Jeffrey Scholz · May 5 · 39 min read

Smart Contract Security

Updated: Sep 8

This article serves as a mini course on smart contract security and provides an extensive list of the issues and vulnerabilities that tend to recur in Solidity smart contracts.

A security issue in Solidity boils down to smart contracts not behaving the way they were intended to. This can fall into four broad categories:

- Funds getting stolen
- Funds getting locked up or frozen inside a contract
- People receive less rewards than anticipated (rewards are delayed or reduced)
- People receive more rewards than anticipated (leading to inflation and devaluation)

It isn't possible to make a comprehensive list of everything that can go wrong. However, just as traditional software engineering has common themes of vulnerabilities such as SQL injection, buffer overruns, and cross site scripting, smart contracts have recurring anti-patterns that can be documented.

Smart contract hacks and vulnerabilities

Think of this guide as more of a reference. It isn't possible to discuss every concept in detail without turning this into a book (fair warning: this article is 10k+ words long, so feel free to bookmark it and read it in chunks). However, it serves as a list of what to look out for and what to study. If a topic feels

unfamiliar, that should serve as an indicator that it is worth putting time into practicing identifying that class of vulnerability.

Prerequisites

This article assumes basic proficiency in Solidity. If you are new to Solidity, please see our free Solidity tutorial.

Reentrancy

We've written extensively on smart contract reentrancy so we won't repeat it here. But here is a quick summary:

Whenever a smart contract calls the function of another smart contract, sends Ether to it, or transfers a token to it, then there is a possibility of re-entrancy.

- When Ether is transferred, the receiving contract's fallback or receive function is called. This hands control over to the receiver.
- Some token protocols alert the receiving smart contract that they have received the token by calling a predetermined function. This hands the control flow over to that function.
- When an attacking contract receives control, it doesn't have to call the same function that handed over control. It could call a different function in the victim smart contract (cross-function reentrancy) or even a different contract (cross-contract reentrancy)
- Read-only reentrancy happens when a view function is accessed while the contract is in an intermediate state.

Despite reentrancy likely being the most well known smart contract vulnerability, it only makes up a small percentage of hacks that happen in the wild. Security researcher Pascal Caversaccio (pcaveraccio) keeps an up-to-date github list of reentrancy attacks. As of April 2023, 46 reentrancy attacks have been documented in that repository.

Access Control

It seems like a simple mistake, but forgetting to place restrictions on who can call a sensitive function (like withdrawing ether or changing ownership) happens surprisingly often.

Even if a modifier is in place, there have been cases where the modifier was not implemented correctly, such as in the example below where the require statement is missing.

```
// DO NOT USE!
modifier onlyMinter {
    minters[msg.sender] == true_;
}
```

This above code is a real example from this audit: https://code4rena.com/reports/2023-01-rabbithole/#h-01-bad-implementation-in-minter-access-control-for-rabbitholereceipt-and-rabbitholetickets-contracts

Here is another way access control can go wrong

```
function claimAirdrop(bytes32 calldata proof[]) {

    bool verified = MerkleProof.verifyCalldata(proof, merkleRoot,
keccak256(abi.encode(msg.sender)));
    require(verified, "not verified");
    require(alreadyClaimed[msg.sender], "already claimed");

    _transfer(msg.sender, AIRDROP_AMOUNT);
}
```

In this case, "alreadyClaimed" is never set to true, so the claimant can issue call the function multiple times.

Real life example: Trader bot exploited

A fairly recent example of insufficient access control was an unprotected function to receive flashloans by a trading bot (which went by the name 0xbad, as the address started with that sequence). It racked up over a million dollars in profit until one day an attacker noticed any address could call the flashloan receive function, not just the flashloan provider.

As is usually the case with trading bots, the smart contract code to execute the trades was not verified, but the attacker discovered the weakness anyway. More info in the rekt news coverage.

Improper Input Validation

If access control is about controlling who calls a function, input validation is about controlling what they call the contract with.

This usually comes down to forgetting to put the proper require statements in place.
Here is a rudimentary example

```
contract UnsafeBank {
    mapping(address => uint256) public balances;

    // allow depositing on other's behalf
    function deposit(address for) public payable {
        balances += msg.value;
    }

    function withdraw(address from, uint256 amount) public {
```

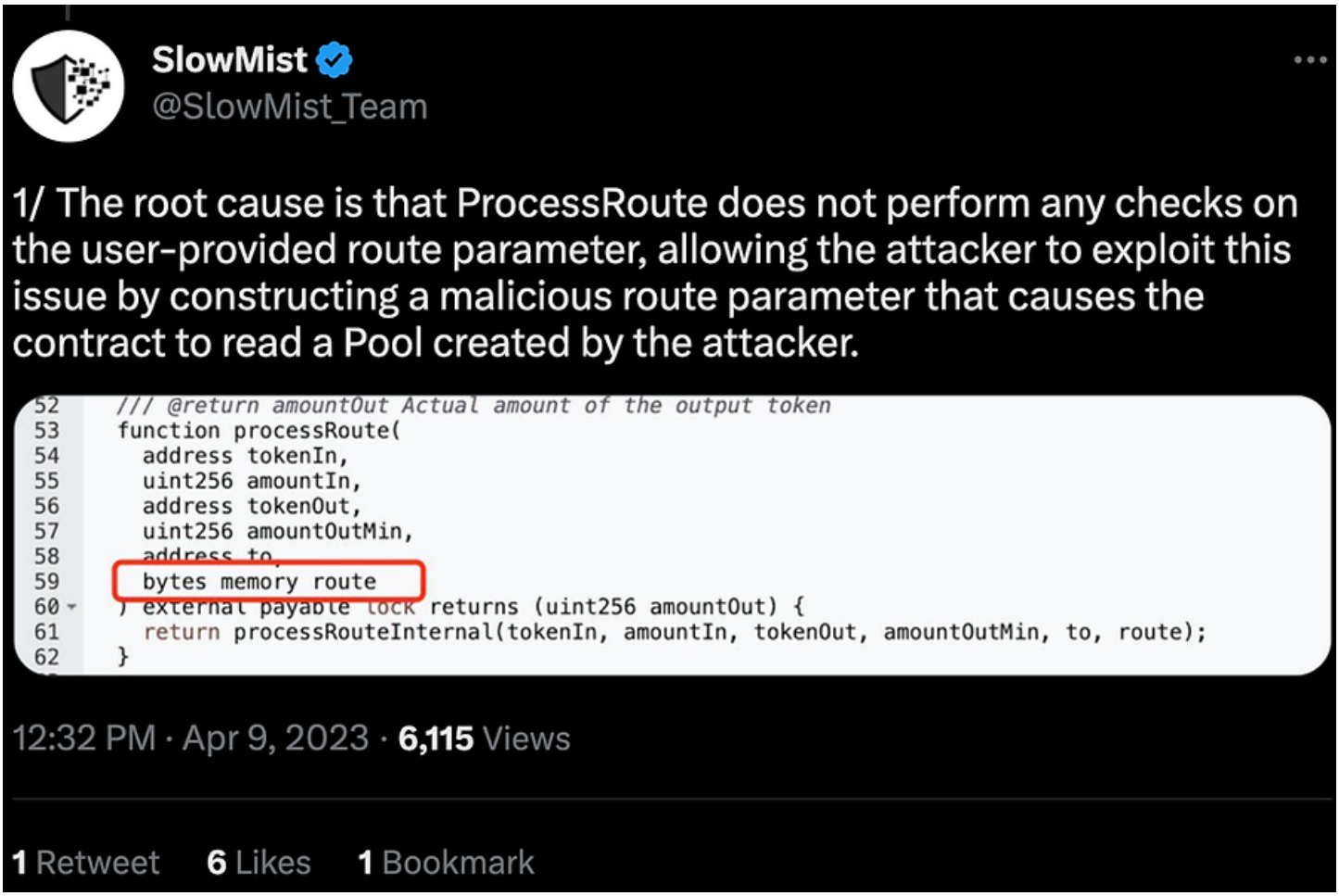
```
        require(balances[from] <= amount, "insufficient balance");

        balances[from] -= amount;
        msg.sender.call{value: amout}("");
    }
}
```

The contract above does check that you aren't withdrawing more than you have in your account, but it doesn't stop you from withdrawing from an arbitrary account.

