Module 1: Flashbots Setup & Configuration

Course Information

- **Duration**: 90 minutes
- **Level**: Beginner-Intermediate
- **Author**: Obelisk Core
- **Course**: Enthusiast Path - Tool Usage Tutorials

Learning Objectives

By the end of this module, you will be able to:
- Understand the fundamentals of Flashbots and bundle submission
- Install and configure the Flashbots SDK
- Set up secure authentication for bundle submission
- Create and submit bundles for MEV extraction
- Handle different bundle types and use cases
- Implement error handling and monitoring
- Optimize bundle performance for better inclusion rates

Table of Contents

Introduction to Flashbots

Flashbots is a decentralized infrastructure that enables trustless transaction orderflow capture and prevents frontrunning of MEV strategies. It provides a direct connection to builders, allowing for private transaction submission and bundle ordering.

Key Benefits of Flashbots

Privacy & Security
- Transactions are submitted privately to builders
- No mempool exposure means no front-running risk
- Direct builder communication ensures priority

Predictable Inclusion
- Bundles have guaranteed ordering within the block
- First-price auction mechanism
- Clear pricing model based on priority fee

MEV Protection
- Strategies are hidden from other searchers
- Competitive advantage in timing-sensitive opportunities
- Protection against sandwich attacks

Understanding Bundle Structure

```
# Basic bundle structure
bundle = {
    'jsonrpc': '2.0',
    'id': 1,
    'method': 'eth_sendBundle',
    'params': [{
        'txs': ['0x...', '0x...'],  # Array of signed transactions
        'blockNumber': '0x1',       # Target block (hex)
        'minTimestamp': 1640995200, # Optional: minimum block time
        'maxTimestamp': 1640995260, # Optional: maximum block time
        'revertingTxHashes': ['0x...'] # Transactions that can revert
    }]
}
```

Installation & Setup

Prerequisites

Before installing Flashbots SDK, ensure you have:
- Python 3.8 or higher
- An Ethereum RPC endpoint (Infura, Alchemy, or local node)
- Private key for transaction signing
- Basic understanding of Ethereum transactions

Installing Flashbots SDK

```
# Create virtual environment
python -m venv flashbots_env
source flashbots_env/bin/activate  # On Windows:
flashbots_env\Scripts\activate

# Install Flashbots SDK
pip install flashbots

# Install additional dependencies
pip install web3 aiohttp eth-account

# Verify installation
python -c "import flashbots; print('Flashbots SDK installed
successfully')"
```

Environment Setup

Create a `.env` file for secure configuration:

```
# .env file
PRIVATE_KEY=your_private_key_here
ETH_RPC_URL=https://eth-mainnet.alchemyapi.io/v2/YOUR_API_KEY
FLASHBOTS_RELAY=https://relay.flashbots.net
LOG_LEVEL=INFO
```

Load environment variables:

```
import os
from dotenv import load_dotenv

load_dotenv()

# Configuration
PRIVATE_KEY = os.getenv('PRIVATE_KEY')
ETH_RPC_URL = os.getenv('ETH_RPC_URL')
FLASHBOTS_RELAY = os.getenv('FLASHBOTS_RELAY')
LOG_LEVEL = os.getenv('LOG_LEVEL', 'INFO')
```

Configuration & Authentication

Basic Flashbots Configuration

```
import flashbots
from flashbots import flashbot
from web3 import Web3
from eth_account import Account
import logging

# Configure logging
logging.basicConfig(level=getattr(logging, LOG_LEVEL))
logger = logging.getLogger(__name__)

class FlashbotsManager:
    def __init__(self, private_key: str, rpc_url: str, relay_url: str):
        self.w3 = Web3(Web3.HTTPProvider(rpc_url))
        self.account = Account.from_key(private_key)
        self.relay_url = relay_url

        # Initialize Flashbots provider
        self.flashbots_provider = flashbot.FlashbotsProvider(
            relay_url,
            private_key
        )

        logger.info(f"Initialized Flashbots for account:
{self.account.address}")
        logger.info(f"Connected to Flashbots:
{self.flashbots_provider.is_available()}")

    def get_account_info(self):
        """Get current account information"""
        balance = self.w3.eth.get_balance(self.account.address)
        nonce = self.w3.eth.get_transaction_count(self.account.address)

        return {
            'address': self.account.address,
            'balance_wei': balance,
            'balance_eth': self.w3.from_wei(balance, 'ether'),
            'nonce': nonce,
            'chain_id': self.w3.eth.chain_id
        }
```

Network Configuration

```
class NetworkConfig:
    """Network-specific configurations"""

    MAINNET = {
        'chain_id': 1,
        'name': 'Ethereum Mainnet',
        'relay_url': 'https://relay.flashbots.net',
        'rpc_url': 'https://eth-mainnet.alchemyapi.io/v2/',
        'explorer': 'https://etherscan.io',
        'gas_price_multiplier': 1.0
    }

    GOERLI = {
        'chain_id': 5,
        'name': 'Goerli Testnet',
        'relay_url': 'https://relay-goerli.flashbots.net',
        'rpc_url': 'https://eth-goerli.alchemyapi.io/v2/',
        'explorer': 'https://goerli.etherscan.io',
        'gas_price_multiplier': 0.1
    }

    @classmethod
    def get_config(cls, network: str):
        return getattr(cls, network.upper(), cls.MAINNET)

# Usage
config = NetworkConfig.get_config('mainnet')
flashbots_manager = FlashbotsManager(
    private_key=PRIVATE_KEY,
    rpc_url=ETH_RPC_URL,
    relay_url=config['relay_url']
)
```

