**Lecture 5-1 Notes**

**Vulnerabilities on Smart Contract**

**Slide 3**

Apriorit has teams working on smart contract development and penetration testing. The main blockchain security issues associated with smart contracts relate to bugs in source code, a network’s virtual machine, the runtime environment for smart contracts, and the blockchain itself.

**Slide 4-7**

Let’s look at each of these attack vectors.

* **Vulnerabilities in contract source code**

If a smart contract has vulnerabilities in its source code, it poses a risk to parties that sign the contract. For instance, bugs discovered in an Ethereum contract cost its owners $80 million in 2016. One of the common vulnerabilities in Solidity opens up a possibility to delegate control to untrusted functions from other smart contracts, known as a reentrancy attack. During this attack, contract A calls a function from contract B that has an undefined behavior. In turn, contract B can call a function from contract A and use it for malicious purposes.

* **Vulnerabilities in virtual machines**

The Ethereum Virtual Machine (EVM) is a distributed stack-based computer where all smart contracts of Ethereum-based blockchains are executed. The most common vulnerabilities of the EVM are the following:

* + Immutable defects — Blockchain blocks are immutable by nature, which means that once a smart contract is created, it can’t be changed. But if a smart contract contains any bugs in its code, they also are impossible to fix. There’s a risk that cybercriminals can discover and exploit code vulnerabilities to steal Ether or create a new fork, as happened with the DAO attack.
  + Cryptocurrency lost in transfer — This is possible if Ether is transferred to an orphaned address that doesn’t have any owner or contract.
  + Bugs in access control — There’s a missed modifier bug in Ethereum smart contracts that allows a hacker to get access to sensitive functionality in a contract.
  + Short address attack — This is possible because the EVM can accept incorrectly padded arguments. Hackers can exploit this vulnerability by sending specifically crafted addresses to potential victims. For instance, during a successful attack on the Coindash ICO in 2017, a modification to the Coindash Ethereum address made victims send their Ether to the hacker’s address.

Also, hackers can compromise smart contracts by applying other methods that are typical for compromising blockchain technology, including DDoS, eclipse, and various low-level attacks.

**Slide 8**

**ATTACKS ON SMART CONTRACTS**

**Front-running aka transaction-ordering dependence**

* The University of Concordia considers front-running to be “a course of action where an entity benefits from prior access to privileged market information about upcoming transactions and trades.” This knowledge of future events in a market can lead to exploitation.
* For example, knowing a very large purchase of a specific token is going to occur, a bad actor can purchase that token in advance and sell the token for a profit when the oversized buy order increases the price.
* Front-running attacks have long been an issue in financial markets, and due to blockchain’s transparent nature, the problem is coming up again in cryptocurrency markets.
* Since the solution to this problem varies on a per-contract basis, it can be hard to protect against. Possible solutions include batching transactions and using a precommit scheme (i.e., allowing users to submit details at a later time).

**DoS with block gas limit**

* In the Ethereum blockchain, the blocks all have a gas limit. One of the benefits of a block gas limit is it prevents attackers from creating an infinite transaction loop, but if the gas usage of a transaction exceeds this limit, the transaction will fail. This can lead to a DoS attack in a couple different ways.

1. **Unbounded operations**

* A situation in which the block gas limit can be an issue is in sending funds to an array of addresses. Even without any malicious intent, this can easily go wrong. Just by having too large an array of users to pay can max out the gas limit and prevent the transaction from ever succeeding.
* This situation can also lead to an attack. Say a bad actor decides to create a significant amount of addresses, with each address being paid a small amount of funds from the smart contract. If done effectively, the transaction can be blocked indefinitely, possibly even preventing further transactions from going through.
* An effective solution to this problem would be to use a pull-payment system over the current push-payment system. To do this, separate each payment into its own transaction and have the recipient call the function.

**Block stuffing**

* In some situations, your contract can be attacked with a block gas limit even if you don’t loop through an array of unspecified length. An attacker can fill several blocks before a transaction can be processed by using a sufficiently high gas price.
* This attack is done by issuing several transactions at a very high gas price. If the gas price is high enough and the transactions consume enough gas, they can fill entire blocks and prevent other transactions from being processed.
* Ethereum transactions require the sender to pay gas to disincentivize spam attacks, but in some situations, there can be enough incentive to go through with such an attack. For example, a block stuffing attack was used on a gambling Dapp, Fomo3D. The app had a countdown timer, and users could win a jackpot by being the last to purchase a key — except every time a user bought a key, the timer would be extended. An attacker bought a key then stuffed the next 13 blocks in a row so they could win the jackpot.
* To prevent such attacks from occurring, it’s important to carefully consider whether it’s safe to incorporate time-based actions in your application.

