Stop-Loss & Take-Profit Systems

Duration: 50 minutes
Level: Beginner-Intermediate
Author: Obelisk Core

Learning Objectives

By the end of this module, you will be able to:

- Implement various stop-loss mechanisms for MEV strategies
- Build automated take-profit systems
- Create dynamic trailing stops
- Design emergency exit procedures
- Integrate exit systems with position sizing

Introduction to Exit Strategies

Exit strategies are as crucial as entry strategies in MEV trading. A well-designed exit system can mean the difference between a small loss and a catastrophic failure. In MEV trading, exits must be fast, automated, and account for the unique characteristics of blockchain transactions.

Why Exit Systems Matter in MEV

- **Gas costs**: Failed transactions still cost gas
- **Slippage**: Poor exits can turn winners into losers
- **Liquidity**: Quick exits may not be possible in illiquid markets
- **MEV extraction**: Other traders may front-run your exits
- **Chain reorganization**: Events can reverse immediately after execution

Types of Exit Triggers

1. **Price-based**: Fixed percentage, volatility-based, technical levels
2. **Time-based**: Maximum holding periods, decay functions
3. **Risk-based**: Stop-loss, maximum drawdown, correlation limits
4. **Event-based**: Protocol upgrades, governance votes, news events
5. **Performance-based**: Profit targets, trailing stops, momentum exits

---

Fixed Stop-Loss Systems

1. Percentage-Based Stop Loss

The simplest form of stop-loss using fixed percentage moves.

Implementation:

```
class PercentageStopLoss:
    def __init__(self, stop_percentage=0.02):
        self.stop_percentage = stop_percentage  # e.g., 2% stop loss

    def calculate_stop_price(self, entry_price, direction='long'):
        """Calculate stop-loss price"""
        if direction == 'long':
            stop_price = entry_price * (1 - self.stop_percentage)
        else:  # short position
            stop_price = entry_price * (1 + self.stop_percentage)

        return stop_price

    def should_exit(self, current_price, entry_price,
direction='long'):
        """Determine if stop-loss should be triggered"""
        if direction == 'long':
            return current_price <=
self.calculate_stop_price(entry_price, direction)
        else:  # short position
            return current_price >=
self.calculate_stop_price(entry_price, direction)

# Example usage
stop_loss = PercentageStopLoss(stop_percentage=0.02)  # 2% stop loss

entry_price = 100
current_prices = [102, 101, 100, 99, 98, 97]

for price in current_prices:
    should_exit = stop_loss.should_exit(price, entry_price, 'long')
    print(f"Price: ${price}, Exit: {should_exit}")
    if should_exit:
        print(f"Stop-loss triggered at ${price}")
        break

# Output:
# Price: $102, Exit: False
# Price: $101, Exit: False
# Price: $100, Exit: False
# Price: $99, Exit: False
```

```
# Price: $98, Exit: True
# Stop-loss triggered at $98
```

2. Volatility-Adjusted Stop Loss

Adjust stop-loss levels based on market volatility to avoid premature exits.

```
import numpy as np
import pandas as pd

class VolatilityAdjustedStopLoss:
    def __init__(self, atr_period=14, stop_multiplier=2.0):
        self.atr_period = atr_period
        self.stop_multiplier = stop_multiplier

    def calculate_atr(self, prices, period=None):
        """Calculate Average True Range"""
        if period is None:
            period = self.atr_period

        if len(prices) < period + 1:
            return None

        # Calculate True Range
        high_low = prices.max() - prices.min()
        high_close = np.abs(prices.max() - prices.shift(1).max())
        low_close = np.abs(prices.min() - prices.shift(1).min())

        true_range = pd.concat([high_low, high_close, low_close],
axis=1).max(axis=1)

        # Calculate ATR
        atr = true_range.rolling(window=period).mean()

        return atr.iloc[-1] if len(atr) > 0 else None

    def calculate_stop_price(self, entry_price, current_volatility,
direction='long'):
        """Calculate volatility-adjusted stop price"""
        if current_volatility is None:
            # Fallback to percentage-based if volatility unavailable
            return entry_price * (1 - 0.02) if direction == 'long' else
entry_price * (1 + 0.02)

        atr_value = current_volatility
        stop_distance = atr_value * self.stop_multiplier

        if direction == 'long':
            stop_price = entry_price - stop_distance
```

```
        else:
            stop_price = entry_price + stop_distance

        return stop_price

    def should_exit(self, current_price, entry_price, price_history,
direction='long'):
        """Determine if stop-loss should be triggered with volatility
adjustment"""
        current_volatility = self.calculate_atr(price_history)
        stop_price = self.calculate_stop_price(entry_price,
current_volatility, direction)

        if direction == 'long':
            return current_price <= stop_price
        else:
            return current_price >= stop_price

# Example usage with realistic price data
np.random.seed(42)
price_data = pd.Series([100 + np.cumsum(np.random.normal(0, 1, 20))])

vol_stop_loss = VolatilityAdjustedStopLoss(atr_period=14,
stop_multiplier=2.0)

entry_price = 100
current_prices = [101, 102, 100, 99, 98, 97]

for i, price in enumerate(current_prices):
    # Use partial price history up to current point
    current_history = price_data[:i+1].append(pd.Series([price]))
    should_exit = vol_stop_loss.should_exit(price, entry_price,
current_history, 'long')

    if i > 0:  # Skip first calculation (insufficient data)
        print(f"Price: ${price}, ATR:
{vol_stop_loss.calculate_atr(current_history):.2f}, Exit:
{should_exit}")
```