Bundle Creation Fundamentals

Transaction Preparation

```
import time
from typing import List, Dict, Optional
from eth_account import Account
from web3.types import TxParams, HexBytes

class BundleBuilder:
    def __init__(self, flashbots_manager):
        self.manager = flashbots_manager
        self.w3 = flashbots_manager.w3
        self.account = flashbots_manager.account

    def create_arbitrage_transaction(self,
                                     contract_address: str,
                                     function_data: str,
                                     gas_limit: int = 200000,
                                     gas_price_multiplier: float = 1.1) -
> TxParams:
        """Create an arbitrage transaction"""

        # Get current gas price with multiplier
        base_gas_price = self.w3.eth.gas_price
        priority_fee = int(base_gas_price * (gas_price_multiplier - 1))
        max_fee = int(base_gas_price * gas_price_multiplier)

        # Create transaction parameters
        tx_params = {
            'chainId': self.w3.eth.chain_id,
            'nonce':
self.w3.eth.get_transaction_count(self.account.address),
            'to': contract_address,
            'data': function_data,
            'gas': gas_limit,
            'gasPrice': max_fee + priority_fee,  # EIP-1559
            'maxFeePerGas': max_fee,
            'maxPriorityFeePerGas': priority_fee,
            'value': 0,
            'type': 2  # EIP-1559 transaction type
        }

        return tx_params

    def sign_transaction(self, tx_params: TxParams) -> str:
```

```
        """Sign transaction with private key"""
        signed_tx = self.account.sign_transaction(tx_params)
        return signed_tx.rawTransaction.hex()
```

Bundle Creation and Submission

```
class BundleManager:
    def __init__(self, flashbots_manager):
        self.manager = flashbots_manager
        self.flashbots_provider = flashbots_manager.flashbots_provider

    def create_bundle(self,
                      transactions: List[str],
                      target_block: int,
                      min_timestamp: Optional[int] = None,
                      max_timestamp: Optional[int] = None,
                      reverting_hashes: Optional[List[str]] = None) ->
Dict:
        """Create a complete bundle for submission"""

        # Ensure transactions are properly formatted
        formatted_txs = []
        for tx in transactions:
            if isinstance(tx, str):
                formatted_txs.append(tx)
            elif hasattr(tx, 'rawTransaction'):
                formatted_txs.append(tx.rawTransaction.hex())
            else:
                raise ValueError("Invalid transaction format")

        # Create bundle parameters
        bundle_params = {
            'txs': formatted_txs,
            'blockNumber': hex(target_block),
        }

        # Add optional parameters
        if min_timestamp:
            bundle_params['minTimestamp'] = min_timestamp

        if max_timestamp:
            bundle_params['maxTimestamp'] = max_timestamp

        if reverting_hashes:
            bundle_params['revertingTxHashes'] = reverting_hashes

        return {
            'jsonrpc': '2.0',
```

```
            'id': int(time.time()),
            'method': 'eth_sendBundle',
            'params': [bundle_params]
        }

    async def submit_bundle(self, bundle: Dict) -> Dict:
        """Submit bundle to Flashbots relay"""

        try:
            # Submit bundle
            result = await self.flashbots_provider.send_bundle(bundle)

            # Log submission details
            logger.info(f"Bundle submitted: {result}")

            return {
                'success': True,
                'result': result,
                'bundle_hash': result.get('bundleHash'),
                'submitted_at': int(time.time())
            }

        except Exception as e:
            logger.error(f"Bundle submission failed: {e}")
            return {
                'success': False,
                'error': str(e),
                'submitted_at': int(time.time())
            }

    def estimate_bundle_success(self,
                                transactions: List[str],
                                target_block: int,
                                gas_price: int) -> Dict:
        """Estimate bundle inclusion probability"""

        # Get current block info
        current_block = self.w3.eth.block_number
        blocks_ahead = target_block - current_block

        # Simple heuristic for success probability
        if blocks_ahead <= 0:
```

```
            probability = 0.0
        elif blocks_ahead <= 2:
            probability = 0.8
        elif blocks_ahead <= 5:
            probability = 0.6
        else:
            probability = 0.3

        # Adjust based on gas price
        base_gas_price = self.w3.eth.gas_price
        gas_multiplier = gas_price / base_gas_price

        if gas_multiplier >= 2.0:
            probability += 0.1
        elif gas_multiplier >= 1.5:
            probability += 0.05

        probability = min(probability, 1.0)

        return {
            'current_block': current_block,
            'target_block': target_block,
            'blocks_ahead': blocks_ahead,
            'gas_price_multiplier': gas_multiplier,
            'estimated_success_probability': probability,
            'recommendation': 'submit' if probability > 0.5 else 'wait'
        }
```

