Lesson 10

Solidity version 0.8.16 has been released

Important Bugfixes:

- Code Generation: Fix data corruption that affected ABI-encoding of calldata values represented by tuples: structs at any nesting level; argument lists of external functions, events and errors; return value lists of external functions. The 32 leading bytes of the first dynamically-encoded value in the tuple would get zeroed when the last component contained a statically-encoded array.

Compiler Features:

- Code Generator: More efficient code for checked addition and subtraction.
- TypeChecker: Support using library constants in initializers of other constants.
- Yul IR Code Generation: Improved copy routines for arrays with packed storage layout.
- Yul Optimizer: Add rule to convert `mod(add(X, Y), A)` into `addmod(X, Y, A)`, if `A` is a power of two.
- Yul Optimizer: Add rule to convert `mod(mul(X, Y), A)` into `mulmod(X, Y, A)`, if `A` is a power of two.

Bugfixes:

- Commandline Interface: Disallow the following options outside of the compiler mode: `--via-ir`, `--metadata-literal`, `--metadata-hash`, `--model-checker-show-unproved`, `--model-checker-div-mod-no-slacks`, `--model-checker-engine`, `--model-checker-invariants`, `--model-checker-solvers`, `--model-checker-timeout`, `--model-checker-contracts`, `--model-checker-targets`.
- Type Checker: Fix compiler crash on tuple assignments involving certain patterns with unary tuples on the left-hand side.
- Type Checker: Fix compiler crash when `abi.encodeCall` received a tuple expression instead of an inline tuple.
- Type Checker: Fix null dereference in `abi.encodeCall` type checking of free function.

Nomad bridge Hack

Thanks to Rekt and @samczsun for their insights.

On 1st August 2022 the Nomad bridge was hacked and within 3 hours about $190 M of funds were taken. The attack affected related projects such as EVMOS, Milkomeda and Moonbeam.

August 1st 2022

The exploits unfolds...

The exploit was spotted as unusual activity by @spreekaway, see this thread

Nomad bridge getting rugged??? Looks very very sus

Nomad: ERC20 Bridge	OUT	0xe4a4df7e1689589efb0...	100
Nomad: ERC20 Bridge	OUT	0xa8ecaf8745c56d5935c...	100
Nomad: ERC20 Bridge	OUT	0xd1a7feb0317bbe40ac...	100
Nomad: ERC20 Bridge	OUT	bitliq.eth	100
Nomad: ERC20 Bridge	OUT	0xb5c55f76f90cc528b26...	100
Nomad: ERC20 Bridge	OUT	0x000000000000660def...	100
Nomad: ERC20 Bridge	OUT	0xf57113d8f6ff35747737...	100

10:37 PM · Aug 1, 2022 · Twitter Web App

It seemed that transactions were draining funds from the bridge, by sending 0.01 WBTC you could get 100 WBTC back

bitliq.eth	IN	Nomad: ERC20 Bridge	0.01
Nomad: ERC20 Bridge	OUT	0xe4a4df7e1689589efb0...	100
Nomad: ERC20 Bridge	OUT	0xa8ecaf8745c56d5935c...	100

In theory the bridge was controlled by a transfer first being. proved and then a transaction could go through to process the movement of the funds.

So what *should* have been happening is that the messages submitted should have been proven, then included in a merkle tree whose root is stored and flagged as confirmed at a certain time.

What seemed to be happening is that messages were being processed that hadn't previously been confirmed. Once the initial exploiting transaction went through, anyone could simply submit a similar transaction to drain funds for themselves.

Subsequently there were many such transactions from exploiters, white hats , along with the use of MEV techniques to take advantage of the situation.