Dynamic Stop-Loss Systems

1. Trailing Stop Loss

Trailing stops follow favorable price movements while protecting against reversals.

```
class TrailingStopLoss:
    def __init__(self, trail_percentage=0.03,
initial_stop_percentage=0.05):
        self.trail_percentage = trail_percentage
        self.initial_stop_percentage = initial_stop_percentage
        self.highest_price = None
        self.current_stop = None
        self.position_opened = False

    def update_position(self, current_price, direction='long'):
        """Update trailing stop based on current price"""
        if not self.position_opened:
            self.highest_price = current_price
            if direction == 'long':
                self.current_stop = current_price * (1 -
self.initial_stop_percentage)
            else:
                self.current_stop = current_price * (1 +
self.initial_stop_percentage)
            self.position_opened = True

        # Update trailing stop for long positions
        if direction == 'long':
            # Calculate new stop level
            trail_stop = current_price * (1 - self.trail_percentage)

            # Only move stop up (never down for long positions)
            if trail_stop > self.current_stop:
                self.current_stop = trail_stop

            # Update highest price
            if current_price > self.highest_price:
                self.highest_price = current_price

        # Update trailing stop for short positions
        else:
            # Calculate new stop level
            trail_stop = current_price * (1 + self.trail_percentage)

            # Only move stop down (never up for short positions)
            if trail_stop < self.current_stop:
                self.current_stop = trail_stop
```

```
            # Update lowest price
            if current_price < self.highest_price:
                self.highest_price = current_price

    def should_exit(self, current_price, direction='long'):
        """Determine if trailing stop should be triggered"""
        if direction == 'long':
            return current_price <= self.current_stop
        else:
            return current_price >= self.current_stop

    def get_stop_level(self):
        """Get current stop level"""
        return self.current_stop

# Example usage
trailing_stop = TrailingStopLoss(trail_percentage=0.03,
initial_stop_percentage=0.05)

# Simulate a rising market with a reversal
prices = [100, 103, 106, 109, 108, 105, 102, 100]

print("Trailing Stop Example:")
print("Price\tStop Level\tTriggered")
print("-" * 35)

for price in prices:
    trailing_stop.update_position(price, 'long')
    should_exit = trailing_stop.should_exit(price, 'long')

    print(f"<span class="math-inline" style="display: inline;"><math
xmlns="http://www.w3.org/1998/Math/MathML"
display="inline"><mrow><mrow><mi>p</mi><mi>r</mi><mi>i</mi><mi>c</
mi><mi>e</mi></mrow><mi>\t</mi></mrow></math></
span>{trailing_stop.get_stop_level():.2f}\t{should_exit}")

    if should_exit:
        print(f"Trailing stop triggered at ${price}")
        break
```

```
# Output shows how stop level follows price up, then triggers on
decline
```