Practical Bundle Submission

Arbitrage Bundle Example

```
class ArbitrageBundle:
    def __init__(self, flashbots_manager):
        self.manager = flashbots_manager
        self.w3 = flashbots_manager.w3
        self.account = flashbots_manager.account

    async def create_arbitrage_bundle(self,
                                      uniswap_router: str,
                                      sushiswap_router: str,
                                      token_in: str,
                                      token_out: str,
                                      amount_in: int,
                                      slippage_tolerance: float = 0.01) -
> Dict:
        """Create a complete arbitrage bundle"""

        try:
            # Calculate optimal trade sizes
            opportunity = await self.calculate_arbitrage_opportunity(
                uniswap_router, sushiswap_router, token_in, token_out,
amount_in
            )

            if not opportunity['profitable']:
                return {
                    'success': False,
                    'reason': 'No profitable opportunity found'
                }

            # Create buy transaction (lower priced exchange)
            buy_tx = await self.create_swap_transaction(
                exchange_address=opportunity['buy_exchange'],
                token_in=token_in,
                token_out=token_out,
                amount_in=amount_in,
                slippage=slippage_tolerance,
                min_amount_out=int(amount_in *
opportunity['buy_price'])
            )

            # Create sell transaction (higher priced exchange)
            sell_tx = await self.create_swap_transaction(
```

```
                exchange_address=opportunity['sell_exchange'],
                token_in=token_out,
                token_out=token_in,
                amount_in=int(amount_in * opportunity['buy_price']),
                slippage=slippage_tolerance,
                min_amount_out=int(amount_in * (1 +
opportunity['profit_margin']))
            )

            # Sign transactions
            signed_buy_tx =
self.account.sign_transaction(buy_tx).rawTransaction.hex()
            signed_sell_tx =
self.account.sign_transaction(sell_tx).rawTransaction.hex()

            # Create bundle
            bundle_manager = BundleManager(self.manager)
            target_block = self.w3.eth.block_number + 1

            bundle = bundle_manager.create_bundle(
                transactions=[signed_buy_tx, signed_sell_tx],
                target_block=target_block,
                reverting_hashes=[]  # Allow reverting if no profit
            )

            return {
                'success': True,
                'bundle': bundle,
                'opportunity': opportunity,
                'estimated_profit': opportunity['estimated_profit'],
                'gas_cost_estimate': self.estimate_gas_cost([buy_tx,
sell_tx])
            }

        except Exception as e:
            logger.error(f"Failed to create arbitrage bundle: {e}")
            return {
                'success': False,
                'error': str(e)
            }

    async def calculate_arbitrage_opportunity(self,
```

```
                                            uniswap_router: str,
                                            sushiswap_router: str,
                                            token_in: str,
                                            token_out: str,
                                            amount_in: int) -> Dict:
        """Calculate if arbitrage opportunity exists"""

        # This is a simplified calculation - in practice you'd need
real price feeds
        # For demo purposes, assume price difference
        uniswap_price = 2350.50  # ETH/USDC
        sushiswap_price = 2352.75

        buy_price = min(uniswap_price, sushiswap_price)
        sell_price = max(uniswap_price, sushiswap_price)

        profit_margin = (sell_price - buy_price) / buy_price
        gas_cost_estimate = 200000 * 20 * 30e-9  # Rough gas cost
estimate

        # Calculate trade size for optimal profit
        optimal_trade_size = int(1000 * 1e18)  # 1000 tokens

        estimated_profit = (optimal_trade_size * (sell_price -
buy_price) / sell_price) - gas_cost_estimate

        return {
            'profitable': estimated_profit > 0,
            'buy_exchange': uniswap_router if uniswap_price <
sushiswap_price else sushiswap_router,
            'sell_exchange': sushiswap_router if uniswap_price <
sushiswap_price else uniswap_router,
            'buy_price': buy_price,
            'sell_price': sell_price,
            'profit_margin': profit_margin,
            'optimal_trade_size': optimal_trade_size,
            'estimated_profit': estimated_profit,
            'gas_cost_estimate': gas_cost_estimate
        }

    async def create_swap_transaction(self,
                                      exchange_address: str,
```

```
                                    token_in: str,
                                    token_out: str,
                                    amount_in: int,
                                    slippage: float,
                                    min_amount_out: int) -> TxParams:
        """Create a swap transaction"""

        # Simplified swap data - in practice use proper ABI encoding
        swap_data = f"0x38ed1739000000000000000000000000{token_in[2:]}
000000000000000000000000{token_out[2:]}
0000000000000000000000000000000000000000000000000000000000000000{min_amount_out}
0000000000000000000000000000000000000000000000000000000000000064000000000000000000

        tx_params = {
            'chainId': self.w3.eth.chain_id,
            'nonce':
self.w3.eth.get_transaction_count(self.account.address),
            'to': exchange_address,
            'data': swap_data,
            'gas': 300000,
            'maxFeePerGas': self.w3.eth.gas_price * 2,
            'maxPriorityFeePerGas': self.w3.eth.gas_price,
            'value': 0
        }

        return tx_params
```

Bundle Monitoring

```
class BundleMonitor:
    def __init__(self, flashbots_manager):
        self.manager = flashbots_manager
        self.w3 = flashbots_manager.w3
        self.submitted_bundles = {}

    async def monitor_bundle_submission(self,
                                        bundle_hash: str,
                                        target_block: int,
                                        max_wait_blocks: int = 10) ->
Dict:
        """Monitor bundle inclusion status"""

        start_block = self.w3.eth.block_number
        end_block = start_block + max_wait_blocks

        for current_block in range(start_block, end_block + 1):
            try:
                # Check if bundle was included in this block
                block = self.w3.eth.get_block(current_block,
full_transactions=True)

                # Check for our transactions in the block
                included_transactions = []
                for tx in block.transactions:
                    # Compare transaction hashes with our bundle
                    # This is simplified - in practice you'd track
specific transaction data
                    pass

                if included_transactions:
                    return {
                        'status': 'included',
                        'included_block': current_block,
                        'included_transactions': included_transactions,
                        'blocks_to_include': current_block -
start_block,
                        'gas_price_paid':
included_transactions[0].get('effectiveGasPrice', 0)
                    }

                # Small delay between block checks
```

```
                await asyncio.sleep(1)

            except Exception as e:
                logger.error(f"Error monitoring block {current_block}:
{e}")
                continue

        return {
            'status': 'not_included',
            'monitored_blocks': max_wait_blocks,
            'reason': 'Bundle not found in target blocks'
        }

    async def get_bundle_status(self, bundle_hash: str) -> Dict:
        """Get detailed status of a submitted bundle"""

        try:
            # Query Flashbots API for bundle status
            # Note: This is a placeholder - actual API implementation
would depend on Flashbots endpoints

            return {
                'bundle_hash': bundle_hash,
                'status': 'pending',  # pending, included,
not_included, reverted
                'submission_block': self.w3.eth.block_number,
                'estimated_inclusion': 'unknown',
                'gas_price_submitted': 'unknown',
                'revenue_generated': 0,
                'errors': []
            }

        except Exception as e:
            logger.error(f"Failed to get bundle status: {e}")
            return {
                'bundle_hash': bundle_hash,
                'status': 'error',
                'error': str(e)
            }
```

Advanced Flashbots Features

Private Mempool Integration

```
class PrivateMempoolManager:
    def __init__(self, flashbots_manager):
        self.manager = flashbots_manager
        self.w3 = flashbots_manager.w3
        self.private_connections = {}

    async def setup_private_mempool(self,
                                    endpoint_url: str,
                                    builder_name: str) -> bool:
        """Set up private mempool connection"""

        try:
            # Initialize private connection
            private_session = aiohttp.ClientSession()
            self.private_connections[builder_name] = {
                'session': private_session,
                'endpoint': endpoint_url,
                'builder': builder_name,
                'status': 'active'
            }

            logger.info(f"Private mempool connected to {builder_name}")
            return True

        except Exception as e:
            logger.error(f"Failed to connect to private mempool
{builder_name}: {e}")
            return False

    async def submit_to_private_mempool(self,
                                        builder_name: str,
                                        bundle: Dict) -> Dict:
        """Submit bundle to private mempool"""

        if builder_name not in self.private_connections:
            return {'success': False, 'error': 'Builder not connected'}

        try:
            private_connection = self.private_connections[builder_name]
            session = private_connection['session']

            # Submit to private mempool endpoint
```

```
            async with session.post(
                f"{private_connection['endpoint']}/bundle",
                json=bundle,
                headers={'Content-Type': 'application/json'}
            ) as response:

                result = await response.json()

                return {
                    'success': response.status == 200,
                    'builder': builder_name,
                    'response': result,
                    'submission_time': int(time.time())
                }

        except Exception as e:
            logger.error(f"Private mempool submission failed: {e}")
            return {
                'success': False,
                'builder': builder_name,
                'error': str(e)
            }

    async def broadcast_to_multiple_builders(self,
                                             bundle: Dict,
                                             builders: List[str]) ->
Dict:
        """Submit bundle to multiple builders simultaneously"""

        tasks = []
        for builder in builders:
            if builder in self.private_connections:
                task = self.submit_to_private_mempool(builder, bundle)
                tasks.append(task)

        # Submit to all builders concurrently
        results = await asyncio.gather(*tasks, return_exceptions=True)

        successful_submissions = []
        failed_submissions = []

        for i, result in enumerate(results):
```

```
            builder = builders[i]
            if isinstance(result, Exception):
                failed_submissions.append({
                    'builder': builder,
                    'error': str(result)
                })
            elif result.get('success'):
                successful_submissions.append({
                    'builder': builder,
                    'response': result
                })
            else:
                failed_submissions.append({
                    'builder': builder,
                    'error': result.get('error', 'Unknown error')
                })

        return {
            'total_submissions': len(builders),
            'successful': len(successful_submissions),
            'failed': len(failed_submissions),
            'successful_submissions': successful_submissions,
            'failed_submissions': failed_submissions
        }
```