	! 0x13ea2c98982d1fde14…	Process	15259332	4 days 11 hrs ago	0xe4a4df7e1689589efb0…
	! 0x8183fe98f81deb8aa37…	Process	15259332	4 days 11 hrs ago	Nomad Bridge Exploiter 3
	0xa304a513cc1b62da3d…	Process	15259312	4 days 11 hrs ago	Resident Arbitrageur
	0xe7fd692fcfae783149b…	Process	15259312	4 days 11 hrs ago	Nomad Bridge Exploiter 1
	0xa9ddf5f8c22c33f814a…	Process	15259312	4 days 11 hrs ago	0xa5eff6157b44d7eba6b…
	0xe0112511747dd3b0c3…	Process	15259303	4 days 11 hrs ago	bitliq.eth
	0x39b107d43d88f1120e…	Process	15259303	4 days 11 hrs ago	Nomad Bridge Exploiter 3
	0xcd54193008330871d8…	Process	15259282	4 days 11 hrs ago	Resident Arbitrageur
	! 0x6ae090b41da74ed958…	Process	15259249	4 days 11 hrs ago	Nomad Bridge Exploiter 2
	! 0x370e7192707fa7d4da…	Process	15259240	4 days 11 hrs ago	Nomad Bridge Exploiter 2
	0xef73e48c1be5f025412…	Process	15259240	4 days 11 hrs ago	Nomad Bridge Exploiter 2
	! 0x8732887090364bd5f8…	Process	15259217	4 days 11 hrs ago	Nomad Bridge Exploiter 2
	! 0x282525e58e17d5f442…	Process	15259217	4 days 11 hrs ago	Nomad Bridge Exploiter 2
	! 0xa9108b394beeb02efc…	Process	15259217	4 days 11 hrs ago	Nomad Bridge Exploiter 2
	! 0x51f0ed5db858a85218…	Process	15259202	4 days 11 hrs ago	0xe4a4df7e1689589efb0…
	! 0xbbb009a4b5cb19d8ae…	Process	15259201	4 days 11 hrs ago	bitliq.eth
	! 0x11f468aa89bc97d748…	Process	15259200	4 days 11 hrs ago	Nomad Bridge Exploiter 1
	0x56b4ade16ad35792f0…	Process	15259159	4 days 11 hrs ago	0xe4a4df7e1689589efb0…
	0x73ae1f3a7f81d140e21…	Process	15259156	4 days 11 hrs ago	0xa5eff6157b44d7eba6b…
	0xb0447743d0d29c5756…	Process	15259155	4 days 11 hrs ago	bitliq.eth
	0x5c5f3325d212b8e086…	Process	15259155	4 days 11 hrs ago	Nomad Bridge Exploiter 3
	0x498a0778cc8448a100…	Process	15259104	4 days 12 hrs ago	0xe4a4df7e1689589efb0…
	0xa5fe9d044e4f3e5aa5b…	Process	15259101	4 days 12 hrs ago	bitliq.eth
	0xb1fe26cc8892f58eb46…	Process	15259101	4 days 12 hrs ago	Nomad Bridge Exploiter 3

