.

Let's consider the relatively simple name registrar contract in NameRegistrar.sol.

Example 12. NameRegistrar.sol

```
// A locked name registrar
contract NameRegistrar {

    bool public unlocked = false;  // registrar locked, no name updates

    struct NameRecord { // map hashes to addresses
        bytes32 name;
        address mappedAddress;
    }

    // records who registered names
    mapping(address => NameRecord) public registeredNameRecord;
    // resolves hashes to addresses
    mapping(bytes32 => address) public resolve;

    function register(bytes32 _name, address _mappedAddress) public {
        // set up the new NameRecord
```

```
        NameRecord newRecord;
        newRecord.name = _name;
        newRecord.mappedAddress = _mappedAddress;

        resolve[_name] = _mappedAddress;
        registeredNameRecord[msg.sender] = newRecord;

        require(unlocked); // only allow registrations if contract is unlocked
    }
}
```

This simple name registrar has only one function. When the contract is `unlocked`, it allows anyone to register a name (as a `bytes32` hash) and map that name to an address. The registrar is initially locked, and the `require` on line 25 prevents `register` from adding name records. It seems that the contract is unusable, as there is no way to unlock the registry! There is, however, a vulnerability that allows name registration regardless of the `unlocked` variable.

To discuss this vulnerability, first we need to understand how storage works in Solidity. As a high-level overview (without any proper technical detail—we suggest reading the Solidity docs for a proper review), state variables are stored sequentially in *slots* as they appear in the contract (they can be grouped together but aren't in this example, so we won't worry about that). Thus, `unlocked` exists in `slot[0]`, `registeredNameRecord` in `slot[1]`, and `resolve` in `slot[2]`, etc. Each of these slots is 32 bytes in size (there are added complexities with mappings, which we'll ignore for now). The Boolean `unlocked` will look like `0x000⋯0` (64 0s, excluding the `0x`) for `false` or `0x000⋯1` (63 0s) for `true`. As you can see, there is a significant waste of storage in this particular example.

The next piece of the puzzle is that Solidity by default puts complex data types, such as structs, in storage when initializing them as local variables. Therefore, `newRecord` on line 18 defaults to storage. The vulnerability is caused by the fact that `newRecord` is not initialized. Because it defaults to storage, it is mapped to storage slot[0], which currently contains a pointer to `unlocked`. Notice that on lines 19 and 20 we then set `newRecord.name` to `_name` and `newRecord.mappedAddress` to `_mappedAddress`; this updates the storage locations of slot[0] and slot[1], which modifies both `unlocked` and the storage slot associated with `registeredNameRecord`.

This means that `unlocked` can be directly modified, simply by the `bytes32 _name` parameter of the `register` function. Therefore, if the last byte of `_name` is nonzero, it will modify the last byte of storage `slot[0]` and directly change `unlocked` to `true`. Such `_name` values will cause the `require` call on line 25 to succeed, as we have set `unlocked` to `true`. Try this in Remix. Note the function will pass if you use a `_name` of the form:

```
0x0000000000000000000000000000000000000000000000000000000000000001
```

Preventative Techniques

The Solidity compiler shows a warning for unintialized storage variables; developers should pay careful attention to these warnings when building smart contracts. The current version of Mist (0.10) doesn't allow these contracts to be compiled. It is often good practice to explicitly use the

`memory` or `storage` specifiers when dealing with complex types, to ensure they behave as expected.

Real-World Examples: OpenAddressLottery and CryptoRoulette Honey Pots

A honey pot named OpenAddressLottery was deployed that used this uninitialized storage variable quirk to collect ether from some would-be hackers. The contract is rather involved, so we will leave the analysis to the Reddit thread where the attack is quite clearly explained.

Another honey pot, CryptoRoulette, also utilized this trick to try and collect some ether. If you can't figure out how the attack works, see "An Analysis of a Couple Ethereum Honeypot Contracts" for an overview of this contract and others.

Floating Point and Precision

As of this writing (v0.4.24), Solidity does not support fixed-point and floating-point numbers. This means that floating-point representations must be constructed with integer types in Solidity. This can lead to errors and vulnerabilities if not implemented correctly.

NOTE	For further reading, see the Ethereum Contract Security Techniques and Tips wiki.