Builder Selection Strategy

```
class BuilderStrategy:
    def __init__(self):
        self.builder_performance = {}
        self.fallback_builders = []
        self.preferred_builders = []

    def update_builder_performance(self,
                                   builder_name: str,
                                   inclusion_success: bool,
                                   block_inclusion_time: int,
                                   revenue_generated: float) -> None:
        """Update builder performance metrics"""

        if builder_name not in self.builder_performance:
            self.builder_performance[builder_name] = {
                'total_submissions': 0,
                'successful_inclusions': 0,
                'total_revenue': 0.0,
                'average_inclusion_time': 0,
                'success_rate': 0.0,
                'revenue_per_bundle': 0.0
            }

        stats = self.builder_performance[builder_name]
        stats['total_submissions'] += 1

        if inclusion_success:
            stats['successful_inclusions'] += 1
            stats['total_revenue'] += revenue_generated
            stats['average_inclusion_time'] = (
                (stats['average_inclusion_time'] *
(stats['total_submissions'] - 1) + block_inclusion_time)
                / stats['total_submissions']
            )

        stats['success_rate'] = stats['successful_inclusions'] /
stats['total_submissions']
        stats['revenue_per_bundle'] = stats['total_revenue'] /
stats['total_submissions']

    def get_optimal_builder(self,
                            strategy_type: str = 'arbitrage',
```

```
                          min_success_rate: float = 0.7) -> str:
        """Get optimal builder based on strategy and performance"""

        valid_builders = []

        for builder, stats in self.builder_performance.items():
            if stats['success_rate'] >= min_success_rate:
                valid_builders.append({
                    'builder': builder,
                    'success_rate': stats['success_rate'],
                    'revenue_per_bundle': stats['revenue_per_bundle'],
                    'average_inclusion_time':
stats['average_inclusion_time']
                })

        if not valid_builders:
            # Fallback to Flashbots if no qualified builders
            return 'flashbots'

        # Sort by revenue per bundle (primary) and success rate
(secondary)
        valid_builders.sort(
            key=lambda x: (x['revenue_per_bundle'], x['success_rate']),
            reverse=True
        )

        return valid_builders[0]['builder']

    def create_builder_portfolio(self,
                                 strategy_type: str = 'arbitrage',
                                 num_builders: int = 3) -> List[str]:
        """Create a portfolio of builders for risk diversification"""

        # Get all qualified builders
        qualified_builders = []
        for builder, stats in self.builder_performance.items():
            if stats['success_rate'] >= 0.5:
                qualified_builders.append({
                    'builder': builder,
                    'score': stats['success_rate'] *
stats['revenue_per_bundle']
                })
```

```
        # Sort by score and select top performers
        qualified_builders.sort(key=lambda x: x['score'], reverse=True)

        # Ensure Flashbots is always included
        portfolio = ['flashbots']
        for builder_data in qualified_builders[:num_builders-1]:
            portfolio.append(builder_data['builder'])

        return portfolio
```