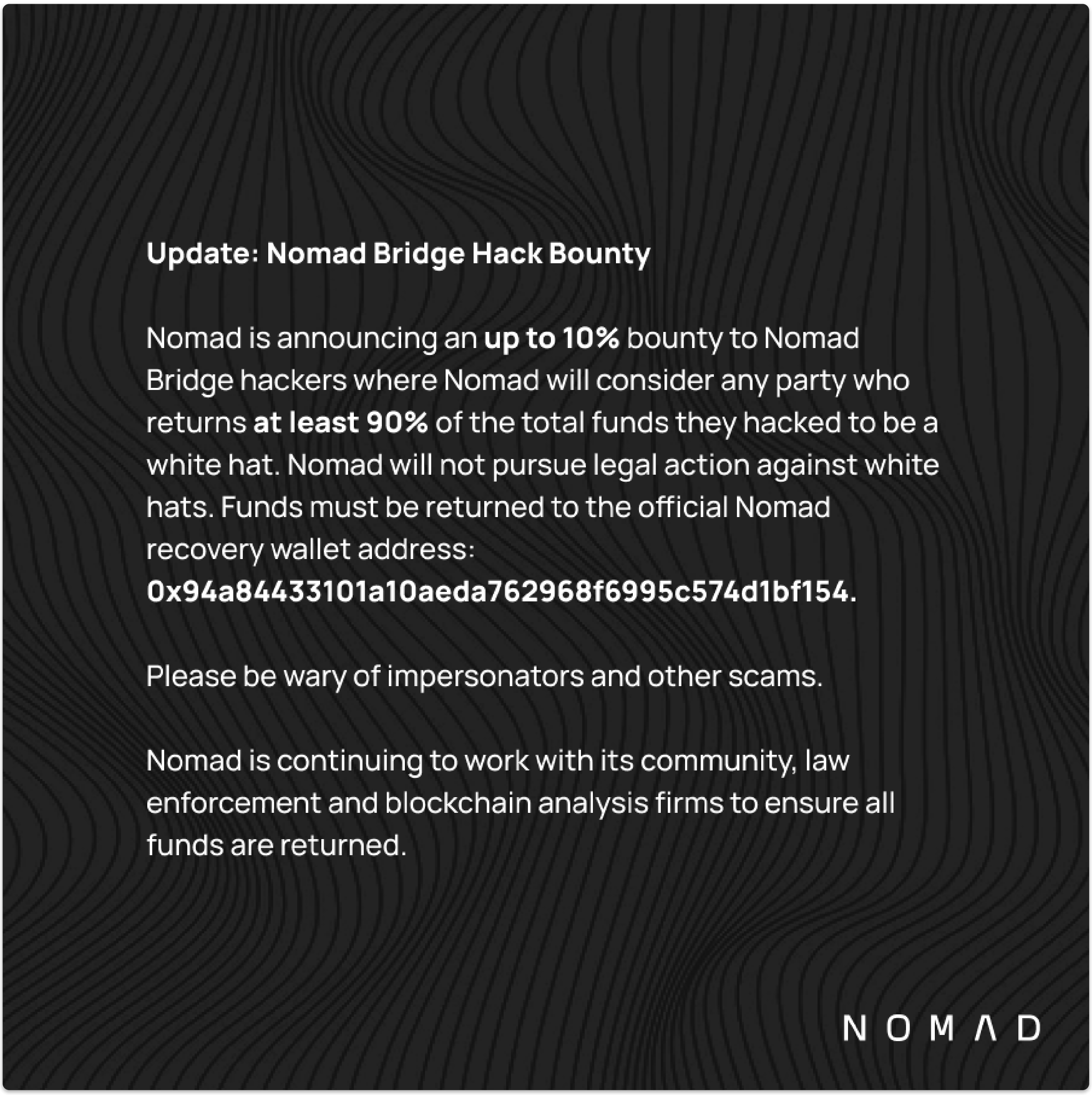
The top 3 exploiters were

0x56D8B635A7C88Fd1104D23d632AF40c1C3Aac4e3 ($47M)
0xBF293D5138a2a1BA407B43672643434C43827179 ($40M)
0xB5C55f76f90Cc528B2609109Ca14d8d84593590E ($8M)

Some exploiters sent their funds through Tornado Cash to obfuscate the trail of funds.

As usual the exploited project appealed for people to return their funds.

Nomad have offered a deal to the exploiters asking for return of 90% of the funds, allowing the other 10% to be regarded as a bug bounty

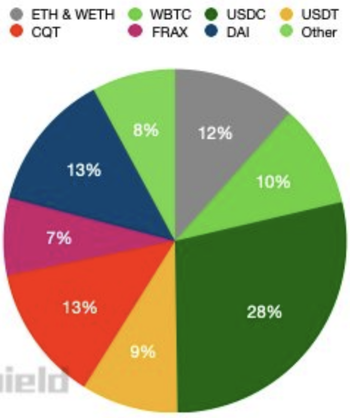


Return of funds

From @PeckShieldAlert as of August 5th, about 11% of funds have been recovered / returned

Nomad Funds Recovery Address

Token	Amount	Value ($)
ETH	241	$399,392
WETH	1,308.86	$2,163,813.5
WBTC	92.7	$2,145,448.8
USDC	6,261,842	$6,268,104
USDT	2,013,335.7	$2,015,349
CQT	34,366,035	$2,836,778.8
FRAX	1,614,400	$1,612,913
DAI	2,879,523.8	$2,882,403
CARDS	158,082.5	$41,911
IAG	247,189,221	$1,308,560
SUSHI	4,557	$6,835
C3	1,649,120	$237,192
GERO	23,101,165.5	$43,500
HBOT	1,200,320	$4,279
FXS	12,503	$93,647



Some white hats have managed to recover some of the funds

Thank you to
- 🍉🍉🍉.eth ($4m)
- 0xE3F40743cc18fd45D475fAe149ce3ECC40aF68c3 ($3.4m)
- darkfi.eth ($1.9m)
- returner-of-beans.eth ($1m)
- anime.eth ($900k)

for returning a total of $11.2m to our recovery address!

We’ve recovered a total of $16.6m so far.