Real life example: Sushiswap

Sushiswap experienced a hack of this type due to one of the parameters of an external function not being sanitized.



https://twitter.com/peckshield/status/1644907207530774530

What is the difference between improper access control and improper input validation?

Improper access control means msg.sender does not have adequate restrictions. Improper input validation means the arguments to the function are not sufficiently sanitized. There is also an inverse

to this anti-pattern: placing too much restriction on a function call.

Excessive function restriction

Excessive validation probably means funds won't get stolen, but it could mean funds get locked into the contract. Having *too* many safeguards in place is not a good thing either.

Real life example: Akutars NFT

One of the most high-profile incidents was the Akutars NFT which ended up with 34 million dollars worth of Eth stuck inside the smart contract and unwithdrawable.
The contract had a well-intentioned mechanism to prevent the owner of the contract from withdrawing until all refunds from paying above the dutch auction price had been given. But due to a bug documented in the Twitter thread linked below, the owner was unable to withdraw the funds.

https://twitter.com/0xInuarashi/status/1517674505975394304

Getting the balance right

Sushiswap gave too much power to untrusted users, and the Akutars NFT gave too little power to the admin. When designing smart contracts, a subjective judgement about how much liberty each class of users must be made, and this decision cannot be left to automated testing and tooling. There are significant tradeoffs with decentralization, security, and UX that must be considered.

For the smart contract programmer, explicitly writing out what users should and should not be able to do with certain functions is an important part of the development process.

We will revisit the topic of overpowered admins later.

Security often boils down to managing the way money exits the contract

As stated in the introduction, there are four primary ways smart contracts get hacked:

- **Money stolen**
- **Money frozen**
- **Insufficient rewards**
- **Excessive rewards**

"Money" here means anything of value, such as tokens, not just cryptocurrency. When coding or auditing a smart contract, the developer must be conscientious of the intended ways value is to flow in and out of the contract. The issues listed above are the primary ways smart contracts get hacked, but there are a lot of other root causes that can cascade into major issues, which are documented below.

Double voting or msg.sender spoofing

Using vanilla ERC20 tokens or NFTs as tickets to weigh vote is unsafe because attackers can vote with one address, transfer the tokens to another address, and vote again from that address.

Here is a minimal example

```
// A malicious voter can simply transfer their tokens to
// another address and vote again.
contract UnsafeBallot {

    uint256 public proposal1VoteCount;
    uint256 public proposal2VoteCount;

    IERC20 immutable private governanceToken;

    constructor(IERC20 _governanceToken) {
        governanceToken = _governanceToken;
    }

    function voteFor1() external notAlreadyVoted {
        proposal1VoteCount += governanceToken.balanceOf(msg.sender);
```

```
    }

    function voteFor2() external notAlreadyVoted {
        proposal2VoteCount += governanceToken.balanceOf(msg.sender);
    }

    // prevent the same address from voting twice,
    // however the attacker can simply
    // transfer to a new address
    modifier notAlreadyVoted {
        require(!alreadyVoted[msg.sender], "already voted");
        _;
        alreadyVoted[msg.sender] = true;
    }
}
```

To prevent this attack, ERC20 Snapshot or ERC20 Votes should be used. By snapshotting a point of time in the past, the current token balances cannot be manipulated to gain illicit voting power.

Flashloan Governance Attacks

However, using an ERC20 token with a snapshot or vote capability doesn't fully solve the problem if someone can take a flashloan to temporarily increase their balance, then take a snapshot of their balance in the same transaction. If that snapshot is used for voting, they will have an unreasonably large amount of votes at their disposal.

A flashloan lends a large amount of Ether or token to an address, but reverts if the money is not repaid in the same transaction.

```
contract SimpleFlashloan {

    function borrowERC20Tokens() public {
        uint256 before = token.balanceOf(address(this));

        // send tokens to the borrower
        token.transfer(msg.sender, amount);

        // hand control back to the borrower to
        // let them do something
        IBorrower(msg.sender).onFlashLoan();

        // require that the tokens got returned
        require(token.balanceOf(address(this) >= before);
```

```
    }
}
```

An attacker can use a flashloan to suddenly gain a lot of votes to swing proposals in their favor and/or do something malicious.

Flashloan Price Attacks

This is arguably the most common (or at least most high-profile) attack on DeFi, accounting for hundreds of millions of dollars lost. Here is a list of high profile ones.
The price of an asset on the blockchain is often calculated as the current exchange rate between assets. For example, if a contract is currently trading 1 USDC for 100 k9coin, then you could say k9coin has a price of 0.01 USDC. However, prices generally move in response to buying and selling pressure, and flash loans can create massive buying and selling pressure.

When querying another smart contract about the price of an asset, the developer needs to be very careful because they are assuming the smart contract they are calling is immune to flash loan manipulation.

Bypassing the contract check

You can "check" if an address is a smart contract by looking at it's bytecode size. Externally owned accounts (regular wallets) don't have any bytecode. Here are a few ways of doing it

```
import "@openzeppelin/contracts/utils/Address.sol"
contract CheckIfContract {

    using Address for address;

    function addressIsContractV1(address _a) {
        return _a.code.length != 0;
    }

    function addressIsContractV2(address _a) {

        // use the openzeppelin libraryreturn _a.isContract();
    }
}
```

However, this has a few limitations

- **If a contract makes an external call from a constructor, then it's apparent bytecode size will be zero because the smart contract deployment code hasn't returned the runtime code yet**

- The space might be empty now, but an attacker might know they can deploy a smart contract there in the future using create2

In general checking if an address is a contract is usually (but not always) an antipattern. Multisignature wallets are smart contracts themselves, and doing anything that might break multisignature wallets breaks composability.

The exception to this is checking if the target is a smart contract before calling a transfer hook. More on this later.

tx.origin

There is rarely a good reason to use tx.origin. If tx.origin is used to identify the sender, then a man-in-the-middle attack is possible. If the user is tricked into calling a malicious smart contract, then the smart contract can use all the authority tx.origin has to wreak havoc.

Consider this following exercise, and the comments above the code.

```
contract Phish {
    function phishingFunction() public {

        // this fails, because this contract does not have
approval/allowance
        token.transferFrom(msg.sender, address(this),
token.balanceOf(msg.sender));

        // this also fails, because this creates approval for the
contract,// not the wallet calling this phishing function
        token.approve(address(this), type(uint256).max);
    }
}
```

This does not mean you are safe calling arbitrary smart contracts. But there is a layer of safety built into most protocols that will be bypassed if tx.origin is used for authentication.

Sometimes, you might see code that looks like this:

```
require(msg.sender == tx.origin, "no contracts");
```

When a smart contract calls another smart contract, msg.sender will be the smart contract and tx.origin will be the user's wallet, thus giving a reliable indication that the incoming call is from a smart contract. This is true even if the call happens from the constructor.

Most of the time, this design pattern not a good idea. Multisignature wallets and Wallets from EIP 4337 won't be able to interact with a function that has this code. This pattern can commonly be seen in NFT

mints, where it's reasonable to expect most users are using a traditional wallet. But as account abstraction becomes more popular, this pattern will hinder more than it helps.

Gas Griefing or Denial of Service

A griefing attack means the hacker is trying to "cause grief" for other people, even if they don't gain economically from doing so.

A smart contract can maliciously use up all the gas forwarded to it by going into an infinite loop. Consider the following example:

```
contract Mal {

    fallback() external payable {

        // infinite loop uses up all the gas
        while (true) {

        }
    }
}
```

If another contract distributes ether to a list of addresses such as follows:

```
contract Distribute {
    funtion distribute(uint256 total) public nonReentrant {
        for (uint i; i < addresses.length; ) {

            (bool ok, ) addresses.call{value: total / addresses.length}
("");
            // ignore ok, if it reverts we move on
            // traditional gas saving trick for for loops
            unchecked {
                ++i;
            }
        }
    }
}
```

Then the function will revert when it sends ether to Mal. The call in the code above forwards 63 / 64 of the gas available (read more about this rule in our article on EIP 150), so there likely won't be enough gas to complete the operation with only 1/64 of the gas left.