Monitoring & Debugging

Comprehensive Bundle Tracker

```
import json
import sqlite3
from datetime import datetime, timedelta
from typing import Dict, List, Optional

class BundleTracker:
    def __init__(self, db_path: str = "bundle_tracking.db"):
        self.db_path = db_path
        self.init_database()
        self.active_tracking = {}

    def init_database(self):
        """Initialize SQLite database for bundle tracking"""

        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()

        cursor.execute('''
            CREATE TABLE IF NOT EXISTS bundle_submissions (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                bundle_hash TEXT UNIQUE,
                strategy_type TEXT,
                block_target INTEGER,
                gas_price INTEGER,
                transactions_count INTEGER,
                estimated_revenue REAL,
                status TEXT,
                submitted_at TIMESTAMP,
                included_at TIMESTAMP,
                revenue REAL,
                gas_used INTEGER,
                error_message TEXT
            )
        ''')

        cursor.execute('''
            CREATE TABLE IF NOT EXISTS builder_performance (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                builder_name TEXT,
                strategy_type TEXT,
                submission_date DATE,
                total_submissions INTEGER,
```

```
                successful_inclusions INTEGER,
                total_revenue REAL,
                average_gas_price INTEGER,
                success_rate REAL,
                created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
            )
        ''')

        conn.commit()
        conn.close()

    def track_bundle_submission(self,
                                bundle_hash: str,
                                strategy_type: str,
                                target_block: int,
                                gas_price: int,
                                transaction_count: int,
                                estimated_revenue: float) -> None:
        """Track a new bundle submission"""

        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()

        cursor.execute('''
            INSERT OR REPLACE INTO bundle_submissions
            (bundle_hash, strategy_type, block_target, gas_price,
transactions_count,
             estimated_revenue, status, submitted_at)
            VALUES (?, ?, ?, ?, ?, ?, ?, ?)
        ''', (
            bundle_hash, strategy_type, target_block, gas_price,
transaction_count,
            estimated_revenue, 'submitted', datetime.now()
        ))

        conn.commit()
        conn.close()

        # Add to active tracking
        self.active_tracking[bundle_hash] = {
            'strategy_type': strategy_type,
            'target_block': target_block,
```

```
            'submitted_at': datetime.now()
        }

    def update_bundle_status(self,
                             bundle_hash: str,
                             status: str,
                             included_block: Optional[int] = None,
                             revenue: Optional[float] = None,
                             gas_used: Optional[int] = None,
                             error_message: Optional[str] = None) ->
None:
        """Update bundle status in database"""

        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()

        update_fields = ['status = ?']
        update_values = [status]

        if included_block:
            update_fields.append('included_at = ?')
            update_values.append(datetime.fromtimestamp(included_block
* 15))  # Approximate block time

        if revenue is not None:
            update_fields.append('revenue = ?')
            update_values.append(revenue)

        if gas_used is not None:
            update_fields.append('gas_used = ?')
            update_values.append(gas_used)

        if error_message:
            update_fields.append('error_message = ?')
            update_values.append(error_message)

        update_values.append(bundle_hash)

        cursor.execute(f'''
            UPDATE bundle_submissions
            SET {', '.join(update_fields)}
            WHERE bundle_hash = ?
```

```
        ''', update_values)

        conn.commit()
        conn.close()

    def get_bundle_analytics(self,
                             days_back: int = 30,
                             strategy_type: Optional[str] = None) ->
Dict:
        """Get comprehensive bundle analytics"""

        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()

        # Date filter
        date_filter = f"submitted_at >= datetime('now', '-{days_back}
days')"

        if strategy_type:
            date_filter += f" AND strategy_type = '{strategy_type}'"

        # Overall statistics
        cursor.execute(f'''
            SELECT
                COUNT(*) as total_bundles,
                SUM(CASE WHEN status = 'included' THEN 1 ELSE 0 END) as
successful_bundles,
                SUM(CASE WHEN status = 'included' THEN revenue ELSE 0
END) as total_revenue,
                AVG(CASE WHEN status = 'included' THEN revenue ELSE
NULL END) as avg_revenue,
                AVG(CASE WHEN status = 'included' THEN gas_used ELSE
NULL END) as avg_gas_used,
                AVG(gas_price) as avg_gas_price
            FROM bundle_submissions
            WHERE {date_filter}
        ''')

        overall_stats = dict(zip([desc[0] for desc in
cursor.description], cursor.fetchone()))

        # Success rate calculation
```

```
        success_rate = (overall_stats['successful_bundles'] /
overall_stats['total_bundles']) if overall_stats['total_bundles'] > 0
else 0
        overall_stats['success_rate'] = success_rate

        # Daily breakdown
        cursor.execute(f'''
            SELECT
                DATE(submitted_at) as submission_date,
                COUNT(*) as daily_bundles,
                SUM(CASE WHEN status = 'included' THEN 1 ELSE 0 END) as
daily_successful,
                SUM(CASE WHEN status = 'included' THEN revenue ELSE 0
END) as daily_revenue
            FROM bundle_submissions
            WHERE {date_filter}
            GROUP BY DATE(submitted_at)
            ORDER BY submission_date DESC
        ''')

        daily_breakdown = [dict(zip([desc[0] for desc in
cursor.description], row))
                           for row in cursor.fetchall()]

        # Strategy breakdown
        cursor.execute(f'''
            SELECT
                strategy_type,
                COUNT(*) as strategy_bundles,
                SUM(CASE WHEN status = 'included' THEN 1 ELSE 0 END) as
strategy_successful,
                SUM(CASE WHEN status = 'included' THEN revenue ELSE 0
END) as strategy_revenue
            FROM bundle_submissions
            WHERE {date_filter}
            GROUP BY strategy_type
        ''')

        strategy_breakdown = [dict(zip([desc[0] for desc in
cursor.description], row))
                              for row in cursor.fetchall()]
```

```
        conn.close()

        return {
            'overall_statistics': overall_stats,
            'daily_breakdown': daily_breakdown,
            'strategy_breakdown': strategy_breakdown,
            'analytics_period_days': days_back
        }
```

Error Handling and Recovery

```
class FlashbotsErrorHandler:
    def __init__(self, flashbots_manager):
        self.manager = flashbots_manager
        self.error_counts = {}
        self.retry_strategies = {}

    def handle_bundle_submission_error(self,
                                       error: Exception,
                                       bundle_data: Dict,
                                       retry_count: int = 0) -> Dict:
        """Handle and categorize bundle submission errors"""

        error_type = type(error).__name__
        error_message = str(error)

        # Update error tracking
        if error_type not in self.error_counts:
            self.error_counts[error_type] = 0
        self.error_counts[error_type] += 1

        logger.error(f"Bundle submission error: {error_type} -
{error_message}")

        # Categorize error and determine recovery strategy
        if 'rate_limit' in error_message.lower() or 'too many requests'
in error_message.lower():
            return self.handle_rate_limit_error(bundle_data,
retry_count)

        elif 'insufficient_funds' in error_message.lower():
            return self.handle_insufficient_funds_error(bundle_data)

        elif 'gas_price' in error_message.lower() or 'max_priority_fee'
in error_message.lower():
            return self.handle_gas_price_error(bundle_data,
retry_count)

        elif 'bundle' in error_message.lower() and 'invalid' in
error_message.lower():
            return self.handle_invalid_bundle_error(bundle_data)

        else:
```

```
            return self.handle_generic_error(bundle_data, error,
retry_count)

    def handle_rate_limit_error(self,
                                bundle_data: Dict,
                                retry_count: int) -> Dict:
        """Handle rate limit errors with exponential backoff"""

        max_retries = 3
        if retry_count >= max_retries:
            return {
                'action': 'fail',
                'reason': 'Max rate limit retries exceeded',
                'retry_count': retry_count
            }

        # Exponential backoff: 2^retry_count seconds
        backoff_seconds = min(2 ** retry_count, 60)  # Cap at 60
seconds

        return {
            'action': 'retry',
            'reason': 'Rate limit exceeded',
            'retry_count': retry_count + 1,
            'backoff_seconds': backoff_seconds,
            'suggested_retry_time': time.time() + backoff_seconds
        }

    def handle_gas_price_error(self,
                               bundle_data: Dict,
                               retry_count: int) -> Dict:
        """Handle gas price related errors"""

        # Increase gas price by 20% for next attempt
        current_gas_price = bundle_data.get('gas_price', 0)
        new_gas_price = int(current_gas_price * 1.2) if
current_gas_price > 0 else None

        return {
            'action': 'retry_with_modified_params',
            'reason': 'Gas price too low',
            'gas_price_multiplier': 1.2,
```

```
            'new_gas_price': new_gas_price,
            'retry_count': retry_count + 1
        }

    def handle_invalid_bundle_error(self, bundle_data: Dict) -> Dict:
        """Handle invalid bundle format errors"""

        return {
            'action': 'fail',
            'reason': 'Invalid bundle format - requires manual review',
            'bundle_data': bundle_data
        }

    def create_error_recovery_bundle(self,
                                     original_bundle: Dict,
                                     recovery_params: Dict) -> Dict:
        """Create a modified bundle for error recovery"""

        if 'gas_price_multiplier' in recovery_params:
            # Modify gas prices in all transactions
            multiplier = recovery_params['gas_price_multiplier']
            for tx_params in original_bundle.get('transactions', []):
                if 'gasPrice' in tx_params:
                    tx_params['gasPrice'] = int(tx_params['gasPrice'] *
multiplier)
                if 'maxFeePerGas' in tx_params:
                    tx_params['maxFeePerGas'] =
int(tx_params['maxFeePerGas'] * multiplier)

        if 'new_gas_price' in recovery_params:
            new_price = recovery_params['new_gas_price']
            for tx_params in original_bundle.get('transactions', []):
                if 'gasPrice' in tx_params:
                    tx_params['gasPrice'] = new_price
                if 'maxFeePerGas' in tx_params:
                    tx_params['maxFeePerGas'] = new_price

        return original_bundle
```