The Vulnerability

As there is no fixed-point type in Solidity, developers are required to implement their own using the standard integer data types. There are a number of pitfalls developers can run into during this process. We will try to highlight some of these in this section.

Let's begin with a code example (we'll ignore over/underflow issues, discussed earlier in this chapter, for simplicity):

```
contract FunWithNumbers {
    uint constant public tokensPerEth = 10;
    uint constant public weiPerEth = 1e18;
    mapping(address => uint) public balances;

    function buyTokens() external payable {
        // convert wei to eth, then multiply by token rate
        uint tokens = msg.value/weiPerEth*tokensPerEth;
        balances[msg.sender] += tokens;
    }

    function sellTokens(uint tokens) public {
        require(balances[msg.sender] >= tokens);
        uint eth = tokens/tokensPerEth;
        balances[msg.sender] -= tokens;
        msg.sender.transfer(eth*weiPerEth);
    }
}
```

This simple token buying/selling contract has some obvious problems. Although the mathematical calculations for buying and selling tokens are correct, the lack of floating-point numbers will give erroneous results. For example, when buying tokens on line 8, if the value is less than `1 ether` the initial division will result in `0`, leaving the result of the final multiplication as `0` (e.g., `200 wei` divided by `1e18 weiPerEth` equals `0`). Similarly, when selling tokens, any number of tokens less than `10` will also result in `0 ether`. In fact, rounding here is always down, so selling `29 tokens` will result in `2 ether`.

The issue with this contract is that the precision is only to the nearest ether (i.e., 1e18 wei). This can get tricky when dealing with decimals in ERC20 tokens when you need higher precision.

Preventative Techniques

Keeping the right precision in your smart contracts is very important, especially when dealing with ratios and rates that reflect economic decisions.

You should ensure that any ratios or rates you are using allow for large numerators in fractions. For example, we used the rate `tokensPerEth` in our example. It would have been better to use `weiPerTokens`, which would be a large number. To calculate the corresponding number of tokens we could do `msg.value/weiPerTokens`. This would give a more precise result.

Another tactic to keep in mind is to be mindful of order of operations. In our example, the calculation to purchase tokens was `msg.value/weiPerEth*tokenPerEth`. Notice that the division occurs before the multiplication. (Solidity, unlike some languages, guarantees to perform operations in the order in which they are written.) This example would have achieved a greater precision if the calculation performed the multiplication first and then the division; i.e., `msg.value*tokenPerEth/weiPerEth`.

Finally, when defining arbitrary precision for numbers it can be a good idea to convert values to higher precision, perform all mathematical operations, then finally convert back down to the precision required for output. Typically uint256s are used (as they are optimal for gas usage); these give approximately 60 orders of magnitude in their range, some of which can be dedicated to the precision of mathematical operations. It may be the case that it is better to keep all variables in high precision in Solidity and convert back to lower precisions in external apps (this is essentially how the `decimals` variable works in ERC20 token contracts). To see an example of how this can be done, we recommend looking at DS-Math. It uses some funky naming (“wads” and “rays”), but the concept is useful.

Real-World Example: Ethstick

The Ethstick contract does not use extended precision; however, it deals with wei. So, this contract will have issues of rounding, but only at the wei level of precision. It has some more serious flaws, but these relate back to the difficulty in getting entropy on the blockchain (see Entropy Illusion). For a further discussion of the Ethstick contract, we’ll refer you to another post by Peter Vessenes, “Ethereum Contracts Are Going to Be Candy for Hackers”.

Tx.Origin Authentication

Solidity has a global variable, `tx.origin`, which traverses the entire call stack and contains the address of the account that originally sent the call (or transaction). Using this variable for authentication in a smart contract leaves the contract vulnerable to a phishing-like attack.

NOTE	For further reading, see dbryson's Ethereum Stack Exchange question, "Tx.Origin and Ethereum Oh My!" by Peter Vessenes, and "Solidity: Tx Origin Attacks" by Chris Coverdale.

The Vulnerability

Contracts that authorize users using the `tx.origin` variable are typically vulnerable to phishing attacks that can trick users into performing authenticated actions on the vulnerable contract.

Consider the simple contract in Phishable.sol.