2. Chandelier Exit

Uses Average True Range (ATR) to create dynamic stop levels.

```
class ChandelierExit:
    def __init__(self, atr_period=22, multiplier=3.0):
        self.atr_period = atr_period
        self.multiplier = multiplier

    def calculate_chandelier_stop(self, price_data, direction='long'):
        """Calculate Chandelier exit level"""
        if len(price_data) < self.atr_period:
            return None

        # Calculate ATR
        high = price_data.rolling(window=self.atr_period).max()
        low = price_data.rolling(window=self.atr_period).min()
        close = price_data.rolling(window=1).mean()

        atr = ((high - low).rolling(window=self.atr_period).mean() +
               (high.shift(1) -
low.shift(1)).rolling(window=self.atr_period).mean()) / 2

        current_atr = atr.iloc[-1]
        current_high = high.iloc[-1]

        if direction == 'long':
            chandelier_stop = current_high - (current_atr *
self.multiplier)
        else:
            chandelier_stop = current_low + (current_atr *
self.multiplier)

        return chandelier_stop

    def should_exit(self, current_price, price_data, direction='long'):
        """Determine if Chandelier exit should be triggered"""
        chandelier_stop = self.calculate_chandelier_stop(price_data,
direction)

        if chandelier_stop is None:
            return False

        if direction == 'long':
            return current_price <= chandelier_stop
        else:
```

```
            return current_price >= chandelier_stop

# Example with realistic data
np.random.seed(42)
price_series = pd.Series([100] + list(np.cumsum(np.random.normal(0.5,
2, 50))))

chandelier = ChandelierExit(atr_period=22, multiplier=3.0)

# Test exit at different points
test_points = [30, 40, 45]  # Days into the trade

for point in test_points:
    current_price = price_series.iloc[point]
    price_history = price_series.iloc[:point+1]

    should_exit = chandelier.should_exit(current_price, price_history,
'long')
    chandelier_stop =
chandelier.calculate_chandelier_stop(price_history, 'long')

    print(f"Day {point}: Price <span class="math-inline"
style="display: inline;"><math xmlns="http://www.w3.org/1998/Math/
MathML" display="inline"><mrow><mrow><mi>c</mi><mi>u</mi><mi>r</
mi><mi>r</mi><mi>e</mi><mi>n</mi><msub><mi>t</mi><mi>p</mi></
msub><mi>r</mi><mi>i</mi><mi>c</mi><mi>e</mi><mi>:</mi><mi>.2</
mi><mi>f</mi></mrow><mo>&#x0002C;</mo><mi>S</mi><mi>t</mi><mi>o</
mi><mi>p</mi></mrow></math></span>{chandelier_stop:.2f}, Exit:
{should_exit}")
```

Take-Profit Systems

1. Fixed Take-Profit

Simple profit targets based on entry price.

```
class FixedTakeProfit:
    def __init__(self, profit_targets=None):
        """
        Initialize with list of profit targets (as percentages)
        e.g., [0.05, 0.10, 0.20] for 5%, 10%, 20% profit targets
        """
        if profit_targets is None:
            profit_targets = [0.05, 0.10, 0.20]  # 5%, 10%, 20%

        self.profit_targets = sorted(profit_targets, reverse=True)
        self.achieved_targets = []

    def calculate_target_prices(self, entry_price, direction='long'):
        """Calculate absolute target prices"""
        if direction == 'long':
            return [entry_price * (1 + target) for target in
self.profit_targets]
        else:
            return [entry_price * (1 - target) for target in
self.profit_targets]

    def check_targets(self, current_price, entry_price,
direction='long'):
        """Check which profit targets have been achieved"""
        achieved = []
        target_prices = self.calculate_target_prices(entry_price,
direction)

        for target_price in target_prices:
            if direction == 'long' and current_price >= target_price:
                achieved.append(target_price)
            elif direction == 'short' and current_price <=
target_price:
                achieved.append(target_price)

        # Update achieved targets
        for target in achieved:
            if target not in self.achieved_targets:
                self.achieved_targets.append(target)

        return achieved
```

```
    def should_partial_exit(self, current_price, entry_price,
direction='long', exit_percentage=0.5):
        """Determine if partial exit should be taken"""
        achieved = self.check_targets(current_price, entry_price,
direction)

        # Exit if any target is achieved
        return len(achieved) > 0

# Example usage
take_profit = FixedTakeProfit([0.03, 0.07, 0.15])  # 3%, 7%, 15%
targets

entry_price = 100
prices = [103, 107, 112, 115]

print("Fixed Take-Profit Example:")
print("Price\tAchieved Targets\tPartial Exit")
print("-" * 40)

for price in prices:
    targets_achieved = take_profit.check_targets(price, entry_price,
'long')
    partial_exit = take_profit.should_partial_exit(price, entry_price,
'long')

    print(f"${price}\t{targets_achieved}\t\t{partial_exit}")

    if partial_exit and len(targets_achieved) == 1:  # Exit on first
target
        print(f"Taking partial profit at ${price} (first target
reached)")
        break
```

