Transactions - Function call

1. Nonce: tx count for the account
2. Gas Price: price per unit of gas(in wei)
3. Gas Limit: max gas that this tx can use
4. To: address that the tx is sent to
5. Value: amount of wei to send
6. Data: what to send to the To address
7. v,r,s: components of tx signature

Transactions - Contract Deployment

1. Nonce: tx count for the account
2. Gas Price: price per unit of gas(in wei)
3. Gas Limit: max gas that this tx can use
4. To: empty
5. Value: amount of wei to send
6. Data: contract init code & contract bytecode
7. v,r,s: components of tx signature

> We can send the data field of the tx ourselves with our function call hexcode
> call: how we call functions to change the state of the blockchain
> staticcall: this is how (at a low level) we do our "view" or "pure" function calls, and potentially don't change the blockchain

```
tx = {
nonce: nonce,
gasPrice: 100000000,
gasLimit: 100000,
to: null,
value: 0,
data: "0x608060...0f0345", //EVM level data, low level
chainId: 1337,
}
```

> For example-
> (bool success,) = recentWinner.call{value: address(this).balance}("");
> Here, we change the value (Line 5) field of a transaction directly, the parantheses at the end is where we put the data (Line 6)
> In our {} we are able to pass specific fields of a transaction, like value
> In our () we are able to pass data in order to call a specific function but there was no function we wanted to call in Line 34.

selfdestruct is a keyword in solidity that is used to delete/destroy a contract.
Proxies can be used to make changes to a smart contract. Kinda negates the "decentralization" part of blockchain. Leads to rag-pulling.

Rough approximation of audit duration

> LoC : Duration -
> 100 : 2.5 days
> 500 : 1 week
> 1000 : 1-2 weeks
> 2500 : 2-3 weeks
> 5000 : 3-5 weeks
> 5000+ : 5+ weeks

Findings are listed by severity in a report - Highs, Mediums, Lows, Informational, Gas efficiencies and non-critical (Last 3 are not vulnerabilities, but ways to improve code).

Tests to see if a protocol is ready for auditing:
GitHub repo -> [1]
The Rekt Test

Tools for auditing

> Test Suites:
> Hardhat, Foundry, Brownie, etc.

> Static Analysis:
> Slither, Aderyn, Mythril

> Fuzzing:
> Foundry, Echidna

> Formal Verification:
> Symbolic Execution - MAAT, Z3, Manticore

Machine-findable bugs and non-machine-findable bugs - [2]
Resource for building secure contracts - [10]

Note-taking methodology

> Bug: //!
> For my information: //i
> Questions: //q
> Potential issue: //@audit
> Explanation: //e
> Make sure to follow up: //@follow-up
> Informational audit note: //@audit-info
> Code location for a filed issue: //@audit-issue
> Not an issue, even if it looks like one: //@audit-ok
> Notes: //@note
> Todo remark: //@todo
> Reminder: //@remind

Look for questions after scoping and figure out answers
Check code coverage with "forge coverage"

Findings

> 1. Convince the protocol this is an issue
> 2. How bad this issue is
> 3. How to fix the issue

Layouts and notes on reports - [3]
Severity Guide - [4]

Reading a private variable in a smart contract

> 1. Create a locally running chain
>
> ```
> make anvil
> ```
>
> 2. Deploy the contract to the chain
>
> ```
> make deploy
> ```
>
> 3. Run the storage tool
>   We use `1` because that's the storage slot of `PasswordStore::s_password` in the contract.
>
> ```
> cast storage <ADDRESS HERE> 1 --rpc-url http://127.0.0.1:8545
> ```
>
> You'll get an output that looks like this:
> `0x6d7950617373776f726400000000000000000000000000000000000000000014`
>
> You can then parse that hex to a string with:
>
> ```
> cast parse-bytes32-string
> 0x6d7950617373776f726400000000000000000000000000000000000000000014
> ```
>
> And get an output of:
>
> ```
> myPassword
> ```

Scoping process

Refer to minimal-onboarding in `[3]`
Check branch

```
git branch
```

Check the commit hash of the scope and then type

```
git checkout <COMMIT HASH HERE>
```

Switch to a new branch

```
git switch -c NewProject-audit
```

Reentrancy Soln.

```
bool locked = false;
//locked is set to see if the function has been entered once, if yes, then
revert if tried to reenter
function withdrawBalance() public {
    if(locked) revert();
    locked = true;

    uint256 balance = userBalance[msg.sender];
    userBalance[msg.sender] = 0;

    (bool success, ) = msg.sender.call{value: balance}("");
    if(!success) {
        revert();
    }

    locked = false;
}
```

Can use OpenZeppelin's `ReentrancyGuard.sol::nonReentrant()` [5] which essentially does the same thing under the hood.
A Historical Collection of Reentrancy Attacks -> [6]

Weak Randomness

```
uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.sender,
block.timestamp, block.difficulty))) % players.length;
```

The values used to create a pseudo-random number can be manipulated or predicted and can be used to get the random number.
fixes: Chainlink VRF, Commit Reveal Scheme

Integer overflow and underflow

Use `chisel` in cmd and then `type(<DATATYPE>).max` to check the maximum value of a datatype. Comes with foundry.