A smart contract can return a large memory array that consumes a lot of gas
Consider the following example

```
function largeReturn() public {

    // result might be extremely long!
    (book ok, bytes memory result) =
        otherContract.call(abi.encodeWithSignature("foo()"));

    require(ok, "call failed");
}
```

Memory arrays use up quadratic amount of gas after 724 bytes, so a carefully chosen return data size can grief the caller.
Even if the variable result is not used, it is still copied to memory. If you want to restrict the return size to a certain amount, you can use assembly

```
function largeReturn() public {
    assembly {
        let ok := call(gas(), destinationAddress, value, dataOffset,
dataSize, 0x00, 0x00);
        // nothing is copied to memory until you
        // use returndatacopy()
    }
}
```

Deleting arrays that others can add to is also an denial of service vector

Although erasing storage is a gas-efficient operation, it still has a net cost. If an array becomes too long, it becomes impossible to delete. Here is a minimal example

```
contract VulnerableArray {

    address[] public stuff;

    function addSomething(address something) public {
        stuff.push(something);
    }

    // if stuff is too long, this will become undeletable due to
    // the gas cost
    function deleteEverything() public onlyOwner {
        delete stuff;
```

```
    }
}
```

ERC777, ERC721, and ERC1155 can also be griefing vectors

If a smart contract transfers tokens that have transfer hooks, an attacker can set up a contract that does not accept the token (it either does not have an onReceive function or programs the function to revert). This will make the token untransferable and cause the entire transaction to revert.

Before using safeTransfer or transfer, consider the possibility that the receiver might force the transaction to revert.

```
contract Mal is IERC721Receiver, IERC1155Receiver, IERC777Receiver {

    // this will intercept any transfer hook
    fallback() external payable {

        // infinite loop uses up all the gaswhile (true) {

        }
    }

    // we could also selectively deny transactions
    function onERC721Received(address operator,
        address from,
        uint256 tokenId,
        bytes calldata data
    ) external returns (bytes4) {

        if (wakeUpChooseViolence()) {
            revert();
        }
        else {
            return IERC721Receiver.onERC721Received.selector;
        }
    }
}
```

Insecure Randomness

It is currently not possible to generate randomness securely with a single transaction on the blockchain. Blockchains need to be fully deterministic, otherwise distributed nodes would not be able

to reach a consensus about the state. Because they are fully deterministic, any "random" number can be predicted. The following dice rolling function can be exploited.

```
contract UnsafeDice {
    function randomness() internal returns (uint256) {
        return keccak256(abi.encode(msg.sender, tx.origin,
block.timestamp, tx.gasprice, blockhash(block.number - 1);
    }

    // our dice can land on one of {0,1,2,3,4,5}function rollDice()
public payable {
        require(msg.value == 1 ether);

        if (randomness() % 6) == 5) {
            msg.sender.call{value: 2 ether}("");
        }
    }
}

contract ExploitDice {
    function randomness() internal returns (uint256) {
        return keccak256(abi.encode(msg.sender, tx.origin,
block.timestamp, tx.gasprice, blockhash(block.number - 1);
    }

    function betSafely(IUnsafeDice game) public payable {
        if (randomness % 6) == 5)) {
            game.betSafely{value: 1 ether}()
        }

        // else don't do anything
    }
}
```

It doesn't matter how you generate randomness because an attacker can replicate it exactly. Throwing in more sources of "entropy" such as the msg.sender, timestamp, etc won't have any effect because the smart contract can measure it too.

Using the Chainlink Randomness Oracle Wrong

Chainlink is a popular solution to get secure random numbers. It does it in two steps. First, the smart contracts sends a randomness request to the oracle, then some blocks later, the oracle responds with a random number.

Since an attacker can't predict the future, they can't predict the random number.
Unless the smart contract uses the oracle wrong.

- The smart contract requesting randomness must not do anything until the random number is returned. Otherwise, an attacker can monitor the mempool for the oracle returning the randomness and frontrun the oracle, knowing what the random number will be.
- The randomness oracles themselves might try to manipulate your application. They cannot pick random numbers without consensus from other nodes, but they can withhold and re-order random numbers if your application requests several at the same time.
- Finality is not instant on Ethereum or most other EVM chains. Just because some block is the most recent one, it doesn't mean it won't necessarily stay that way. This is called a "chain re-org". In fact, the chain can alter more than just the final block. This is called the "re-org depth." Etherscan reports re-orgs for various chains, for example Ethereum reorgs and Polygon reorgs. Reorgs can be as deep as 30 or more blocks on Polygon, so waiting fewer blocks can make the application vulnerable (this may change when the zk-evm becomes the standard consensus on Polygon, because the finality will match Ethereum's but this is a future prediction, not a fact about the present).
- Here are the other chainlink randomness security considerations.

Getting stale data from a price Oracle

There is no SLA (service level agreement) for Chainlink to keep it's price oracles up to date within a certain time frame. When the chain is severely congested (such as when the Yuga Labs Otherside mint overwhealmed Ethereum to the point of no transactions going through), the price updates might be delayed.

A smart contract that uses a price oracle must explicitly check the data is not stale, I.e. has been updated recently within some threshold. Otherwise, it cannot make a reliable decision with respect to prices.

There is an added complication that if the price doesn't change past a deviation threshold, the oracle might not update the price to save gas, so this could affect what time threshold is considered "stale."

It is important to understand the SLA of an oracle a smart contract relies on.

Relying on only one oracle

No matter how secure an oracle seems, an attack may be discovered in the future. The only defense against this is to use multiple independent oracles.

Oracles in general are hard to get right

The blockchain can be quite secure, but putting data onto the chain in the first place necessitates some kind of off-chain operation which forgoes all the security guarantees blockchains provides. Even if oracles remain honest, their source of data can be manipulated. For example, an oracle can reliably report prices from a centralized exchange, but those can be manipulated with large buy and sell orders. Similarly, oracles that depend on sensor data or some web2 API are subject to traditional hacking vectors.

A good smart contract architecture avoids the use of oracles altogether where possible.

Mixed accounting

Consider the following contract

```
contract MixedAccounting {
    uint256 myBalance;

    function deposit() public payable {
        myBalance = myBalance + msg.value;
    }

    function myBalanceIntrospect() public view returns (uint256) {
        return address(this).balance;
    }

    function myBalanceVariable() public view returns (uint256) {
        return myBalance;
    }

    function notAlwaysTrue() public view returns (bool) {
        return myBalanceIntrospect() == myBalanceVariable();
    }
}
```

The contract above does not have a receive or fallback function, so directly transferring Ether to it will revert. However, a contract can forcefully send Ether to it with selfdestruct. In that case, myBalanceIntrospect() will be greater than myBalanceVariable(). Ether accounting method is fine, but if you use both, then the contract may have inconsistent behavior.

The same applies for ERC20 tokens.

```
contract MixedAccountingERC20 {

    IERC20 token;
    uint256 myTokenBalance;
```

```
    function deposit(uint256 amount) public {
        token.transferFrom(msg.sender, address(this), amount);
        myTokenBalance = myTokenBalance + amount;
    }

    function myBalanceIntrospect() public view returns (uint256) {
        return token.balanceOf(address(this));
    }

    function myBalanceVariable() public view returns (uint256) {
        return myTokenBalance;
    }

    function notAlwaysTrue() public view returns (bool) {
        return myBalanceIntrospect() == myBalanceVariable();
    }
}
```

Again we cannot assume that myBalanceIntrospect() and myBalanceVariable() will always return the same value. It is possible to directly transfer ERC20 tokens to MixedAccountingERC20, bypassing the deposit function and not updating the myTokenBalance variable.

When checking the balances with introspection, strict using equality checks should be avoided as the balance can be changed by an outsider at will.

Treating cryptographic proofs like passwords

This isn't a quirk of Solidity, more of a common misunderstanding among developers about how to use cryptography to give addresses special privileges. The following code is insecure

```
contract InsecureMerkleRoot {
    bytes32 merkleRoot;
    function airdrop(bytes[] calldata proof, bytes32 leaf) external {

        require(MerkleProof.verifyCalldata(proof, merkleRoot, leaf),
"not verified");
        require(!alreadyClaimed[leaf], "already claimed airdrop");
        alreadyClaimed[leaf] = true;

        mint(msg.sender, AIRDROP_AMOUNT);
    }
```

```
}
```

This code is insecure for three reasons:

1. **Anyone who knows the addresses that are selected for the airdrop can recreate the merkle tree and create a valid proof.**
2. **The leaf isn't hashed. An attacker can submit a leaf that equals the merkle root and bypass the require statement.**
3. **Even if the above two issues are fixed, once someone submits a valid proof, they can be frontrun.**

Cryptographic proofs (merkle trees, signatures, etc) need to be tied to msg.sender, which an attacker cannot manipulate without acquiring the private key.