Performance Optimization

Gas Price Optimization

```
class GasOptimizer:
    def __init__(self, flashbots_manager):
        self.manager = flashbots_manager
        self.w3 = flashbots_manager.w3
        self.price_history = []
        self.optimization_cache = {}

    async def get_optimal_gas_price(self,
                                    strategy_type: str = 'arbitrage',
                                    urgency: str = 'normal') -> Dict:
        """Calculate optimal gas price for strategy"""

        current_block = self.w3.eth.block_number

        # Get base gas price from network
        base_gas_price = self.w3.eth.gas_price

        # Strategy-specific multipliers
        strategy_multipliers = {
            'arbitrage': 1.8,      # High urgency for arbitrage
            'liquidation': 2.0,    # Highest priority for liquidations
            'sandwich': 1.5,       # Medium priority
            'general': 1.3         # Low priority
        }

        # Urgency modifiers
        urgency_modifiers = {
            'critical': 2.5,       # Very urgent
            'high': 2.0,           # High urgency
            'normal': 1.5,         # Normal urgency
            'low': 1.2             # Low urgency
        }

        base_multiplier = strategy_multipliers.get(strategy_type, 1.3)
        urgency_multiplier = urgency_modifiers.get(urgency, 1.5)

        # Calculate network congestion factor
        congestion_factor = await self.calculate_network_congestion()

        # Final gas price calculation
        final_multiplier = base_multiplier * urgency_multiplier *
congestion_factor
```

```
        # EIP-1559 gas pricing
        base_fee = await self.get_base_fee()
        priority_fee = int(base_gas_price * (final_multiplier - 1))
        max_fee = base_fee + priority_fee + base_fee  # Add buffer

        return {
            'gas_price': max_fee + priority_fee,
            'max_fee_per_gas': max_fee,
            'max_priority_fee_per_gas': priority_fee,
            'base_fee': base_fee,
            'multiplier_used': final_multiplier,
            'congestion_factor': congestion_factor,
            'strategy_type': strategy_type,
            'urgency': urgency
        }

    async def calculate_network_congestion(self) -> float:
        """Calculate current network congestion factor"""

        try:
            # Get recent block data
            current_block = self.w3.eth.block_number

            # Sample last 20 blocks for congestion analysis
            recent_blocks = []
            for block_num in range(current_block - 19, current_block +
1):
                try:
                    block = self.w3.eth.get_block(block_num)
                    recent_blocks.append({
                        'gas_used': block.gasUsed,
                        'gas_limit': block.gasLimit,
                        'transaction_count': len(block.transactions)
                    })
                except:
                    continue

            if not recent_blocks:
                return 1.0  # Default congestion factor

            # Calculate congestion metrics
```

```
            avg_gas_used = sum(block['gas_used'] for block in
recent_blocks) / len(recent_blocks)
            avg_gas_limit = sum(block['gas_limit'] for block in
recent_blocks) / len(recent_blocks)
            gas_usage_ratio = avg_gas_used / avg_gas_limit

            # Transaction density
            avg_tx_count = sum(block['transaction_count'] for block in
recent_blocks) / len(recent_blocks)

            # Congestion factor calculation
            congestion_factor = 1.0

            # High gas usage increases congestion
            if gas_usage_ratio > 0.8:
                congestion_factor += 0.5
            elif gas_usage_ratio > 0.6:
                congestion_factor += 0.3
            elif gas_usage_ratio > 0.4:
                congestion_factor += 0.1

            # High transaction count increases congestion
            if avg_tx_count > 150:
                congestion_factor += 0.3
            elif avg_tx_count > 100:
                congestion_factor += 0.2
            elif avg_tx_count > 50:
                congestion_factor += 0.1

            return min(congestion_factor, 2.0)  # Cap congestion factor

        except Exception as e:
            logger.error(f"Error calculating network congestion: {e}")
            return 1.0  # Default to normal congestion

    async def get_base_fee(self) -> int:
        """Get current base fee per gas (EIP-1559)"""

        try:
            current_block = self.w3.eth.get_block('latest')
            return current_block.baseFeePerGas
        except:
```

```
            # Fallback for older clients without EIP-1559
            return self.w3.eth.gas_price

    def optimize_bundle_for_inclusion(self,
                                      bundle: Dict,
                                      target_success_rate: float = 0.8) -
> Dict:
        """Optimize bundle parameters for better inclusion
probability"""

        current_block = self.w3.eth.block_number
        target_block = int(bundle.get('block_number', current_block +
1), 16)

        blocks_ahead = target_block - current_block

        # Adjust gas price based on inclusion probability
        gas_multiplier = 1.0

        if blocks_ahead <= 1:
            gas_multiplier = 2.0  # Very high for next block
        elif blocks_ahead <= 2:
            gas_multiplier = 1.8
        elif blocks_ahead <= 5:
            gas_multiplier = 1.5
        elif blocks_ahead <= 10:
            gas_multiplier = 1.3
        else:
            gas_multiplier = 1.2

        # Apply optimization to transactions
        optimized_bundle = bundle.copy()

        for tx in optimized_bundle.get('transactions', []):
            if 'gasPrice' in tx:
                tx['gasPrice'] = int(tx['gasPrice'] * gas_multiplier)
            if 'maxFeePerGas' in tx:
                tx['maxFeePerGas'] = int(tx['maxFeePerGas'] *
gas_multiplier)

        return {
            'optimized_bundle': optimized_bundle,
```

```
            'gas_multiplier_applied': gas_multiplier,
            'blocks_ahead': blocks_ahead,
            'estimated_success_rate': min(0.9, 0.3 + blocks_ahead *
0.1)
        }
```