**DoS with (unexpected) revert**

* DoS (denial-of-service) attacks can occur in functions when you try to send funds to a user and the functionality relies on that fund transfer being successful.
* This can be problematic in the case that the funds are sent to a smart contract created by a bad actor, since they can simply create a fallback function that reverts all payments.

## Forcibly sending Ether to a contract

* Occasionally, it’s unwanted for users to be able to send Ether to a smart contract. Unfortunately for these circumstances, it’s possible to bypass a contract fallback function and forcibly send Ether.

## Insufficient gas griefing

* Griefing is a type of attack often performed in video games, where a malicious user plays a game in an unintended way to bother other players, aka trolling. This type of attack is also used to prevent transactions from being performed as intended.
* This attack can be done on contracts which accept data and use it in a subcall on another contract. This method is often used in multisignature wallets as well as transaction relayers. If the subcall fails, either the whole transaction is reverted, or execution is continued.

## Reentrancy

Reentrancy is an attack that can occur when a bug in a contract function can allow a function interaction to proceed multiple times when it should otherwise be prohibited. This can be used to drain funds from a smart contract if used maliciously. In fact, reentrancy was the attack vector used in the DAO hack.

1. **Single-function reentrancy**

A single-function reentrancy attack occurs when a vulnerable function is the same function an attacker is trying to recursively call.

1. **Cross-function reentrancy**

A cross-function reentrancy attack is a more complex version of the same process. Cross-function reentrancy occurs when a vulnerable function shares a state with a function an attacker can exploit.

**Slide 9**

**Vulnerabilities**

Nearly all vulnerabilities listed here can be found in the Smart Contract Weakness Classification.

* **Integer overflow and underflow**

In solidity, integer types have maximum values. For example:

uint8 => 255

uint16 => 65535

uint24 => 16777215

uint256 => (2^256) - 1

Overflow and underflow bugs can occur when you exceed the maximum value (overflow) or when you go below the minimum value (underflow). When you exceed the maximum value, you go back down to zero, and when you go below the minimum value, it brings you back up to the maximum value.

Since smaller integer types — like uint8, uint16, etc. — have smaller maximum values, it can be easier to cause an overflow; thus, they should be used with greater caution.

Likely, the best available solution to overflow and underflow bugs is to use the OpenZeppelin SafeMath library when performing mathematical operations.

* **Timestamp dependence**

The timestamp of a block, accessed by now or block.timestamp, can be manipulated by a miner. There are three considerations you should consider when using a timestamp to execute a contract function.

1. **Timestamp manipulation**

If a timestamp is used in an attempt to generate randomness, a miner can post a timestamp within 15 seconds of block validation, giving them the ability to set the timestamp as a value that’d increase their odds of benefiting from the function.

For example, a lottery application may use the block timestamp to pick a random bidder in a group. A miner may enter the lottery then modify the timestamp to a value that gives them better odds at winning the lottery. Timestamps should thus not be used to create randomness.

1. **The 15-second rule**

Ethereum’s reference specification, the “Yellow Paper,” doesn’t specify a limit as to how much blocks can change in time - it just has to be bigger than the timestamp of it’s parent. Popular protocol implementations reject blocks with timestamps greater than 15 seconds in the future, so as long as your time-dependent event can safely vary by 15 seconds, it’s safe to use a block timestamp.

Don’t use block.number as a timestamp. You can estimate the time difference between events using block.number and the average block time. But block times may change and break the functionality, so it's best to avoid this use.

Authorization through tx.origin, tx.origin is a global variable in Solidity which returns the address that sent a transaction. It's important you never use tx.origin for authorization since another contract can use a fallback function to call your contract and gain authorization since the authorized address is stored in tx.origin. Consider this example:

1. **Floating pragma**

It’s considered best practice to pick one compiler version and stick with it. With a floating pragma, contracts may accidentally be deployed using an outdated or problematic compiler version - which can cause bugs, putting your smart contract’s security in jeopardy. For open-source projects, the pragma also tells developers which version to use should they deploy your contract. The chosen compiler version should be thoroughly tested and considered for known bugs.

The exception in which it’s acceptable to use a floating pragma is in the case of libraries and packages. Otherwise, developers would need to manually update the pragma to compile locally.

1. **Function default visibility**

Function visibility can be specified as either public, private, internal, or external. It’s important to consider which visibility is best for your smart contract function.

Many smart contract attacks are caused by a developer forgetting or forgoing to use a visibility modifier. The function is then set as public by default, which can lead to unintended state changes.