5:19 AM · Aug 4, 2022 · Twitter Web App

Exploit Details

The vulnerable contract is the Replica contract
This contract was upgraded and had the following initializer

```
function initialize(
        uint32 _remoteDomain,
        address _updater,
        bytes32 _committedRoot,
        uint256 _optimisticSeconds
        ) public initializer {

        __NomadBase_initialize(_updater);
        // set storage variables
        entered = 1;
        remoteDomain = _remoteDomain;
        committedRoot = _committedRoot;
        // pre-approve the committed root.
        confirmAt[_committedRoot] = 1;
        _setOptimisticTimeout(_optimisticSeconds);

}
```

It was initialized with the zero address
Transaction

```
Function: initialize(uint32 _remoteDomain, address _updater, bytes32 _committedRoot, uint256
_optimisticSeconds)

MethodID: 0xe7e7a7b7
[0]:  0000000000000000000000000000000000000000000000000000000000000000
[1]:  0000000000000000000000000000000000000000000000000000000000000000
[2]:  0000000000000000000000000000000000000000000000000000000000000000
[3]:  0000000000000000000000000000000000000000000000000000000000000000
```

```
    function initialize(
        uint32 _remoteDomain,
        address _updater,
        bytes32 _committedRoot,
        uint256 _optimisticSeconds
    ) public initializer {
        __NomadBase_initialize(_updater);
        // set storage variables
        entered = 1;
        remoteDomain = _remoteDomain;
```

```
        committedRoot = _committedRoot;
        // pre-approve the committed root.
        confirmAt[_committedRoot] = 1;
        _setOptimisticTimeout(_optimisticSeconds);
    }
```

In the initialize function, committedRoot is set to 0x00

```
        committedRoot = _committedRoot;
        // pre-approve the committed root.
        confirmAt[_committedRoot] = 1;
```

and so
`confirmAt[_committedRoot] = 1;`

The first attempt to exploit the contract failed, it was an expensive failure costing 215 ETH.

Block:	15259103 28418 Block Confirmations
Timestamp:	4 days 10 hrs ago (Aug-01-2022 09:32:59 PM +UTC) | Confirmed within 30 secs
From:	0xb5c55f76f90cc528b2609109ca14d8d84593590e (Nomad Bridge Exploiter 3)
To:	Contract 0x0db09d04d33539e3366de2591e920eba71edc879 Warning! Error encountered during contract execution [**execution reverted**]
Value:	0.00000000000000466 Ether (< $0.000001) - [CANCELLED]
Transaction Fee:	215.881231529184168417 Ether ($371,313.56)
Gas Price:	0.000358358465198001 Ether (358,358.465198001 Gwei)
Ether Price:	$1,630.62 / ETH
Gas Limit & Usage by Txn:	1,232,758 | 602,417 (48.87%)
Gas Fees:	Base: 18.74448131 Gwei

This was followed by the exploiting transaction calling the process function with the following arguments

Txn Type: 0 (Legacy) | Nonce: 25 | Position: 3


```
[0]:  0000000000000000000000000000000000000000000000000000000000000020
[1]:  00000000000000000000000000000000000000000000000000000000000000d1
[2]:  6265616d000000000000000000000000d3dfd3ede74e0dcebc1aa685e1513328
[3]:  57efce2d000013d60065746800000000000000000000000088a69b4e698a4b09
[4]:  0df6cf5bd7b2d47325ad30a3006574680000000000000000000000002260fac5
[5]:  e5542a773aa44fbcfedf7c193bc2c59903000000000000000000000000b5c55f
[6]:  76f90cc528b2609109ca14d8d84593590e000000000000000000000000000000
[7]:  00000000000000000000000002540be400e6e85ded018819209cfb948d074cb6
[8]:  5de145734b5b0852e4a5db25cac2b8c39a000000000000000000000000000000
```


View Input As | Decode Input Data

The process function can be called by anyone, and is intended to process messages that have already been proved.