Solidity does not upcast to the final uint size

```
function limitedMultiply(uint8 a, uint8 b) public pure returns (uint256
product) {
    product = a * b;
}
```

Although product is a uint256 variable, the multiplication result cannot be larger than 255 or the code will revert.

This issue can be mitigated by individually upcasting each variable.

```
function unlimitedMultiply(uint8 a, uint8 b) public pure returns
(uint256 product) {
    product = uint256(a) * uint256(b);
}
```

A situation like this can occur if multiplying integers packed in a struct. You should be mindful of this when multiplying small values that were packed in a struct

```
struct Packed {
     uint8 time;
     uint16 rewardRate
}

//...
```

```
Packed p;
p.time * p.rewardRate; // this might revert!
```

Solidity sneakily makes some literals uint8

The following code will revert because the ternary operator here returns a uint8.

```
function result(bool inp) external pure returns (uint256) {
    return uint256(255) + (inp ? 1 : 0);
}
```

To make it not revert, do the following:

```
function result(bool inp) external pure returns (uint256) {
    return uint256(255) + (inp ? uint256(1) : uint256(0));
}
```

You can learn more about this phenomenon in this tweet thread.

Solidity downcasting does not revert on overflow

Solidity does not check if it is safe to cast an integer to a smaller one. Unless some business logic ensures that the downcasting is safe, a library like SafeCast should be used.

```
function test(int256 value) public pure returns (int8) {
     return int8(value + 1); // overflows and does not revert
}
```

Writes to storage pointers don’t save new data.

The code looks like it copies the data in myArray[1] to myArray[0], but it doesn’t. If you comment out the final line in the function, the compiler will say the function should be turned to a view function. The write to foo doesn’t write to the underlying storage.

```
contract DoesNotWrite {
    struct Foo {
        uint256 bar;
    }
    Foo[] public myArray;
```

```
    function moveToSlot0() external {
        Foo storage foo = myArray[0];
        foo = myArray[1]; // myArray[0] is unchanged
        // we do this to make the function a state
        // changing operation
        // and silence the compiler warning
        myArray[1] = Foo({bar: 100});
    }
}
```

So don't write to storage pointers.

Deleting structs that contain dynamic datatypes does not delete the dynamic data

If a mapping (or dynamic array) is inside a struct, and the struct is deleted, the mapping or array will not be deleted.

With the exception of deleting an array, the delete keyword can only delete one storage slot. If the storage slot contains references to other storage slots, those won't be deleted.

```
contract NestedDelete {

    mapping(uint256 => Foo) buzz;

    struct Foo {
        mapping(uint256 => uint256) bar;
    }

    Foo foo;

    function addToFoo(uint256 i) external {
        buzz[i].bar[5] = 6;
    }

    function getFromFoo(uint256 i) external view returns (uint256) {
        return buzz[i].bar[5];
    }

    function deleteFoo(uint256 i) external {
        // internal map still holds the data in the
        // mapping and array
```

```
        delete buzz[i];
    }
}
```

Now let's do the following transaction sequence

1. **addToFoo(1)**
2. **getFromFoo(1) returns 6**
3. **deleteFoo(1)**
4. **getFromFoo(1) still returns 6!**

Remember, maps are never "empty" in Solidity. So if someone accesses an item which has been deleted, the transaction will not revert but instead return the zero value for that datatype.

ERC20 token issues

If you only deal with trusted ERC20 tokens, most of these issues do not apply. However, when interacting with an arbitrary or partially untrusted ERC20 token, here are some things to watch out for.

ERC20: Fee on transfer

When dealing with untrusted tokens, you shouldn't assume that your balance necessarily increases by the amount. It is possible for an ERC20 token to implement it's transfer function as follows:

```
contract ERC20 {

    // internally called by transfer() and transferFrom()
    // balance and approval checks happen in the caller
    function _transfer(address from, address to, uint256 amount)
internal returns (bool) {
        fee = amount * 100 / 99;

        balanceOf[from] -= to;
        balanceOf[to] += (amount - fee);

        balanceOf[TREASURY] += fee;

        emit Transfer(msg.sender, to, (amount - fee));
        return true;
    }
}
```

This token applies a 1% tax to every transaction. So if a smart contract interacts with the token as follows, we will either get unexpected reverts or stolen money.

```
contract Stake {

    mapping(address => uint256) public balancesInContract;

    function stake(uint256 amount) public {
        token.transferFrom(msg.sender, address(this), amount);

        balancesInContract[msg.sender] += amount; // THIS IS WRONG!
    }

    function unstake() public {
        uint256 toSend = balancesInContract[msg.sender];
        delete balancesInContract[msg.sender];

        // this could revert because toSend is 1% greater than// the
amount in the contract. Otherwise, 1% will be "stolen"// from other
depositors.
        token.transfer(msg.sender, toSend);
    }
}
```

ERC20: rebasing tokens

The rebasing token was popularized by Olympus DAO's sOhm token and Ampleforth's AMPL token. Coingecko maintains a list of rebasing ERC20 tokens.

When a token rebases, the total supply changes and everyone's balance increases or decreases depending on the rebase direction.

The following code is likely to break when dealing with a rebasing token

```
contract WillBreak {
    mapping(address => uint256) public balanceHeld;
    IERC20 private rebasingToken

    function deposit(uint256 amount) external {
        balanceHeld[msg.sender] = amount;
        rebasingToken.transferFrom(msg.sender, address(this), amount);
    }
```

```
    function withdraw() external {
        amount = balanceHeld[msg.sender];
        delete balanceHeld[msg.sender];

        // ERROR, amount might exceed the amount
        // actually held by the contract
        rebasingToken.transfer(msg.sender, amount);
    }
}
```

The solution of many contracts is to simply disallow rebasing tokens. However, one could modify the code above to check balanceOf(address(this)) before transfering the account balance to the sender. Then it would still work even if the balance changes.

ERC20: ERC777 in ERC20 clothing

ERC20, if implemented according to the standard, ERC20 tokens do not have transfer hooks, and thus transfer and transferFrom do not have a reentrancy issue.

There are meaningful advantages to tokens with transfer hooks, which is why all NFT standards implement them, and why ERC777 was finalized. However, it's caused enough confusion that Openzeppelin deprecated the ERC777 library.

If you want your protocol to be compatible with tokens that behave like ERC20 tokens but have transfer hooks, then it's a simple matter of treating the functions transfer and transferFrom like they will issue a function call to the receiver.

This ERC777 re-entrancy happened to Uniswap (Openzeppelin documented the exploit here if you are curious).

ERC20: Not all ERC20 tokens return true

The ERC20 specification dictates that an ERC20 token must return true when a transfer succeeds. Because most ERC20 implementations cannot fail unless the allowance is insufficient or the amount transferred is too much, most devs have become accustomed to ignoring the return value of ERC20 tokens and assuming a failed trasfer will revert.

Frankly, this is not consequential if you are only working with a trusted ERC20 token you know the behavior of. But when dealing with arbitrary ERC20 tokens, this variance in behavior must be accounted for.

There is an implicit expectation in many contracts that failed transfers should always revert, not return false because most ERC20 tokens don't have a mechanism to return false, so this has lead to a lot of confusion.

Further complicating this matter is that some ERC20 tokens don't follow the protocol of returning true, notably Tether. Some tokens revert on a failure to transfer, which will cause the revert to bubble up to the caller. Thus, some libraries wrap ERC20 token transfer calls to intercept the revert and return a boolean instead. Here are some implementations

Openzeppelin SafeTransfer
Solady SafeTransfer (considerably more gas efficient)

ERC20: Address Poisoning

This is not a smart contract vulnerability, but we mention it here for completeness.
Transferring zero ERC20 tokens is permitted by the specification. This can lead to confusion for frontend applications, and possible trick users about who they recently sent tokens to. Metamask has more on that in this thread.

ERC20: Just flat out rugged

(In web3 parlance "rugged" means "having the rug pulled out from under you.")
There's nothing stopping someone from adding a function to an ERC20 token that lets them create, transfer, and burn tokens at will — or selfdestructing or upgrading. So fundamentally, there is a limit to how "untrusted" an ERC20 token can be.