Bundle Timing Optimization

```
class TimingOptimizer:
    def __init__(self, flashbots_manager):
        self.manager = flashbots_manager
        self.w3 = flashbots_manager.w3
        self.block_timing_history = []

    async def get_optimal_block_timing(self,
                                       strategy_type: str = 'arbitrage',
                                       execution_time_estimate: int = 30)
-> Dict:
        """Determine optimal block timing for strategy"""

        current_block = self.w3.eth.block_number
        current_block_info = self.w3.eth.get_block(current_block)

        # Strategy-specific timing preferences
        strategy_timing = {
            'arbitrage': {
                'preferred_blocks_ahead': 1,
                'max_blocks_ahead': 2,
                'timing_sensitivity': 'high'
            },
            'liquidation': {
                'preferred_blocks_ahead': 1,
                'max_blocks_ahead': 1,
                'timing_sensitivity': 'critical'
            },
            'sandwich': {
                'preferred_blocks_ahead': 2,
                'max_blocks_ahead': 3,
                'timing_sensitivity': 'medium'
            }
        }

        timing_config = strategy_timing.get(strategy_type,
strategy_timing['arbitrage'])

        # Calculate optimal block
        if timing_config['timing_sensitivity'] == 'critical':
            # Liquidations must be in next block
            target_block = current_block + 1
        elif timing_config['timing_sensitivity'] == 'high':
```

```
            # Arbitrage prefers next block but can wait one more
            target_block = current_block + 1
        else:
            # General strategies can wait longer
            target_block = current_block + 2

        # Validate timing feasibility
        time_per_block = 12  # Average block time in seconds
        total_time_needed = timing_config['preferred_blocks_ahead'] *
time_per_block

        if total_time_needed < execution_time_estimate:
            # Need more time, delay block target
            additional_blocks = int((execution_time_estimate -
total_time_needed) / time_per_block) + 1
            target_block += additional_blocks

        # Check if target block is too far
        if target_block > current_block +
timing_config['max_blocks_ahead']:
            target_block = current_block +
timing_config['max_blocks_ahead']

        return {
            'current_block': current_block,
            'target_block': target_block,
            'blocks_ahead': target_block - current_block,
            'time_until_target': (target_block - current_block) *
time_per_block,
            'execution_time_estimate': execution_time_estimate,
            'timing_feasible': execution_time_estimate <= (target_block
- current_block) * time_per_block,
            'strategy_timing_config': timing_config
        }

    async def monitor_block_timing(self,
                                   target_block: int,
                                   monitoring_duration: int = 60) ->
Dict:
        """Monitor block timing for bundle submission"""

        start_block = self.w3.eth.block_number
```

```
        start_time = time.time()
        monitoring_end = start_time + monitoring_duration

        block_times = []
        submission_ready = False

        while time.time() < monitoring_end:
            current_block = self.w3.eth.block_number

            if current_block >= target_block:
                submission_ready = True
                time_to_block = time.time() - start_time
                break

            # Record block timing
            if len(block_times) > 0:
                time_since_last = time.time() - block_times[-1]
['timestamp']
                block_times.append({
                    'block': current_block,
                    'timestamp': time.time(),
                    'time_since_last': time_since_last
                })

            # Short sleep to avoid excessive API calls
            await asyncio.sleep(0.5)

        return {
            'submission_ready': submission_ready,
            'target_block': target_block,
            'current_block': current_block,
            'blocks_monitored': len(block_times),
            'average_block_time': sum(bt['time_since_last'] for bt in
block_times) / len(block_times) if block_times else 0,
            'time_to_submission': time.time() - start_time if
submission_ready else None
        }
```