The process function

```
function process(bytes memory _message) public returns (bool _success) {
      // ensure message was meant for this domain
      bytes29 _m = _message.ref(0);
      require(_m.destination() == localDomain, "!destination");
      // ensure message has been proven
      bytes32 _messageHash = _m.keccak();
      require(acceptableRoot(messages[_messageHash]), "!proven");
      // check re-entrancy guard
      require(entered == 1, "!reentrant");
      entered = 0;
      // update message status as processed
      messages[_messageHash] = LEGACY_STATUS_PROCESSED;
      // call handle function
      IMessageRecipient(_m.recipientAddress()).handle(
          _m.origin(),
          _m.nonce(),
          _m.sender(),
          _m.body().clone()
      );
      // emit process results
      emit Process(_messageHash, true, "");
      // reset re-entrancy guard
      entered = 1;
      // return true
      return true;
  }
```

This line

`require(acceptableRoot(messages[_messageHash]), "!proven");`

checks the validity of the merkle root of of the messages

using this function

```
function acceptableRoot(bytes32 _root) public view returns (bool) {
 // this is backwards-compatibility for messages proven/processed
 // under previous versions
 if (_root == LEGACY_STATUS_PROVEN) return true;
 if (_root == LEGACY_STATUS_PROCESSED) return false;uint256 _time =
confirmAt[_root];
 if (_time == 0) {
 return false;
 }
 return block.timestamp >= _time;
 }
```

using the following constants declared earlier.

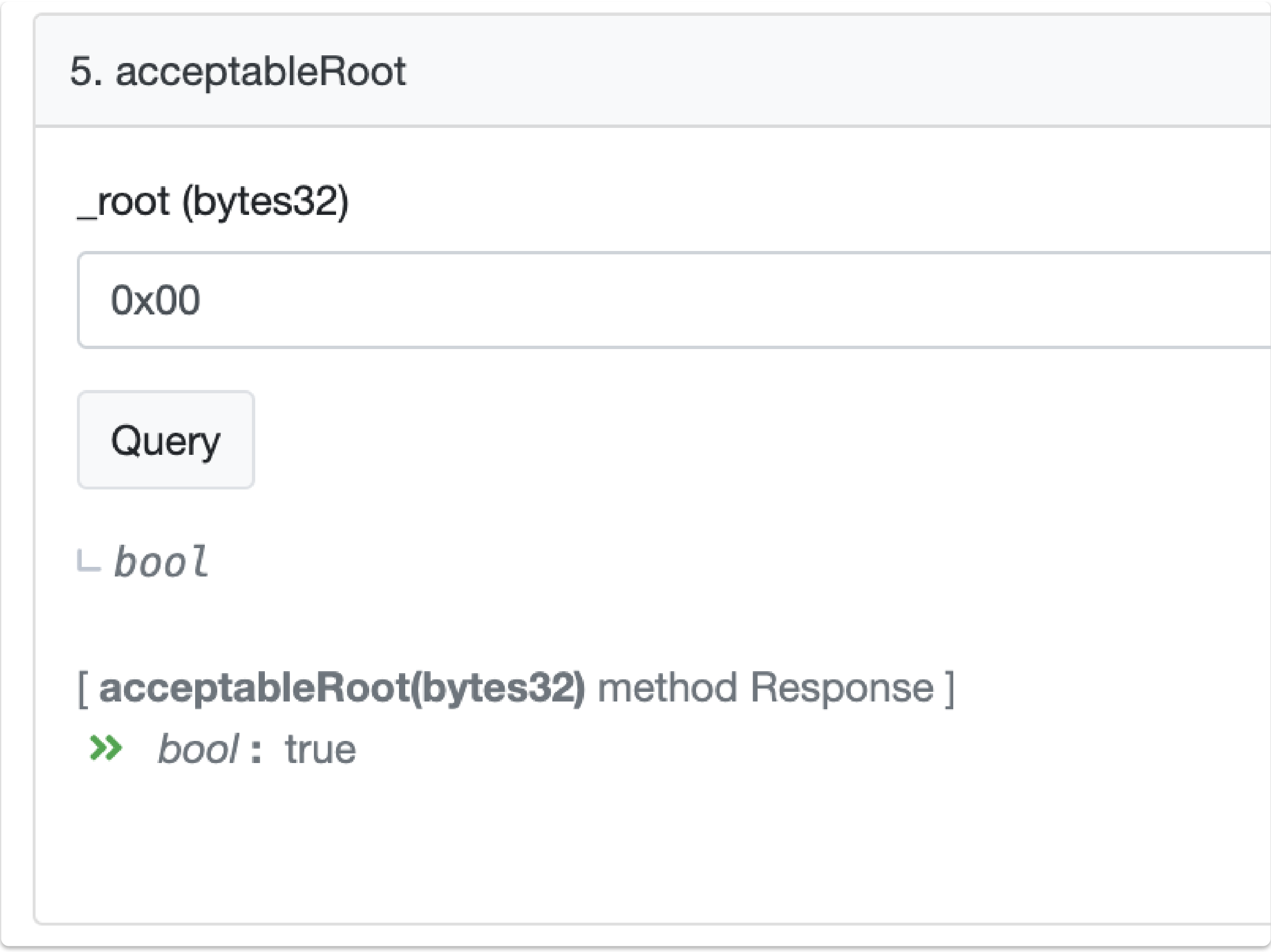
```
bytes32 public constant LEGACY_STATUS_NONE = bytes32(0);
bytes32 public constant LEGACY_STATUS_PROVEN = bytes32(uint256(1));
bytes32 public constant LEGACY_STATUS_PROCESSED = bytes32(uint256(2));
```