Example 13. Phishable.sol

```
contract Phishable {
    address public owner;

    constructor (address _owner) {
        owner = _owner;
    }

    function () external payable {} // collect ether

    function withdrawAll(address _recipient) public {
        require(tx.origin == owner);
        _recipient.transfer(this.balance);
    }
}
```

Notice that on line 11 the contract authorizes the `withdrawAll` function using `tx.origin`. This contract allows for an attacker to create an attacking contract of the form:

```
import "Phishable.sol";

contract AttackContract {

    Phishable phishableContract;
    address attacker; // The attacker's address to receive funds

    constructor (Phishable _phishableContract, address _attackerAddress) {
        phishableContract = _phishableContract;
        attacker = _attackerAddress;
```

```
    }

    function () payable {
        phishableContract.withdrawAll(attacker);
    }
}
```

The attacker might disguise this contract as their own private address and socially engineer the victim (the owner of the Phishable contract) to send some form of transaction to the address—perhaps sending this contract some amount of ether. The victim, unless careful, may not notice that there is code at the attacker's address, or the attacker might pass it off as being a multisignature wallet or some advanced storage wallet (remember that the source code of public contracts is not available by default).

In any case, if the victim sends a transaction with enough gas to the `AttackContract` address, it will invoke the fallback function, which in turn calls the `withdrawAll` function of the `Phishable` contract with the parameter `attacker`. This will result in the withdrawal of all funds from the `Phishable` contract to the `attacker` address. This is because the address that first initialized the call was the victim (i.e., the owner of the `Phishable` contract). Therefore, `tx.origin` will be equal to `owner` and the `require` on line 11 of the `Phishable` contract will pass.

Preventative Techniques

`tx.origin` should not be used for authorization in smart contracts. This isn't to say that the `tx.origin` variable should never be used. It does have some legitimate use cases in smart contracts. For example, if one wanted to deny external contracts from calling the current contract, one could implement a `require` of the form `require(tx.origin == msg.sender)`. This prevents intermediate contracts being used to call the current contract, limiting the contract to regular codeless addresses.

Contract Libraries

There is a lot of existing code available for reuse, both deployed on-chain as callable libraries and off-chain as code template libraries. On-platform libraries, having been deployed, exist as bytecode smart contracts, so great care should be taken before using them in production. However, using well-established existing on-platform libraries comes with many advantages, such as being able to benefit from the latest upgrades, and saves you money and benefits the Ethereum ecosystem by reducing the total number of live contracts in Ethereum.

In Ethereum, the most widely used resource is the OpenZeppelin suite, an ample library of contracts ranging from implementations of ERC20 and ERC721 tokens, to many flavors of crowdsale models, to simple behaviors commonly found in contracts, such as `Ownable`, `Pausable`, or `LimitBalance`. The contracts in this repository have been extensively tested and in some cases even function as *de facto* standard implementations. They are free to use, and are built and maintained by Zeppelin together with an ever-growing list of external contributors.

Also from Zeppelin is ZeppelinOS, an open source platform of services and tools to develop and manage smart contract applications securely. ZeppelinOS provides a layer on top of the EVM that makes it easy for developers to launch upgradeable DApps linked to an on-chain library of well-

tested contracts that are themselves upgradeable. Different versions of these libraries can coexist on the Ethereum platform, and a vouching system allows users to propose or push improvements in different directions. A set of off-chain tools to debug, test, deploy, and monitor decentralized applications is also provided by the platform.

The project ethpm aims to organize the various resources that are developing in the ecosystem by providing a package management system. As such, their registry provides more examples for you to browse:

- Website: https://www.ethpm.com/
- Repository link: https://www.ethpm.com/registry
- GitHub link: https://github.com/ethpm
- Documentation: https://www.ethpm.com/docs/integration-guide

Conclusions

There is a lot for any developer working in the smart contract domain to know and understand. By following best practices in your smart contract design and code writing, you will avoid many severe pitfalls and traps.

Perhaps the most fundamental software security principle is to maximize reuse of trusted code. In cryptography, this is so important it has been condensed into an adage: "Don’t roll your own crypto." In the case of smart contracts, this amounts to gaining as much as possible from freely available libraries that have been thoroughly vetted by the community.