Logic bugs in lending protocols

When considering how lending and borrowing based DeFi protocols can break, it's helpful to think about bugs propagate at the software level and affect the business logic level. There are a lot of steps to forming and closing a bond contract. Here are some attack vectors to consider.

Ways lenders lose out

- Bugs that enable the principal due to reduce (possible to zero) without making any payments.
- The buyer's collateral cannot be liquidated when the loan is not paid back or the collateral drops below the threshold.
- If the protocol has a mechanism for transferring debt ownership, this could be a vector for stealing bonds from lenders.
- The due date of the loan principal or payments is improperly moved to a later date.

Ways borrowers lose out

- A bug where paying back the principal does not lead to principal reduction.
- A bug or griefing attack prevents the user from making payment.
- The principal or interest rate is illegitimately increased.
- Oracle manipulation leads to devaluing the collateral.

- The due date of the loan principal or payments is improperly moved to an earlier date.

If collateral is drained from the protocol, then both the lender and borrower lose out, since the borrower has no incentive to pay back the loan, and the borrower loses the principal.

As can be seen above, there are a lot more levels to a DeFi protocol getting "hacked" than a bunch of money being drained from the protocol (the kind of events that usually make the news).

Vulnerabilities in staking protocols

The kind of hacks that make the news are staking protocols getting hacked for millions of dollars, but that is not the only issue to look for.

- Can rewards be delayed in payout, or claimed too early?
- Can rewards be improperly reduced or increased? In the worse case, can the user be prevented from receiving any reward?
- Can people claim principal or rewards that don't belong to them, in the worst case draining the protocol?
- Can deposited assets get stuck in the protocol (partially or fully) or be improperly delayed in withdrawal?
- Conversely, if staking requires a time commitment, can users withdraw before the commitment time?
- If the payout is in a different asset or currency, can the value of it be manipulated within the scope of the smart contract in question? This is relevant if the protocol mints its own tokens to reward liquidity providers or stakers.
- If there is an expected and disclosed element of risk of losing principal staking, can that risk be improperly manipulated?
- Do key parameters of the protocol have admin, centralization, or governance risk?

The key areas to look are the areas of the code that touch the "money exit" portions of the code.

There is a "money entrance" vulnerability to look for too.

- Can users who have a right to participate in staking assets in the protocol be improperly prevented from doing so?

Rewards users receive have an implicit risk-reward profile and an expected time-value of money aspect. It's helpful to be explicit about what those assumptions are and how the protocol could be caused to deviate from expectations.

Unchecked return values

There are two ways to call an external smart contract: 1) calling the function with an interface definition; 2) using the .call method. This is illustrated below

```
contract A {
    uint256 public x;

    function setx(uint256 _x) external {
        require(_x > 10, "x must be bigger than 10");
        x = _x;
    }
}

interface IA {
    function setx(uint256 _x) external;
}

contract B {
    function setXV1(IA a, uint256 _x) external {
        a.setx(_x);
    }

    function setXV2(address a, uint256 _x) external {
        (bool success, ) =
            a.call(abi.encodeWithSignature("setx(uint256)", _x));
        // success is not checked!
    }
}
```

In contract B, setXV2 can silently fail if _x is less than 10. When a function is called via the .call method, the callee can revert, but the parent will not revert. The value of success must be checked and the code behavior must branch accordingly.

msg.value in a loop

Using msg.value inside a loop is dangerous because this might allow the sender to “re-use” the msg.value.

This can show up with payable multicalls. Multicalls enable a user to submit a list of transactions to avoid paying the 21,000 gas transaction fee over and over. However, msg.value gets “re-used” while looping through the functions to execute, potentially enabling the user to double spend.

This was the root cause of the Opyn Hack.

Private Variables

Private variables are still visible on the blockchain, so sensitive information should never be stored there. If they weren't accessible, how would the validators be able to process transactions that depend on their values? Private variables cannot be read from an outside Solidity contract, but they can be read off-chain using an Ethereum client.

To read a variable, you need to know its storage slot. In the following example, the storage slot of myPrivateVar is 0.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
contract PrivateVarExample {
    uint256 private myPrivateVar;

    constructor(uint256 _initialValue) {
        myPrivateVar = _initialValue;
    }
}
```

Here is the javascript code to read the private variable of the deployed smart contract

```
const Web3 = require("web3");
const PRIVATE_VAR_EXAMPLE_ADDRESS = "0x123..."; // Replace with your
contract address

async function readPrivateVar() {
  const web3 = new Web3("http://localhost:8545"); // Replace with your
provider's URL

  // Read storage slot 0 (where 'myPrivateVar' is stored)
  const storageSlot = 0;
  const privateVarValue = await web3.eth.getStorageAt(
    PRIVATE_VAR_EXAMPLE_ADDRESS,
    storageSlot
  );

  console.log("Value of private variable 'myPrivateVar':",
  web3.utils.hexToNumberString(privateVarValue));
}

readPrivateVar();
```

Try Catch is hard to get right

Whereas a low-level call will simply return true or false based on success, a try catch makes a distinction between a panic and a revert, and may silently fail if the return data cannot be parsed properly. It's best to avoid it and simply use low level calls.

Insecure Delegate Call

Delegatecall should never be used with untrusted contracts as it hands over all control to the delegatecallee. In this example, the untrusted contract steals all the ether in the contract.

```
contract UntrustedDelegateCall {
    constructor() payable {
        require(msg.value == 1 ether);
    }

    function doDelegateCall(address _delegate, bytes calldata data)
public {
        (bool ok, ) = _delegate.delegatecall(data);
        require(ok, "delegatecall failed");
    }

    }

    contract StealEther {
        function steal() public {
            // you could also selfdestruct here
            // if you really wanted to be mean
            (bool ok,) =
                tx.origin.call{value: address(this).balance}("");
            require(ok);
        }

        function attack(address victim) public {
            UntrustedDelegateCall(victim).doDelegateCall(
                address(this),
                abi.encodeWithSignature("steal()"));
        }
}
```

Upgrade bugs related to proxies

We can't do justice to this topic in a single section. Most upgrade bugs can be generally avoided by using the hardhat plugin from Openzeppelin and reading about what issues it protects against.

(https://docs.openzeppelin.com/upgrades-plugins/1.x/).

As a quick summary, here are issues related to smart contract upgrades:

- selfdestruct and delegatecall should not be used inside implementation contracts
- care must be taken that storage variables never overwrite each other during upgrades
- calling external libraries should be avoided in implementation contracts because it isn't possible to predict how they will affect storage access
- deployer must never neglect to call the initialization function
- not including a gap variable in base contracts to prevent storage collision when new variables are added to the base contract (this is handled by the hardhat plugin automatically)
- the values in immutable variables are not preserved between upgrades
- doing anything in the constructor is highly discouraged because future upgrades would have to carry out identical constructor logic to maintain compatibility.

Overpowered Admins

Just because a contract has an owner or an admin, it doesn't mean that their power needs to be unlimited. Consider an NFT. It's reasonably for only the owner to withdraw the earnings from the NFT sale, but being able to pause the contract (block transfers) could wreak havoc if the owner's private keys get compromised. Generally, administrator priviledges should be as minimal as possible to minimize unnecessary risk.

Speaking of contract ownership…

Use Ownable2Step instead of Ownable

This is technically not a vulnerability, but OpenZeppelin ownable can lead to loss of contract ownership if ownership is transferred to a non-existent address. Ownable2step requires the receiver to confirm ownership. This insures against accidentally sending ownership to a mistyped address.

Rounding Errors

Solidity does not have floats, so rounding errors are inevitable. The designer must be conscious of whether the right thing to do is to round up or to round down, and in whose favor the rounding should be.

Division should always be performed last. The following code incorrectly converts between stablecoins that have a different number of decimals. The following exchange mechanism allows a user to take a small amount of USDC (which has 6 decimals) for free when exchanging for dai (which has 18 decimals). The variable daiToTake will round down to zero, taking nothing from the user in exchange for a non-zero usdcAmount.