2. Risk-Reward Based Take-Profit

Scale take-profit levels based on initial risk.

```
class RiskRewardTakeProfit:
    def __init__(self, risk_reward_ratios=None):
        """
        Initialize with risk-reward ratios
        e.g., [1, 2, 3] means 1:1, 2:1, 3:1 risk-reward
        """
        if risk_reward_ratios is None:
            risk_reward_ratios = [1, 2, 3]  # 1:1, 2:1, 3:1

        self.risk_reward_ratios = risk_reward_ratios
        self.entry_price = None
        self.stop_loss = None

    def set_entry_and_stop(self, entry_price, stop_loss_price):
        """Set entry price and stop-loss level"""
        self.entry_price = entry_price
        self.stop_loss = stop_loss_price

    def calculate_risk_amount(self):
        """Calculate risk amount based on entry and stop"""
        if self.entry_price is None or self.stop_loss is None:
            return None

        return abs(self.entry_price - self.stop_loss)

    def calculate_target_prices(self, direction='long'):
        """Calculate target prices based on risk-reward ratios"""
        if self.entry_price is None or self.stop_loss is None:
            return []

        risk_amount = self.calculate_risk_amount()
        target_prices = []

        for ratio in self.risk_reward_ratios:
            if direction == 'long':
                target = self.entry_price + (risk_amount * ratio)
            else:
                target = self.entry_price - (risk_amount * ratio)
            target_prices.append(target)

        return target_prices
```

```
    def should_exit(self, current_price, direction='long',
exit_ratio=2):
        """Check if profit target should be taken based on risk-
reward"""
        if self.entry_price is None:
            return False

        target_prices = self.calculate_target_prices(direction)

        # Exit when closest target is reached
        if exit_ratio <= len(self.risk_reward_ratios):
            target_price = target_prices[exit_ratio - 1]  # 0-indexed

            if direction == 'long':
                return current_price >= target_price
            else:
                return current_price <= target_price

        return False

# Example usage
rr_take_profit = RiskRewardTakeProfit([1, 2, 3, 5])  # Various RR
ratios

entry_price = 100
stop_loss = 97  # $3 risk
rr_take_profit.set_entry_and_stop(entry_price, stop_loss)

print("Risk-Reward Take-Profit Example:")
print(f"Entry: <span class="math-inline" style="display:
inline;"><math xmlns="http://www.w3.org/1998/Math/MathML"
display="inline"><mrow><mrow><mi>e</mi><mi>n</mi><mi>t</mi><mi>r</
mi><msub><mi>y</mi><mi>p</mi></msub><mi>r</mi><mi>i</mi><mi>c</
mi><mi>e</mi></mrow><mo>&#x0002C;</mo><mi>S</mi><mi>t</mi><mi>o</
mi><mi>p</mi><mi>:</mi></mrow></math></span>{stop_loss}")
print(f"Risk: ${abs(entry_price - stop_loss)}")
print()

risk_amount = rr_take_profit.calculate_risk_amount()
targets = rr_take_profit.calculate_target_prices('long')

print("Risk-Reward Targets:")
```

```
for i, (ratio, target) in enumerate(zip([1, 2, 3, 5], targets)):
    print(f"{ratio}:1 -> <span class="math-inline" style="display:
inline;"><math xmlns="http://www.w3.org/1998/Math/MathML"
display="inline"><mrow><mrow><mi>t</mi><mi>a</mi><mi>r</mi><mi>g</
mi><mi>e</mi><mi>t</mi><mi>:</mi><mi>.2</mi><mi>f</mi></mrow><mo
stretchy="false">&#x00028;</mo></mrow></math></span>{target -
entry_price:.2f} profit)")

# Test exits
test_prices = [103, 106, 112, 118, 125]

print("\nExit Tests:")
for price in test_prices:
    # Check different exit ratios
    for ratio in [1, 2, 3]:
        should_exit = rr_take_profit.should_exit(price, 'long',
exit_ratio=ratio)
        if should_exit:
            print(f"Price ${price}: Exit at {ratio}:1 target")
            break
    else:
        print(f"Price ${price}: No exit triggered")
```