Self Destruct

If a contract doesn't have a `receive` or `fallback` function, it reverts any attempts to send it any value but an attackerSelfDestruct contract can self-destruct and force the contract to accept the money. This will cause the `address(this).balance == totalBalance` check to fail.

Check this smart contract for an example - [7]

Invariants

Invariants in ERC20 and ERC721 - [7]

1. Stateless Fuzzing - Open

> Stateless fuzzing (often known as just "fuzzing") is when you provide random data to a function to get some invariant or property to break.
>
> It is "stateless" because after every fuzz run, it resets the state, or it starts over.

2. Stateful Fuzzing - Open

> Stateful fuzzing is when you provide random data to your system, and for 1 fuzz run your system starts from the resulting state of the previous input data.
>
> Or more simply, you keep doing random stuff to the same contract.

3. Stateful Fuzzing - Handler

> Handler based stateful fuzzing is the same as Open stateful fuzzing, except we restrict the number of "random" things we can do.
>
> If we have too many options, we may never randomly come across something that will actually break our invariant. So we restrict our random inputs to a set of specfic random actions that can be called.

4. Formal Verification

> Formal verification is the process of mathematically proving that a program does a specific thing, or proving it doesn't do a specific thing.
>
> For invariants, it would be great to prove our programs always exert our invariant.
>
> One of the most popular ways to do Formal Verification is through Symbolic Execution. Symbolic Execution is a means of analyzing a program to determine what inputs cause each part of a program to execute. It will convert the program to a symbolic expression (hence its name) to figure this out.

foundry.toml ->

```
[fuzz]
runs = 256
seed = '0x2'

[invariant]
runs = 64
depth = 32
fail_on_revert = false
```

`seed` helps input the randomness. Different seed == different random runs.
`fail_on_revert = false` implies that we only care about the invariant breaking and no other reverts matter. Other reverts executed if set to true.

Two types of invariants:

1. **Function-level invariants:**

- Doesn't rely on much of the system OR could be stateless
- Can be tested in an isolated fashion
- Examples: Associative property of addition OR depositing tokens in a contract

2. **System-level invariants:**

- Relies on the deployment of a large part or entire system
- Invariants are usually stateful
- Examples: user's balance < total supply OR yield is monotonically increasing

Weird ERC20s

It is very important to know what tokens one protocol is working with in order to check for weird ERC20 tokens. They can break our contract with their strange states/rules.

Weird ERC20 list - [9]

Tincho Method

After you are done running your automated tools(Slither, Aderyn, etc.), copy down all the in-scope contracts' details (nSLOC, complexity score, etc.) from the Solidity Metrics report and paste it in an Excel sheet. Then sort the sheet in order of ascending complexity score or nSLOC and go through each of them from lowest to highest.

Invariant Testing

1. Go through the documentation and look for invariants before even looking at the code
2. Categorize your invariants based on their properties
3. Write invariants in order of priority
4. Bound values to not waste fuzz runs

```
// GOOD
function testDeposit1(uint256 amount) public {
    amount = bound(amount, 0, address(this).balance);
    // ...
}

// BAD
function testDeposit2(uint256 amount) public {
    vm.assume(amount > 0); // When 0 is passed as a value, it will not pass
through this cheatcode, and will go to the next fuzz run with a different
value
    vm.assume(amount < address(this).balance);
    // ...
}
```

5. Code your tests using the Hoare logic (preconditions, then actions, then postconditions)

```
function addLiquidity(uint amount1, uint amount2) public {
    // PRECONDITIONS
    amount1 = clampBetween(...);
    amount2 = clampBetween(...);

    // ACTIONS
    bool success = _addLiquidity(amount1, amount2);

    // POSTCONDITIONS
    if(succes) {
        // ...
    }
}
```

6. Can take about a week to set up your invariant tests. Take a day to figure out all the invariants to test. Second day to set up your environment, foundry invariant tests, etc. You have three days to kind of fine-tune your tests, think about all the scenarios you want to test. Takes time to set it up if a lot of external libraries used.

7. Use ghost variables to check "before/after"

```
function deposit(uint256 assets) public virtual {
    asset.mint(address(this), assets);

    asset.approve(address(token), assets);

    uint256 shares = token.deposit(asset, address(this));
    // sumBalanceOf is a ghost variable which can be later checked against
token contract's total shares
    sumBalanceOf += shares;
}
```

8. Check logic against different/deoptimized implementations

```
function invariant_totalDebtEqualsSum() internal view returns (bool) {
    uint256 totalDebt = vault.totalDebt();
    uint256 sum = 0;
    for(uint i = 0; i<positions.length; i++) {
      sum += positions.getDebt();
    }

    return sum == totalDebt;
}
```

9. Use multiple actors for more realistic scenarios

```
modifier useActor(uint256 actorIndexSeed) {
    currentActor = actors[bound(actorIndexSeed, 0, actors.length-1);
    // If your protocol has many roles/actors, you can set up a modifier
like this one and randomly choose an actor for testing
    vm.startPrank(currentActor);
    _;
    vm.stopPrank();
}
```

10. Limit the number of targets and selectors the fuzzer is calling (including state vars). In case of Foundry, make sure to manually select the selectors you want to test and in case of Echidna, make sure your state vars are `internal` as it will try to fuzz the public state vars and waste time as they are read-only.