```
contract Exchange {

    uint256 private constant CONVERSION = 1e12;

    function swapDAIForUSDC(uint256 usdcAmount) external pure returns
(uint256 a) {
        uint256 daiToTake = usdcAmount / CONVERSION;
        conductSwap(daiToTake, usdcAmount);
    }
}
```

Frontrunning

Frontrunning in the context of Etheruem (and similar chains) means observing a pending transaction and executing another transaction before it by paying a higher gas price. That is, the attacker has "run in front" of the transaction. If the transaction is a profitable trade, then it makes sense to copy the transaction exactly except pay a higher gas price. This phenomenon is sometimes referred to as MEV, which means miner extractable value, but sometimes maximal extractable value in other contexts. Block producers have unlimited power to reorder transactions and insert their own, and historically, block producers were miners before Ethereum went to proof of stake, hence the name.

Frontrunning: Unprotected withdraw

Withdrawing Ether from a smart contract can be considered a "profitable trade." You execute a zero-cost transaction (aside from the gas) and end up with more cryptocurrency than you started with.

```
contract UnprotectedWithdraw {

    constructor() payable {
        require(msg.value == 1 ether, "must create with 1 eth");
    }

    function unsafeWithdraw() external {
        (bool ok, ) = msg.sender.call{value: address(this).value}("");
        require(ok, "transfer failed").
    }
}
```

If you deploy this contract and try to withdraw, a frontrunner bot will notice your call to "unsafeWithdraw" in the mempool and copy it to get the Ether first.

Frontrunning: ERC4626 Inflation attack, a combination of frontrunning and rounding errors

We've written in depth about the ERC-4626 inflation attack in our ERC4626 tutorial. But the gist of it is that an ERC4626 contract distributes "share" tokens based on the percentage of "assets" that a trader contributes. Roughly, it works as follows:

```
function getShares(...) external {
    // code
    shares_received = assets_contributed / total_assets;
    // more code
}
```

Of course, nobody will contribute assets and get no shares back, but they can't predict that will happen if someone can frontruns the trade to get the shares.
For example, they contributes 200 assets when the pool has 20, they expect to get 100 shares. But if someone frontruns the transaction to deposit 200 assets, then the formula will be 200 / 220, which rounds down to zero, causing the victim to lose assets and get zero shares back.

Frontrunning: ERC20 approval

It's best to illustrate this with a real example rather than describe it in the abstract

1. Suppose Alice approves Eve for 100 tokens. (Eve is always the evil person, not Bob, so we will keep convention).
2. Alice changes her mind and sends a transaction to change Eve's approval to 50.
3. Before the transaction to change the approval to 50 is included in the block, it sits in the mempool where Eve can see it.
4. Eve sends a transaction to claim her 100 tokens to frontrun the approval for 50.
5. The approval for 50 goes through
6. Eve collects the 50 tokens.

Now Eve has 150 tokens instead of 100 or 50. The solution to this is to set the approval to zero before increasing or decreasing it, when dealing with untrusted approvals.

Frontrunning: Sandwich attacks

The price of an asset moves in response to buying and selling pressure. If a large order is sitting in the mempool, traders have an incentive to copy the order but with a higher gas price. That way, they purcase the asset, let the large order move the price up, then they sell right away. The sell order is sometimes called "backrunning." The sell order can be done with by placing a sell order with a lower gas price so that the sequence looks like this

1. frontrun buy
2. large buy

3. sell

The primary defense against this attack is to provide a “slippage” parameter. If the “frontrun buy” itself pushes the price up past a certain threshld, the “large buy” order will revert making the frontrunner fail on the trade. See this resource for additional bugs and vulnerabilities related to slippage.

It’s called a sandwhich, because the large buy is sandwhiched by the frontrun buy and the backrun sell. This attack also works with large sell orders, just in the opposite direction.

Learn more about frontrunning

Frontrunning is a massive topic. Flashbots has researched the topic extensively and published several tools and research articles to help minimize it’s negative externalities. Whether frontrunning can be “designed away” with proper blockchain architecture is a subject for debate which has not been conclusively settled. The following two articles are enduring classics on the subject:

Ethereum is a dark forest
Escaping the dark forest

Signature Related

Digital signatures have two uses in the context of smart contracts:

- enabling addresses to authorize some transaction on the blockchain without making an actual transaction
- proving to a smart contract that the sender has some authority to do something, according to a predefined address

Here is an example of using digital signatures safely to give a user the priviledge to mint an NFT:

```
import "@openzeppelin/contracts/utils/cryptography/ECDSA.sol";
import "@openzeppelin/contracts/token/ERC721/ERC721.sol";

contract NFT is ERC721("name", "symbol") {
    function mint(bytes calldata signature) external {
        address recovered =
keccak256(abi.encode(msg.sender)).toEthSignedMessageHash().recover(sign
ature);
        require(recovered == authorizer, "signature does not match");
    }
}
```

A classic example is the Approve functionality in ERC20. To approve an address to withdraw a certain amount of tokens from our account, we have to make an actual Ethereum transaction, which costs

gas.

It's sometimes more efficient to pass a digital signature to the recipient off-chain, then the recipient supplies the signature to the smart contract to prove they were authorized to conduct the transaction.

ERC20Permit enables approvals with a digital signature. The function is described as follows

```
function permit(address owner,
    address spender,
    uint256 amount,
    uint256 deadline,
    uint8 v,
    bytes32 r,
    bytes32 s
) public
```

Rather than sending an actual approve transaction, the owner can "sign" the approval for the spender (along with a deadline). The approved spender can then call the permit function with the provided parameters.

Anatomy of a signature

You'll see the variables, v, r, and s frequently. They are represented in solidity with the datatypes uint8, bytes32, and bytes32 respectively. Sometimes, signatures are represented as a 65 byte array which is all of these values concatenated together as abi.encodePacked(r, s, v);

The other two essential components of a signature are the message hash (32 bytes) and the signing address. The sequence looks like this

1. A private key (privKey) is used to generate a public address (ethAddress)
2. A smart contract stores ethAddress in advance
3. An offchain user hashes a message and signs the hash. This produces the pair msgHash and the signature (r, s, v)
4. The smart contract receives a message, hashes it to produce msgHash, then combines it with (r, s, v) to see what address comes out.
5. If the address matches ethAddress, the signature is valid (under certain assumptions which we will see soon!)

Smart contracts use the precompiled contract ecrecover in step 4 to do what we called the combination and get the address back.

There are a lot of steps in this process where things can go sideways.

Signatures: ecrecover returns address(0) and doesn't revert when the address is invalid

This can lead to a vulnerability if an uninitialized variable is compared to the output of ecrecover.

This code is vulnerable

```
contract InsecureContract {

    address signer;
    // defaults to address(0)
    // who lets us give the beneficiary the airdrop without them// spending gas
    function airdrop(address who, uint256 amount, uint8 v, bytes32 r, bytes32 s) external {

        // ecrecover returns address(0) if the signature is invalid
        require(signer == ecrecover(keccak256(abi.encode(who, amount)), v, r, s), "invalid signature");

        mint(msg.sender, AIRDROP_AMOUNT);
    }
}
```

Signature replay

The signature replay happens when a contract doesn't track if a signature has been used previously. In the following code, we fix the previous issue, but it's still not secure.

```
contract InsecureContract {

    address signer;

    function airdrop(address who, uint256 amount, uint8 v, bytes32 r, bytes32 s) external {

        address recovered == ecrecover(keccak256(abi.encode(who, amount)), v, r, s);
        require(recovered != address(0), "invalid signature");
        require(recovered == signer, "recovered signature not equal signer");


        mint(msg.sender, amount);
    }
```

```
}
```

People can claim the airdrop as many times as they want!

We could add the following lines

```
bytes memory signature = abi.encodePacked(v, r, s);
require(!used[signature], "signature already used");
// mapping(bytes => bool);
used[signature] = true;
```

Alas, the code is still not secure!

Signature malleability

Given a valid signature, an attacker can do some quick arithmetic to derive a different one. The attacker can then "replay" this modified signature. But first, let's provide some code that demonstrates we can start with a valid signature, modify it, and show the new signature still passes.

