Smart Contracts Exercise 08: Maximal Extractable Value

1 Introduction

We first encountered frontrunning in Exercise 3: ERC20 - CTUToken. Frontrunning falls under the more general term known as MEV - Maximal Extractable Value. MEV has become a very important topic for Ethereum that needs to be addressed. In this exercise, we will examine this phenomenon in more detail. We'll explore additional MEV techniques and examples, such as DEX arbitrage, liquidations, sandwich trading, and more. An important topic of the exercise will also be calculating transaction fees, gas, and EIP-1559. In the practical exercises, you will try frontrunning an NFT auction and executing a sandwich attack on a DEX yourself. Finally, we'll look at how MEV is being solved today and introduce terms such as Proposer-Builder Separation and Builder APIs.

Project Setup

You have two options for working with this exercise. Using docker container or local installation. Choose the one that best fits your preferences.

1.1 Using Docker with VS Code

This option uses Docker to create a development environment with all the necessary tools and dependencies pre-installed.

Prerequisites:

- **Docker** - A platform for developing, shipping, and running applications in containers.
- **Visual Studio Code** - A lightweight but powerful source code editor.
- **Dev Containers** - An extension to VS Code that lets you use a Docker container as a full-featured development environment.

Setting Up the Project:

1. Visit the following GitLab repository and clone it to your local machine.
2. Open the repository folder in VS Code.
3. When prompted, click "Reopen in Container" or use the command palette (F1) and run `Dev Containers: Reopen in Container`.

1.2 Local Setup

If you prefer working directly on your machine without Docker, you can set up the development environment locally.

Prerequisites

- **Node.js** - https://nodejs.org/en/ - An open-source, cross-platform, back-end JavaScript runtime environment that runs on the V8 engine and executes JavaScript code outside a web browser.
- **NPM**: Node Package Manager, which comes with Node.js.

Open your terminal and run the following commands to verify the installations:

```
$ node -v
$ npm -v
```

Both commands should return the installed version numbers of Node.js and NPM respectively. Node.js provides the runtime environment required to execute JavaScript-based tools like Hardhat, while NPM is used to manage the packages and dependencies needed for development.

Setting Up the Project

1. Visit the following GitLab repository and clone it to your local machine.
2. Open a terminal and navigate to the project directory.
3. Install the project dependencies by running `npm install`.

2 Maximal Extractable Value (MEV)

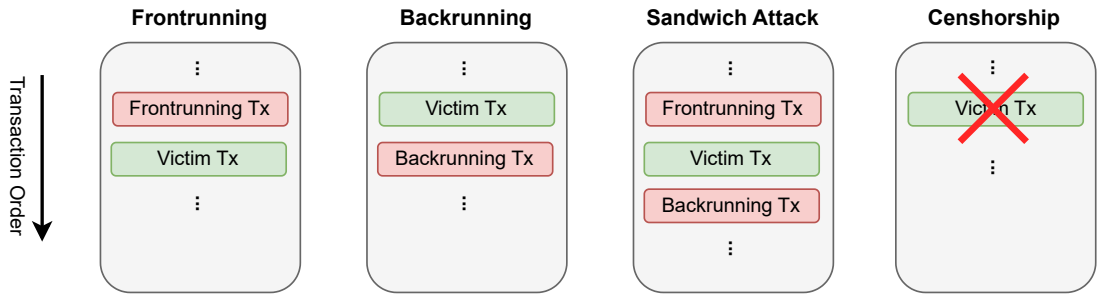
MEV stands for Maximal Extractable Value, which refers to the maximum value that can be extracted during block production in addition to the standard block rewards and gas fees. This value can be extracted by adding transactions to a block, removing transactions from it, or changing the order of transactions within the block. Originally, the acronym stood for "Miner Extractable Value," but since the Merge replaced miners with validators, it's now known as "Maximal Extractable Value".

Types of MEV

Several types of MEV extraction can be identified:

1. **Frontrunning**: An attacker observes a pending transaction in the mempool and submits their own transaction to be included in the block before the original transaction.
2. **Backrunning**: An attacker observes a pending transaction in the mempool and submits their own transaction to be included in the block immediately after the original transaction.

3. **Sandwich Attack**: An attacker places two transactions around the victim's transaction - one before and one after.
4. **Transaction Censorship**: An attacker omits a transaction from the block to prevent it from being executed.



To better understand why these techniques are profitable for attackers, let's examine some specific examples:

DEX Arbitrage

If two DEXes offer tokens at different prices, someone can include a transaction in a block that buys tokens on one DEX and sells them at a higher price on another DEX. In this situation, the block creator can exploit this opportunity to profit from the arbitrage. Here's an example of such transaction.