The line

`uint256 _time = confirmAt[_root];`

now returns 1 (remember in initialize, `confirmAt[0x00]` is set to 1)

You can check the return value of this function yourself on etherscan as I did :



Thus any message sent to process would succeed !

Note

The audit report from Quanstamp is available, their finding ‘QSP–19 Proving With An Empty Leaf’ is particularly relevant to this exploit, but perhaps not understood by the Nomad team.

MEV

See SoK : Front Running Attacks on Blockchain
Useful Overview - MEV Wiki

"Front-running is a course of action where someone benefits from early access to market information about upcoming transactions and trades"

Financial Systems that have transparency with respect to their transactions are somewhat unique.

From https://hackmd.io/@flashbots/quantifying-REV
Maximal (formerly Miner) Extractable Value is the value that can be extracted from a blockchain by any agent without special permissions. Considering this permissionless nature, any agent with transaction ordering rights will be in a privileged position to perform the extraction.

There are features of Ethereum (and other blockchains) that allow front running

1. All transactions are available in a public mempool before they are mined
2. All transaction data is public
3. Transactions can be cloned

Sometimes people also include things like liquidations or arbitrage within the MEV category even though strictly speaking they don't rely on Tx reordering. For this reason it's helpful to make a diction between transaction re-ordering/gas auctions, front running etc & other activity such as arbitrage and liquidations.

Introductory Video

https://www.youtube.com/watch?v=UZ-NNd6yjFM

GENERIC FRONT RUNNING BOTS

The above example shows how bots can extract value to a transaction irrespective of the contract called.

Good Vs Bad MEV

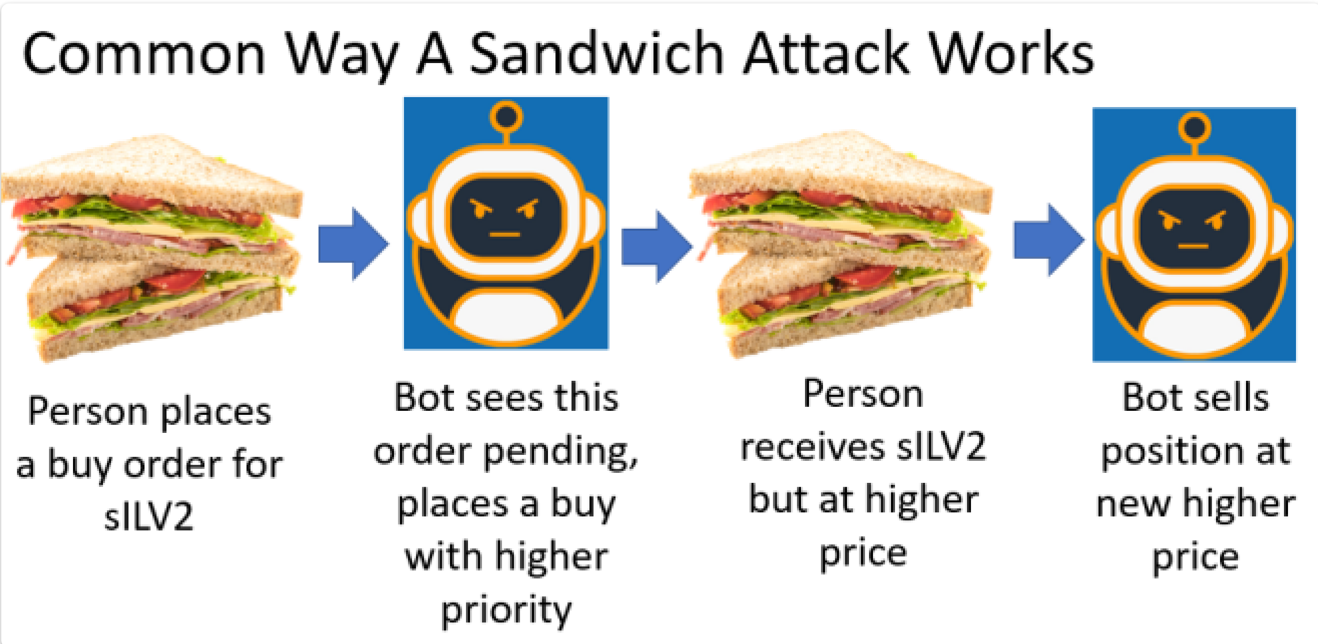
- Good MEV (not always good but certainly less harmful/existential): Arbitrage, liquidations etc
- Bad: Basically TX re-ordering. Front running, backing running, sandwich attacks etc.