```
contract Malleable {

    // v = 28
    // r =
0xf8479d94c011613baeffe9239e4ff65e2adbac744c34217ca7d51378e72c5204
    // s =
0x57af17590a914b759c45aaeabaf513d5ef72d7da1bdd19d9f2e1bc371ece5b86
    // m =
0x0000000000000000000000000000000000000000000000000000000000000003
    function foo(bytes calldata msg, uint8 v, bytes32 r, bytes32 s)
public pure returns (address, address){
        bytes32 h = keccak256(msg);
        address a = ecrecover(h, v, r, s);


        // The following is math magic to invert the
        // signature and create a valid one
        // flip s
        bytes32 s2 =
bytes32(uint256(0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25
E8CD0364141) - uint256(s));

        // invert v
        uint8 v2;
```

```
        require(v == 27 || v == 28, "invalid v");
        v2 = v == 27 ? 28 : 27;

        address b = ecrecover(h, v2, r, s2);

        assert(a == b);
        // different signatures, same address!;
        return (a, b);
    }
}
```

As such, our running example is still vulnerable. Once someone presents a valid signature, it's mirror image signature can be produced and bypass the used signature check.

```
contract InsecureContract {

    address signer;

    function airdrop(address who, uint256 amount, uint8 v, bytes32 r,
bytes32 s) external {

        address recovered == ecrecover(keccak256(abi.encode(who,
amount)), v, r, s);
        require(recovered != address(0), "invalid signature");
        require(recovered == signer, "recovered signature not equal
signer");

        bytes memory signature = abi.encodePacked(v, r, s);
        require(!used[signature], "signature already used"); // this
can be bypassed
        used[signature] = true;

        mint(msg.sender, amount);
    }
}
```

Secure signatures

You're probably wanting some secure signature code at this point, right? We refer you to our tutorial on creating signatures in solidity and testing them in foundry. But here is the checklist.

- **Use openzeppelin's library to prevent malleability attacks and recover to zero issues**

- Don't use signatures as a password. The messages needs to contain information that attackers cannot easily re-use (e.g. msg.sender)
- Hash what you are signing on-chain
- Use a nonce to prevent replay attacks. Better yet, follow EIP712 so that usese can see what they are signing and you can prevent signatures from being re-used between contracts and different chains.

Signatures can be forged or crafted without proper safeguards

The attack above can be generalized further if hashing is not done on chain. In the examples above, the hashing was done in the smart contract, so the above examples are not vulnerable to the following exploit.

Let's look at the code for recovering signatures

```
// this code is vulnerable!
function recoverSigner(bytes32 hash, uint8 v, bytes32 r, bytes32 s)
public returns (address signer) {
    require(signer == ecrecover(hash, v, r, s), "signer does not
match");
    // more actions
}
```

The user supplies both the hash and the signatures. If the attacker has already seen a valid signature from the signer, they can simply reuse the hash and signature of another message.

This is why it is very important to hash the message **in the smart contract**, not off-chain.

To see this exploit in action, see the CTF we posted on Twitter.
Original Challenge:
Part 1: https://twitter.com/RareSkills_io/status/1650869999266037760
Part 2: https://twitter.com/RareSkills_io/status/1650897671543197701

Solutions:
https://twitter.com/RareSkills_io/status/1651527648676573185
https://twitter.com/RareSkills_io/status/1651224817465540611

Signatures as identifiers

Signatures should not be used to identify users. Because of malleability, they cannot be assumed to be unique. Msg.sender has much stronger uniqueness guarantees.

Some Solidity compiler versions have bugs

See a security exercise we hosted on Twitter here. When auditing a codebase, check the Solidity version against the release announcements on the Solidity page to see if a bug might be present.

Assuming smart contracts are immutable

Smart contracts can be upgraded with the Proxy Pattern (or more rarely, the metamorphic pattern). Smart contracts should not rely on the functionality of an arbitrary smart contract to remain unchanged.

Transfer() and send() can break with multi-signature wallets

The solidity functions transfer and send should not be used. They intentionally limit the amount of gas forwarded with the transaction to 2,300, which will cause most operations to run out of gas.

The commonly used gnosis safe multi-signature wallet supports forwarding the call to another address in the fallback function. If someone uses transfer or send to send Ether to the multisig wallet, the fallback function could run out of gas and the transfer would fail. A screenshot of the gnosis safe fallback function is provided below. The reader can clearly see there is more than enough operations to use up the 2300 gas.

Gnosis Safe Fallback Function

If you need to interact with a contract that uses transfer and send, see our article on Ethereum access list transactions that allows you to reduce the gas cost of storage and contract access operations.

Is Arithmetic overflow still relevant?

Solidity 0.8.0 has built in overflow and underflow protection. So unless an unchecked block is present, or low level code in Yul is used, there is no danger of overflow. As such, SafeMath libraries should not be used as they waste gas on the extra checks.

What about block.timestamp?

Some literature documents that block.timestamp is a vulnerability vector because miners can manipulate it. This usually applies to using timestamps as a source of randomness, which should not be done anyway as documented earlier. Post-merge Ethereum updates the timestamp in exactly 12 second (or multiples of 12 second) intervals. However, measuring time in second-level granularity is an anti-pattern. On the scale of one minute, there is considerable opportunity for error if a validator misses their block slot and a 24 second gap in block production happens.

Corner Cases, Edge Cases, and Off By One Errors

Corner cases cannot be easily defined, but once you have seen enough of them, you start to develop an intuition for them. A corner case can be something like someone trying to claim a reward, but having nothing staked. This is valid, we should just give them zero reward. Similarly, we generally want to divide up rewards evenly, but what if there is only one recipient, and technically no division should happen?

Corner Case: Example 1

This example was taken from Akshay Srivastav's twitter thread and modified.
Consider the case where someone can conduct a privileged action if a set of privileged addresses provide a signature for it.

```
contract VulnerableMultisigAuthorization {
    struct Authorization {
        bytes signature;
        address authorizer;
        bytes32 hashOfAction;
        // more fields
    }

    // more codef
    unction takeAction(Authorization[] calldata auths, bytes calldata
action) public {
        // logic for avoiding replay attacks
        for (uint256 i; i < auths.length; ++i) {

            require(validateSignature(auths[i].signature,
auths[i].authorizer), "invalid signature");
            require(authorizers[auths[i].authorizer], "address is not
an authorizer");

        }

        doTheAction(action)
    }
}
```

If any of the signatures are not valid, or the signatures don't match to a valid address, the revert will happen. But what if the array is empty? In that case, it will jump all the way down to doTheAction without the need for any signatures.

Off-By-One: Example 2

```
contract ProportionalRewards {

    mapping(address => uint256) originalId;
    address[] stakers;

    function stake(uint256 id) public {
        nft.transferFrom(msg.sender, address(this), id);
        stakers.append(msg.sender);
    }

    function unstake(uint256 id) public {
        require(originalId[id] == msg.sender, "not the owner");

        removeFromArray(msg.sender, stakers);

        sendRewards(msg.sender,
            totalRewardsSinceLastclaim() / stakers.length());

        nft.transferFrom(address(this), msg.sender, id);
    }
}
```

Although the code above doesn't show all the function implementations, even if the functions behave as their names describe, there is still a bug. Can you spot it? Here is a picture to give you some space to not see the answer before you scroll down.

Stop Scrolling

The removeFromArray and sendRewards function are in the wrong order. If there is only one user in the stakers array, there will be a divide by zero error, and the user won't be able to withdraw their NFT. Furthermore, the rewards are probably not divided the way the author intends. If there were original four stakers, and one person withdraws, he will get a third of the rewards since the array length is 3 at the time of withdrawal.

Corner Case Example 3: Compound Finance Reward Miscalculation

Let's use a real example that by some estimates caused over $100 million dollars of damage. Don't worry if you don't fully understand the Compound protocol, we will only focus on the relevant parts. (Also the Compound protocol is one of the most important and consequential protocols in the history

of DeFi, we teach it in our DeFi bootcamp, so if this is your first impression of the protocol, don't be misguided).
Anyway, the point of Compound is to reward users for lending their idle cryptocurrency to other traders who might have a use for it. The lenders are paid both in interest and in COMP tokens (the borrowers could claim a COMP token reward to, but we won't focus on that right now).

The Compound Comptroller is a proxy contract that delegates calls to implementations that can be set by the Compound Governance.

At governance proposal 62 on September 30, 2021, the implementation contract was set to an implementation contract that had the vulnerability. The same day it went live, it was observed on Twitter that some transactions were claiming COMP rewards despite staking zero tokens.

The vulnerable function distributeSupplierComp()

Here is the original code

The bug, ironically, is in the TODO comment. “Don’t distribute supplier COMP if the user is not in the supplier market.” But there is no check in the code for that. As long as the user holds staking token in their wallet (CToken(cToken).balanceOf(supplier);), then

Proposal 64 fixed the bug on October 9, 2021.

Although this could be argued to be an input validation bug, the users didn’t submit anything malicious. If someone tries to claim a reward for not staking anything, the correct computation should be zero. Arguably, it’s more of a business logic or corner case error.

Real World Hacks

DeFi hacks that happen in the real world often times don’t fall into the nice categories above.

Pairity Wallet Freeze (November 2017)

The parity wallet was not intended to be used directly. It was a reference implementation that smart contract clones would point to. To implementation allowed for the clones to selfdestruct if desired, but this required all the wallet owners to sign off on it.

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

function initWallet(address[] _owners, uint _required, uint _daylimit)
only_uninitialized {
  initDaylimit(_daylimit);
  initMultiowned(_owners, _required);
}
```

The wallet owners are declared

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

Some literature describes this as an “unprotected selfdestruct” i.e. an access control failure, but this isn’t quite accurate. The problem was that the initWallet function was not called on the implementation contract and that allowed someone to call the initWallet function themselves and make themselves the owner. That gave them the authority to call the kill function. The root cause was that the implementation was not initialized. Therefore, the bug was introduced not due to faulty solidity code, but due to a faulty deployment process.