Sandwich Attack

If someone wants to exchange a large amount of a specific token on a DEX, an attacker can add transactions that buy tokens before the large transaction and sell them afterward. Thanks to how DEXes set prices using the constant product formula ($x \cdot y = k$), this approach can be profitable. See an example of such attack. The process is detailed in Table 1 and illustrated in Figure 1.

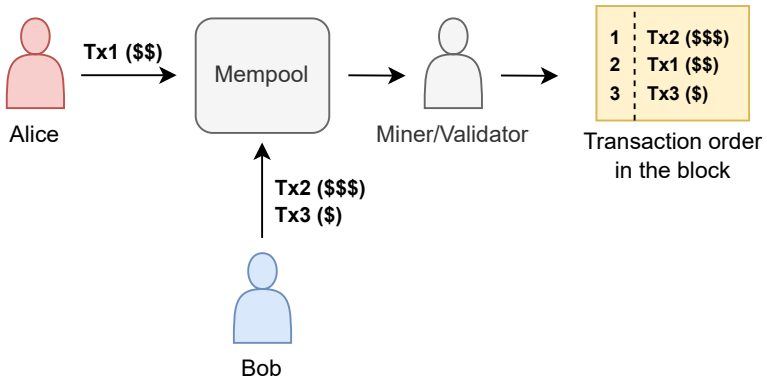


Figure 1: Sandwich Attack visualization showing how transactions are ordered in the mempool and block.

Step	Action	Description
1	Alice creates transaction	Alice submits transaction Tx1 ($$) to the mempool to swap a large amount of tokens on a DEX.
2	Bob monitors mempool	Bob identifies Alice's pending transaction as an opportunity for a sandwich attack.
3	Bob creates frontrun	Bob submits transaction Tx2 ($$$) with higher gas fees to buy the same tokens before Alice's swap executes.
4	Token price increases	After Bob's purchase, token price increases according to $P_{\text{new}} = \frac{y-\Delta y}{x+\Delta x}$ where Δx is Bob's input and Δy is tokens received.
5	Alice's transaction executes	Alice's swap executes at worse price conditions due to Bob's prior purchase, receiving fewer tokens than expected.
6	Token price increases further	Alice's large swap further increases the token price due to the constant product formula, making Bob's tokens worth more.
7	Bob executes backrun	Bob's transaction Tx3 ($) sells the tokens at the higher price caused by Alice's large swap.
8	Bob profits	Bob profits from the price difference: Profit = Sell proceeds − Purchase cost − Gas fees.
9	Price impact on Alice	Alice experiences negative price impact, receiving fewer tokens than expected in a fair market.

Table 1: Sandwich Attack Mechanism on a DEX. The attack exploits the AMM pricing formula and transaction ordering. For visual representation, see Figure 1.

Liquiditations

When borrowers in lending protocols fall below required collateralization ratios, their positions become eligible for liquidation. Block producers can prioritize their own liquidation transactions, allowing them to claim the liquidation rewards and discounted collateral before other participants. This gives them an advantage in capturing value from distressed positions. See examples of such transactions.

Other MEVs

MEV opportunities are derived from smart contracts and their functionalities. Consider, for example, the auction from the previous exercise. If the block creator is interested in the auctioned item, they can simply add their transaction at the end of the auction, placing it before the transaction that ends the auction.

See also this example where a MEV searcher purchased every single Cryptopunk at the floor price for 7 million USD: Searcher's address.

Generalized frontrunning

Rather than developing complicated algorithms, it's possible to simply simulate transactions in the mempool. If a transaction results in profit, a frontrunner can copy the potentially profitable transaction and replace the address with their own. If the transaction still generates profit with the modified address, the attacker can include it in the block and claim the profit for themselves.

2.1 MEV Threat and Solutions

MEV bots attempting to include their MEV transactions in blocks have led to network congestion and increased transaction fees for regular users. MEV extraction could

eventually lead to gradual centralization of staking pools, as large pools will have more financial resources to develop sophisticated algorithms to extract MEV from their own MEV extraction activities. To address these MEV problems, solutions have been proposed: Proposer-builder separation and Builder APIs. In proposer-builder separation, validators remain responsible for proposing new blocks, but a new entity of specialized builders has emerged who search for MEV opportunities and offer constructed blocks to validators. This reduces the risk of validator centralization. For smart contract developers, it's important to always keep MEV risks in mind and prevent them during application design when possible.

More about MEV:

- **Read:** Maximal extractable value (MEV)
- **Animated Video:** Decoding MEV: Past, Present, Future
- **MEV Real Time Detection:** https://eigenphi.io
- **Read:** Flashbots research blog
- **Block Building Visualisation:** https://payload.de/data
- **MEV Boost Dashboard:** https://mevboost.pics

3 Gas and Fees

Gas is a unit that measures the computational effort required to execute a specific operation on the Ethereum network. The **gas fee** is the amount of gas used for a particular operation multiplied by the price per unit of gas. The costs of individual operations can be viewed at EVM Opcodes.