Problems with MEV

- Centralisation of block sequencing
- Increase in gas prices (gas auctions)
- Negative User experience
- Breaking of concensus
- Increased cost to users

A Taxonomy of Front-running Attacks

- **Front Running**: Monitoring mempool for profitable trades/TXs then submitting them but with higher gas fees.
- **Back Running**: monitoring of a mempool to execute a transaction immediately after a pending target transaction.
- **Sandwich Attack**: Combination of front & back running to sandwich a trade.



Example

2021-08-19 12:53:18	sell	$2.6153992	0.00651959	379.05537	991.38112	2.4712861	0xbf3da4...eda2
2021-08-19 12:53:15	buy	$2.6812377	0.00668371	70.340944	188.60079	0.47013857	0xdc6b3e...9a83
2021-08-19 12:53:15	buy	$2.6044158	0.00649221	379.05537	987.2178	2.4609079	0xbf3da4...eda2

2021-08-18 21:03:02	sell	$2.8045737	0.00707065	1,659.0006	4,652.7896	11.730214	0xbf3da4...eda2
2021-08-18 21:03:02	buy	$3.1003609	0.00781636	255.38776	791.79423	1.9962038	0xe4c0f3...52b0
2021-08-18 21:03:02	buy	$2.7204888	0.00685866	1,659.0006	4,513.2925	11.378526	0xbf3da4...eda2

- **Time Bandit Attacks**: Time bandit attacks involve the re-mining of blocks in order to maximize profits. An opportunity for a time bandit attack might occur when block rewards are small enough compared to MEV. This incentivizes miners to destabilize the consensus to reap the maximum profits.

  For example, let's say the highest block of a blockchain network is numbered `#B10` and its block reward is $100. Incidentally, a miner notices an MEV opportunity worth $1000 on block number `#B7`. This opportunity might incentivize the said miner to remine the `#B7` block and all subsequent blocks to reap the MEV rewards while adhering to the longest-chain rule.

- **Uncle attacks**: When bundled transactions are mined into an uncle block, they're open for everyone to see. In this case, an attacker can select transactions from the bundle to front-run or back-run them. This also shows that attacks extend beyond the mempool and into uncled blocks as well.

Is MEV unique to Ethereum?

No, hypothetically MEV can also be seen on Bitcoin. The incentives to censor Lightning channels or to double-spend colored coins are technically MEV. Bitcoin is inherently less exposed to MEV than blockchains like Etheruem.

The reason for that lies in the complexity and "statefulness" of the respective blockchain:

1. The rate at which MEV accumulates on a given blockchain is generally proportional to the complexity of its application-layer behavior.
2. Arbitrarily flexible protocols, such as Ethereum, cannot bound this complexity and are inherently biased towards greater complexity over time.
3. MEV incentives cannot be easily mitigated without altering Ethereum's UX.

Ethereum's complexity may be a curse.

Who is doing this ?

From MEV and Me (Feb 2021)
The defining feature of Ethereum's current era is that most miners are not attempting to exploit MEV themselves (yet). Nearly all of the current activity is driven by non-mining traders. However, some MEV can only be captured by miners, because they have the authority to arbitrarily order (or exclude) transactions. Non-mining traders can access a strictly smaller subset of "simple" MEV; "complex" preferences cannot be efficiently expressed through PGA's.