Badger DAO Hack (December 2021)

No Solidity code was exploited in this hack. Instead, the attackers obtain the Cloudflare API key and injected a script into the website frontend that altered user transactions to direct withdrawals to the attacker address. Read more in this article.

Attack vectors for wallets

Private keys with insufficient randomness

The motivation for discovering addresses with a lot of leading zeros is that they are more gas efficient to use. An Ethereum transaction is charged 4 gas for a zero byte in the transaction data and 16 gas for a non-zero byte. As such,
Wintermute was hacked because it used the profanity address (writeup). Here is 1inch's writeup of how the profanity address generator was compromised.

The trust wallet had a similar vulnerability documented in this article (https://blog.ledger.com/Funds-of-every-wallet-created-with-the-Trust-Wallet-browser-extension-could-have-been-stolen/)

Note that this does not apply to smart contracts with leading zeros discovered by changing the salt in create2, as smart contracts do not have private keys.

Reused nonces or insufficiently random nonces.

The "r" and "s" point on the Elliptic Curve signature is generated as follows

```
r = k * G (mod N)
s = k^-1 * (h + r * privateKey) (mod N)
```

G, r, s, h, an N are all publicly known. If "k" becomes public, then "privateKey" is the only unknown variable, and can be solved for. Because of this wallets need to generate k perfectly randomly and never reuse it. If the randomness isn't perfectly random, then k can be inferred. Insecure randomness generation in the Java library left a lot of Android bitcoin wallets vulnerable in 2013. (Bitcoin uses the same signature algorithm as Ethereum.) (https://arstechnica.com/information-technology/2013/08/all-android-created-bitcoin-wallets-vulnerable-to-theft/).

Most vulnerabilities are application specific

Training yourself to quickly recognize the anti-patterns in this list will make you a more effective smart contract programmer, but most smart contract bugs of consequence are due to a mismatch between the intended business logic and what the code actually does.
Other areas where bugs can occur:

- bad tokenomic incentives
- off by one errors
- typographical errors

- admins or users getting their private keys stolen

Many vulnerabilities could have been caught with unit tests

Smart contract unit testing is arguably the most basic safeguards for smart contract, but a shocking number of smart contracts either lack them or have insufficient test coverage.

But unit tests tend only to test the “happy path” (expected/designed behavior) of contracts. To test the surprising cases, additional test methodologies must be applied.

Before a smart contract is sent for audit, the following should be done first:

- Static analysis with tools such as Slither to ensure basic mistakes were not missed
- 100% line and branch coverage through unit testing
- Mutation testing to ensure the unit tests have robust assert statements
- Fuzz testing, especially for arithmetic
- Invariant testing for stateful properties
- Formal verification where appropriate

For those unfamiliar with some of the methodologies here, Patrick Collins of Cyfrin Audits has a humorous introduction to stateful and stateless fuzzing in his video.
Tools to accomplish these tasks are rapidly becoming more widespread and easier to use.

More resources

Some authors have compiled a list of previous DeFi hacks in these Repos:

- https://github.com/coinspect/learn-evm-attacks
- https://github.com/SunWeb3Sec/DeFiHackLabs
- https://rekt.news/

Secureum has been widely used to study and practice security, but keep in mind the repo hasn’t been substantially updated for 2 years

- https://github.com/x676f64/secureum-mind_map

You can practice exploiting solidity vulnerabilities with our Solidity Riddles repository.

- https://github.com/RareSkills/solidity-riddles

DamnVulnerableDeFi is a classic wargame every developer should practice

- https://damnvulnerabledefi.xyz

Capture The Ether and Ethernaut are classics, but keep in mind some of the problems are unrealistically easy or teach outdated Solidity concepts

- https://capturetheether.com
- https://ethernaut.openzeppelin.com

Some reputable crowdsourced security firms have a useful list of past audits to study.

- https://code4rena.com/
- https://www.sherlock.xyz/

Becoming a smart contract auditor

If you aren't fluent in Solidity, then there is no way you'll be able to audit Ethereum smart contracts. See our free Solidity tutorial if you are just starting off.
There is no industry recognized certification for becoming a smart contract auditor. Anyone can create a website and social media profiles claiming to be a solidity auditor and start selling services, and many have done so. Therefore, use caution and get referrals before hiring one.

To become a smart contract auditor, you need to be substantially better than the average solidity developer at spotting bugs. As such, **the "roadmap" to becoming an auditor is nothing more than months and months of relentless and deliberate practice until you are better smart contract bug catcher than most.**

If you lack the determination to outperform your peers at identifying vulnerabilities, it's unlikely you'll spot the critical issues before the highly trained and motivated criminals do.

Cold truth about your chances of success of becoming a smart contract security auditor

Smart contract auditing recently has been perceived as a desirable field to work in due to the perception that it is lucrative. Indeed, some bug bounty payouts have exceeded 1 million dollars, but this is the exceedingly rare exception, not the norm.
Code4rena has a public leaderboard of payouts from competitors in their audit contests, which gives us some data about success rates.
There are 1171 names on the board, yet

- Only 29 competitors have over $100,000 in lifetime earnings (2.4%)
- Only 57 have over $50,000 in lifetime earnings (4.9%)
- Only 170 have over $10,000 in lifetime earnings (14.5%)

Also consider this, when Openzeppelin opened up an application for a security research fellowship (not a job, a pre-job screening and training), they received over 300 applications only to select fewer than 10 candidates, of which even fewer would get a full time job.

https://twitter.com/David_Bessin/status/1625167906328944640

That's a lower admission rate than Harvard.

Smart contract auditing is a competitive zero-sum game. There are only so many projects to audit, only so much budget for security, and only so many bugs to find. If you begin studying security now, there are dozens of highly motivated individuals and teams with a massive headstart on you. Most projects are willing to pay a premium for an auditor with a reputation rather than an untested new auditor.
In this article, we've listed at least 20 different categories of vulnerabilites. If you spent one week mastering each one (which is somewhat optimistic), you're only just starting to understand what is common knowledge to experienced auditors. We haven't covered gas optimization or tokenomics in this article, both of which are important topics for an auditor to understand. Do the math and you'll see this not a short journey.

That said, the community is generally friendly and helpful to newcomers and tips and tricks abound. But for those reading this article in hopes of making a career out of smart contract security, it is important to clearly understand that the odds of obtaining a lucrative career are not in your favor. Success is not the default outcome.
It *can* be done of course, and quite a few people have gone from knowing no Solidity to having a lucrative career in auditing. It's arguably easier to get a job as a smart contract auditor in a two year timespan than it is to get admitted into law school and pass the bar exam. It certainly has more upside compared to a lot of other career choices.

But it will nevertheless require herculean perseverance on your part to master the mountain of rapidly evolving knowledge ahead of you and hone your intuition for spotting bugs.

This is not to say that learning smart contract security is not a worthwhile pursuit. It absolutely is. But if you are approaching the field with dollar signs in your eyes, keep your expectations in check.

Conclusion

It is important to be aware of the known anti-patterns. However, most real-world bugs are application specific. Identifying either category of vulnerabilities requires continual and deliberate practice.

Learn smart contract security, and many more Ethereum development topics with our industry-leading solidity training.