Gas Price Determination

Previously, users would specify a `gasprice` which operated as a simple auction system - higher gas price meant higher priority. However, this solution had several drawbacks, so Ethereum transitioned to a new transaction fee mechanism according to EIP-1559. Instead of users trying to estimate how much to bid, the Ethereum protocol now determines a **base fee** that must be paid for a transaction to be included in a block. This base fee dynamically increases or decreases by a maximum of 12.5% with each block. Each block has a limit of 30 million gas, but the target is 15 million:

- If a block used less than 15 million gas, the base fee decreases
- If the block was congested, the base fee increases

The base fee doesn't go to validators but is **burned** (removed from circulation). EIP-1559 also introduced a **priority fee**, which goes directly to validators as a tip to incentivize them to include the transaction in a block. Transaction initiators can also specify a **maximum gas price** (`max fee`) they're willing to pay per unit of gas. The actual fee per gas paid by a transaction is calculated as:

$$\text{gasprice} = \min(\text{base fee} + \text{priority fee}, \text{max fee}) \tag{1}$$

Where:

- `base fee` is burned (removed from circulation)
- `priority fee` (or tip) goes to the validator as an incentive
- If `base fee` + `priority fee` > `max fee`, the priority fee is reduced to stay below max fee

Any difference between the max fee and the base fee is refunded to the transaction initiator. The total fee paid for a transaction is:

$$\text{transaction fee} = \text{gas used} \times \text{gasprice} \tag{2}$$

More about Gas and Fees:

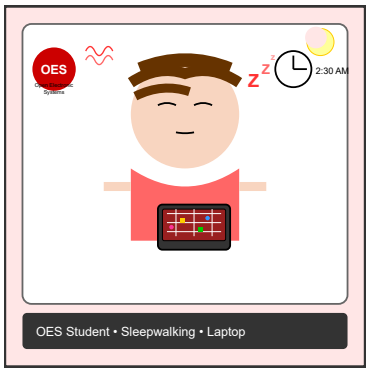
- **Read:** Gas and Fees
- **Gas tracker:** https://etherscan.io/gastracker
- **Video:** Can ETH Become DEFLATIONARY? EIP 1559 Explained

4 Task

Task 1: NFT Auction Frontrunning

Another rare NFT from the FEL Student Collection is up for auction - specifically, a student from the OES program caught sleepwalking with a laptop (apparently a common occurrence during exam season). This time the auction is well-protected against DoS attacks and fully implements the push-over-pull pattern from the previous exercise. The bidding has already reached 1.5 ETH. You don't want to place bids until the last possible moment to avoid being outbid. You've been monitoring the public mempool, waiting for the event organizer to call the `endAuction()` function so you can be the final bidder.

With 1.51 ETH in your wallet and a strong desire for this NFT, you're watching closely for your chance to place a bid just before the auction officially ends...



Your Mission:

- Monitor the mempool for the `endAuction()` transaction
- Gain the OES student NFT

Code your solution in the `test/NFTAuction.js` file. Use the player account only. See Ethers v6 transaction documentation to learn how to specify transaction fees. Verify your solution by running:

```
$ npm run auction
```

Files that are relevant for this challenge:

- test/**NFTAuction.js**: The test file where you should code your solution.
- contracts/**NFTAuction.sol**: The improved auction contract.
- contracts/**FELStudentNFT.sol**: The NFT collection contract.

Task 2: Sandwich Attack on a DEX

You've been monitoring the mempool and spot a juicy opportunity: someone (let's call them "Innocent Victim") is about to swap a massive 20 ETH for USDC tokens on SimpleDEX. With your knowledge of DeFi and MEV, you realize this is a perfect opportunity for a sandwich attack.

You have 15 ETH at your disposal, and your blockchain professor would be so proud to see you apply your learnings in real life (or maybe not...).

Your Mission:

- Execute a sandwich attack on the victim's transaction
- Make a profit of at least 1 ETH from this MEV extraction

Code your solution in the `test/SandwichAttack.js` file. Use the player account for your transactions. Verify your solution by running:

```
$ npm run sandwich
```

Files that are relevant for this challenge:

- test/**SandwichAttack.js**: The test file where you should code your solution.
- contracts/**SimpleDEX.sol**: A decentralized exchange with a one liquidity pool containing 500 ETH and 1000000 USDC.
- contracts/**USDCToken.sol**: An ERC20 token representing simplified version of USDC.