11. Check both success and failure cases for max coverage with either `try catch` or `if else`. Echidna case -

```
if(success) {
    gt(
        vars.kAfter,
        vars.kBefore,
        "P-01 | Adding Liquidity increases K"
    );
    // ...
} else {
    eq(
        vars.kAfter,
        vars.kBefore,
        "P-08 | Adding liquidity should not change anything if it fails"
    );
    // ...
}
```

12. Think about how your invariants may change depending on the state of the system.
13. Always check the code coverage. In Echidna, you can see the coverage with a coverage report, which is an HTML file in which you can see the parts that have been covered with tests. Shows green for parts covered, yellow for parts that haven't been covered properly and red means parts that have not been covered at all.
14. Reduce the input space of the fuzzer

Echidna

- Needs the `echidna_` prefix for every function identifier for Echidna to be able to realize its inputs. (PROPERTY TESTING)
- `echidna-test <test_file_name> --contract <test_contract_name>` for running Echidna.
- Declare a test function without the echidna prefix, can use `test_`prefix instead, write code for asserting a condition, and run echidna with the following input at the end: `--test-mode assertion`. This allows you to add parameters to the test as well. (ASSERTION TESTING) [11]
- Testing system-level invariants require initialization
    - **Simple initialization**
        - Deploy everything in the constructor
    - **Complex initialization**
        - Leverage your unit tests framework with etheno `//q what is etheno?`
    - **NOTE: Function-level invariants may also need some system initialization**
- Medusa can be used for a more detailed fuzz test
    - `medusa init` is the first command, generates a .json file which can be modified based on our requirements and to avoid using flags in the cmd
        - In `medusa.json`, `targetContracts` sets the contract containing the test functions for fuzzing
        - `assertionTesting` section should have `enabled` set to `true` for assertion testing and `enabled` set to `false` for `propertyTesting`, and vice-versa for property testing
    - `medusa fuzz` is the command to run the test
- Echidna has its own set of cheatcodes in `hevm`
    - They are almost the same as the cheatcodes in Foundry

Decoding function calls in MetaMask

Before approving/confirming a transaction on MetaMask, check the data section and the hex section, run `cast sig <function-sig>` to check the desired hex, and if it doesn't match, might be a malicious call. If there are parameters passed, run `cast --calldata-decode <function-sig> <calldata/hex value>` and it will tell you what parameters were passed.

Generating a random number (Chainlink VRF)

1st step: `docs.chain.link/vrf`
2nd step: Create a subscription in subscription manager
3rd step: Create a fund the subscription
4th step: Use the subscription ID while deploying the contract
5th step: Add a consumer contract to use the random number by giving the deployed contract address
Check [12] for sample code using Chainlink VRF and natspec
You need to mention how and where you are using the RNG in your contract and use it in the `fulfillRandomWords` function

Chainlink Automation (Keepers)

1st step: `docs.chain.link/chainlink-automation`
2nd step: You will mostly be using a custom-based upkeep, so register one
3rd step: There are two functions that Chainlink needs to automate your code: `checkUpkeep()` and `performUpkeep()`. Add those functions
4th step: `checkUpkeep() returns(bool upkeepNeeded,)` checks the conditions required for the next automated call and if they are met, it will call `performUpkeep()`.
Check [12] for sample code using Chainlink Automation and natspec

Useful Links

> 1. **openchain.xyz** is a hex values database, can search random hex values to decrypt them
> 2. **codeslaw.app** is a tool to look up function signatures called anywhere on the mainnet
> 3. **etherscan.deth.net** while looking at any verified contracts on `etherscan.io`, change the postfix to `.deth.net` to see the code in a virtual VSC sim
> 4. **Ethereum EIPs**
> 5. **ZKSync documentation**

References

[1]nascentxyz / simple-security-toolkit
[2]ZhangZhuoSJTU / Web3Bugs
[3]Cyfrin / security-and-auditing-full-course-s23
[4]docs.codehawks.com - How to determine a finding severity
[5]@openzeppelin/contracts/util/ReentrancyGuard.sol
[6]pcaversaccio / reentrancy-attacks
[7]Cyfrin / sc-exploits-minimized - src/mishandling-of-eth
[8]crytic / properties
[9]d-xo / weird-erc20
[10]secure-contracts.com
[11]crytic / building-secure-contracts / program-analysis / echidna / exercises / exercise2
[12]KatrixReloaded / FoundryLottery