* **Outdated compiler version**

Developers often find bugs and vulnerabilities in existing software and make patches. For this reason, it’s important to use the most recent compiler version possible. See bugs from past compiler versions here.

* **Unchecked call-return value**

If the return value of a low-level call is not checked, the execution may resume even if the function call throws an error. This can lead to unexpected behavior and break the program logic. A failed call can even be caused by an attacker, who may be able to further exploit the application.

In Solidity, you can either use low-level calls such as address.call(), address.callcode(), address.delegatecall(), and address.send(), or you can use contract calls such as ExternalContract.doSomething(). Low-level calls will never throw an exception- instead they will return false if they encounter an exception, whereas contract calls will automatically throw.

In the case that you use low-level calls, be sure to check the return value to handle possible failed calls.

* **Unprotected Ether withdrawal**

Without adequate access controls, bad actors may be able to withdraw some or all the Ether from a contract. This can be caused by misnaming a function intended to be a constructor, giving anyone access to reinitialize the contract. To avoid this vulnerability, only allow withdrawals to be triggered by those authorized or as intended and name your constructor appropriately.

* **Unprotected self-destruct instruction**

In contracts that have a self-destruct method, if there are missing or insufficient access controls, malicious actors can self-destruct the contract. It's important to consider whether self-destruct functionality is absolutely necessary. If it’s necessary, consider using a multisig authorization to prevent an attack.

This attack was used in the Parity attack. An anonymous user located and exploited a vulnerability in the “library” smart contract, making themselves the contract owner. The attacker then proceeded to self-destruct the contract. This led to funds being blocked in 587 unique wallets, holding a total of 513,774.16 Ether.

* **State variable default visibility**

It’s common for developers to explicitly declare function visibility but not so common to declare variable visibility. State variables can have one of three visibility identifiers: public, internal, or private. Luckily, the default visibility for variables is internal and not public, but even if you intend on declaring a variable as internal, it's important to be explicit so there are no incorrect assumptions as to who can access the variable.

* **Uninitialized storage pointer**

Data is stored in the EVM as either storage, memory, or calldata. It’s important the two are well understood and correctly initialized. Incorrectly initializing data-storage pointers, or simply leaving them uninitialized, can lead to contract vulnerabilities.

As of Solidity 0.5.0, uninitialized storage pointers are no longer an issue since contracts with uninitialized storage pointers will no longer compile. This being said, it's still important to understand what storage pointers you should be using in certain situations.

* **Assert violation**

In Solidity 0.4.10, the following functions were created: assert(), require(), and revert(). Here, we'll discuss the assert function and how to use it.

Formally said, the assert() function is meant to assert invariants; informally said, assert() is an overly assertive bodyguard that protects your contract but steals your gas in the process. Properly functioning contracts should never reach a failing assert statement. If you've reached a failing assert statement, you've either improperly used assert() or there is a bug in your contract that puts it in an invalid state.

If the condition checked in the assert() is not actually an invariant, it's suggested that you replace it with a require() statement.

* **Use of deprecated functions**

As time goes by, functions in Solidity are deprecated and often replaced with better functions. It’s important to not use deprecated functions, as it can lead to unexpected effects and compilation errors.

Here’s a list of deprecated functions and alternatives. Many alternatives are simple aliases and won’t break current behavior if used as a replacement for its deprecated counterpart.

* **Delegate call to untrusted callee**

Delegate call is a special variant of a message call. It’s almost identical to a regular message call except the target address is executed in the context of the calling contract and msg.sender and msg.value remain the same. Essentially, delegate call delegates other contracts to modify the calling contract's storage.

Since delegate call gives so much control over a contract, it's very important to only use this with trusted contracts, such as your own. If the target address comes from user input, be sure to verify that it’s a trusted contract.

* **Signature malleability**

Often, people assume the use of a cryptographic signature system in smart contracts verifies that signatures are unique; however, this isn’t the case. Signatures in Ethereum can be altered without the private key and remain valid. For example, elliptic-key cryptography consists of three variables v, r, and s and if these values are modified in just the right way, you can obtain a valid signature with an invalid private key.

To avoid the problem of signature malleability, never use a signature in a signed message hash to check if previously signed messages have been processed by the contract because malicious users can find your signature and recreate it.

* **Incorrect constructor name**

Before Solidity 0.4.22, the only way to define a constructor was by creating a function with the contract name. In some cases, this was problematic. For example, if a smart contract is reused with a different name but the constructor function isn't also changed, it simply becomes a regular, callable function.

Now with modern versions of Solidity, you can define the constructor with the constructor keyword, effectively deprecating this vulnerability. Thus, the solution to this problem is simply to use modern Solidity compiler versions.