Hands-on Exercise

Complete Flashbots Implementation

```
class CompleteFlashbotsStrategy:
    """Complete implementation of a MEV strategy using Flashbots"""

    def __init__(self, config: Dict):
        self.config = config
        self.flashbots_manager = FlashbotsManager(
            config['private_key'],
            config['rpc_url'],
            config['relay_url']
        )
        self.bundle_tracker = BundleTracker()
        self.gas_optimizer = GasOptimizer(self.flashbots_manager)
        self.timing_optimizer = TimingOptimizer(self.flashbots_manager)
        self.error_handler =
FlashbotsErrorHandler(self.flashbots_manager)

    async def run_arbitrage_strategy(self) -> Dict:
        """Execute complete arbitrage strategy using Flashbots"""

        logger.info("Starting arbitrage strategy execution")

        try:
            # Step 1: Opportunity Detection
            opportunity = await self.detect_arbitrage_opportunity()

            if not opportunity['profitable']:
                return {
                    'success': False,
                    'reason': 'No profitable arbitrage opportunity
found',
                    'opportunity': opportunity
                }

            # Step 2: Bundle Creation
            bundle_builder = BundleBuilder(self.flashbots_manager)

            # Create transactions
            buy_tx = bundle_builder.create_arbitrage_transaction(
                contract_address=opportunity['contract_address'],
                function_data=opportunity['swap_data'],
                gas_limit=opportunity['gas_estimate']
            )
```

```
            # Optimize gas prices
            gas_config = await
self.gas_optimizer.get_optimal_gas_price(
                strategy_type='arbitrage',
                urgency='high'
            )

            buy_tx['maxFeePerGas'] = gas_config['max_fee_per_gas']
            buy_tx['maxPriorityFeePerGas'] =
gas_config['max_priority_fee_per_gas']

            # Step 3: Bundle Assembly
            signed_buy_tx =
self.flashbots_manager.account.sign_transaction(buy_tx).rawTransaction.hex()

            bundle_manager = BundleManager(self.flashbots_manager)
            target_block_info = await
self.timing_optimizer.get_optimal_block_timing(
                strategy_type='arbitrage',
                execution_time_estimate=30
            )

            bundle = bundle_manager.create_bundle(
                transactions=[signed_buy_tx],
                target_block=target_block_info['target_block']
            )

            # Step 4: Bundle Submission
            bundle_hash = f"arb_{int(time.time())}"
            self.bundle_tracker.track_bundle_submission(
                bundle_hash=bundle_hash,
                strategy_type='arbitrage',
                target_block=target_block_info['target_block'],
                gas_price=gas_config['gas_price'],
                transaction_count=1,
                estimated_revenue=opportunity['estimated_profit']
            )

            # Submit bundle
            submission_result = await
bundle_manager.submit_bundle(bundle)
```

```
            if submission_result['success']:
                logger.info(f"Bundle submitted successfully:
{bundle_hash}")

                # Monitor bundle inclusion
                monitor_result = await
BundleMonitor(self.flashbots_manager).monitor_bundle_submission(
                    bundle_hash=bundle_hash,
                    target_block=target_block_info['target_block']
                )

                # Update tracker with results
                self.bundle_tracker.update_bundle_status(
                    bundle_hash=bundle_hash,
                    status=monitor_result['status'],

included_block=monitor_result.get('included_block'),
                    revenue=opportunity['estimated_profit'] if
monitor_result['status'] == 'included' else 0
                )

                return {
                    'success': True,
                    'bundle_hash': bundle_hash,
                    'opportunity': opportunity,
                    'bundle_result': submission_result,
                    'monitoring_result': monitor_result,
                    'estimated_revenue':
opportunity['estimated_profit']
                }
            else:
                logger.error(f"Bundle submission failed:
{submission_result}")
                return {
                    'success': False,
                    'reason': 'Bundle submission failed',
                    'error': submission_result
                }

        except Exception as e:
            logger.error(f"Strategy execution failed: {e}")
```

```
            return {
                'success': False,
                'reason': 'Strategy execution failed',
                'error': str(e)
            }

    async def detect_arbitrage_opportunity(self) -> Dict:
        """Detect and evaluate arbitrage opportunities"""

        # This is a simplified example - implement actual opportunity
detection
        # In practice, you would:
        # 1. Monitor multiple DEXs for price differences
        # 2. Calculate potential profits after gas costs
        # 3. Assess execution feasibility

        # Mock opportunity for demonstration
        opportunity = {
            'profitable': True,
            'token_pair': 'ETH/USDC',
            'buy_exchange': 'Uniswap',
            'sell_exchange': 'Sushiswap',
            'price_difference': 2.25,  # $2.25 spread
            'trade_size': 1.0,  # 1 ETH
            'estimated_profit': 2.25 - 1.50,  # Profit after gas costs
            'gas_estimate': 200000,
            'execution_feasibility': 'high',
            'contract_address':
'0x7a250d5630B4cF539739dF2C5dAcb4c659F2488D',  # Uniswap V2 Router
            'swap_data': '0x...'  # Encoded swap data
        }

        return opportunity

# Configuration
config = {
    'private_key': os.getenv('PRIVATE_KEY'),
    'rpc_url': os.getenv('ETH_RPC_URL'),
    'relay_url': 'https://relay.flashbots.net',
    'log_level': 'INFO'
}
```

```
# Run the strategy
async def main():
    strategy = CompleteFlashbotsStrategy(config)
    result = await strategy.run_arbitrage_strategy()

    if result['success']:
        print(f"Strategy executed successfully!")
        print(f"Bundle Hash: {result['bundle_hash']}")
        print(f"Estimated Revenue: ${result['estimated_revenue']:.2f}")
    else:
        print(f"Strategy failed: {result['reason']}")

    # Display analytics
    analytics =
strategy.bundle_tracker.get_bundle_analytics(days_back=7)
    print(f"7-day success rate: {analytics['overall_statistics']
['success_rate']:.2%}")

if __name__ == "__main__":
    asyncio.run(main())
```

Module Summary

Key Takeaways

Flashbots Fundamentals
- Flashbots provides private bundle submission to prevent frontrunning
- Bundles offer guaranteed ordering within blocks through first-price auction
- Direct builder connections enable priority transaction inclusion

Implementation Best Practices
- Always use environment variables for sensitive configuration
- Implement comprehensive error handling and retry logic
- Monitor bundle inclusion and track performance metrics
- Optimize gas prices based on network conditions and strategy urgency

Performance Optimization
- Use dynamic gas pricing based on network congestion
- Implement timing optimization for strategy-specific requirements
- Monitor and track builder performance for continuous improvement
- Apply exponential backoff for rate limit handling

Security Considerations
- Never expose private keys in code or logs
- Use private RPC endpoints for better latency and reliability
- Implement proper authentication for all external services
- Monitor for unusual activity and potential MEV competition

Next Steps

In the next module, you'll learn about **Tenderly Transaction Simulation**, where we'll explore:
- Setting up Tenderly for safe transaction testing
- Creating comprehensive simulation workflows
- Debugging failed transactions with Tenderly's tools
- Optimizing gas usage through simulation analysis
- Implementing automated testing pipelines

Practice Exercises

1. **Basic Setup**: Install Flashbots SDK and create a simple bundle submission
2. **Error Handling**: Implement comprehensive error handling with retry logic
3. **Performance Tracking**: Set up bundle tracking and analytics dashboard
4. **Advanced Optimization**: Create dynamic gas pricing and timing optimization
5. **Complete Strategy**: Implement end-to-end MEV strategy with monitoring

Additional Resources

- Flashbots Documentation
- Builder Directory
- MEV Research Papers
- Community Discord

Author: Obelisk Core
Course: Enthusiast Path - Tool Usage Tutorials
Module 1: Flashbots Setup & Configuration (90 min)
Version: 1.0

This module provides comprehensive coverage of Flashbots integration for MEV extraction strategies. Complete the hands-on exercises to gain practical experience with bundle creation and submission.