Advanced Exit Strategies

1. Time-Based Exits

Exit positions based on time rather than price.

```
from datetime import datetime, timedelta

class TimeBasedExit:
    def __init__(self, max_holding_period_hours=24):
        self.max_holding_period_hours = max_holding_period_hours
        self.entry_time = None

    def set_entry_time(self, entry_time=None):
        """Set position entry time"""
        if entry_time is None:
            entry_time = datetime.now()
        self.entry_time = entry_time

    def should_exit_by_time(self, current_time=None):
        """Check if position should exit due to time limit"""
        if current_time is None:
            current_time = datetime.now()

        if self.entry_time is None:
            return False

        time_elapsed = current_time - self.entry_time
        return time_elapsed.total_seconds() / 3600 >=
self.max_holding_period_hours

    def get_remaining_time(self, current_time=None):
        """Get remaining time before forced exit"""
        if current_time is None:
            current_time = datetime.now()

        if self.entry_time is None:
            return self.max_holding_period_hours

        time_elapsed = (current_time -
self.entry_time).total_seconds() / 3600
        return max(0, self.max_holding_period_hours - time_elapsed)

# Example usage
time_exit = TimeBasedExit(max_holding_period_hours=6)  # 6-hour limit

# Simulate time passage
entry_time = datetime.now()
```

```
time_exit.set_entry_time(entry_time)

print("Time-Based Exit Example:")
for hour in range(1, 8):  # Test over 7 hours
    current_time = entry_time + timedelta(hours=hour)
    should_exit = time_exit.should_exit_by_time(current_time)
    remaining = time_exit.get_remaining_time(current_time)

    print(f"Hour {hour}: Should exit: {should_exit}, Remaining:
{remaining:.1f} hours")
```

2. Decay-Based Exit

Reduce position size over time as opportunity decays.

```
class DecayExit:
    def __init__(self, initial_size, decay_half_life_hours=4):
        self.initial_size = initial_size
        self.half_life = decay_half_life_hours
        self.entry_time = None

    def set_entry_time(self, entry_time=None):
        """Set position entry time"""
        if entry_time is None:
            entry_time = datetime.now()
        self.entry_time = entry_time

    def calculate_remaining_size(self, current_time=None):
        """Calculate remaining position size based on decay"""
        if current_time is None:
            current_time = datetime.now()

        if self.entry_time is None:
            return self.initial_size

        time_elapsed = (current_time -
self.entry_time).total_seconds() / 3600

        # Exponential decay: remaining_size = initial_size * 0.5^(time/
half_life)
        decay_factor = 0.5 ** (time_elapsed / self.half_life)
        remaining_size = self.initial_size * decay_factor

        return remaining_size

    def should_complete_exit(self, current_time=None,
min_size_threshold=0.1):
        """Check if position should be completely exited"""
        if current_time is None:
            current_time = datetime.now()

        remaining_size = self.calculate_remaining_size(current_time)
        return remaining_size <= (self.initial_size *
min_size_threshold)

# Example usage
decay_exit = DecayExit(initial_size=1000, decay_half_life_hours=4)
```

```
entry_time = datetime.now()
decay_exit.set_entry_time(entry_time)

print("Decay-Based Exit Example:")
print(f"Initial Size: {decay_exit.initial_size}")
print(f"Half-life: {decay_exit.half_life} hours")
print()

for hour in range(0, 13, 2):  # Check every 2 hours for 12 hours
    current_time = entry_time + timedelta(hours=hour)
    remaining_size = decay_exit.calculate_remaining_size(current_time)
    should_exit = decay_exit.should_complete_exit(current_time)

    print(f"Hour {hour}: Size: {remaining_size:.1f}, Exit:
{should_exit}")
```

MEV-Specific Exit Strategies

1. Gas-Cost Adjusted Exit

Factor in gas costs when deciding to exit positions.

```
class GasAdjustedExit:
    def __init__(self, eth_price=2000, gas_limit=150000,
base_gas_price=20):
        self.eth_price = eth_price
        self.gas_limit = gas_limit
        self.base_gas_price = base_gas_price  # gwei

    def calculate_exit_gas_cost(self, number_of_transactions=1):
        """Calculate total gas cost for exit"""
        gas_cost_eth = (self.gas_limit * self.base_gas_price *
number_of_transactions) / 1e9
        gas_cost_usd = gas_cost_eth * self.eth_price
        return gas_cost_usd

    def should_exit_due_to_gas(self, position_value,
min_profit_margin=0.05):
        """Determine if position should exit due to high gas costs"""
        exit_gas_cost = self.calculate_exit_gas_cost()
        min_profit_needed = exit_gas_cost * (1 + min_profit_margin)

        # Exit if position value is too small relative to gas costs
        return position_value < min_profit_needed

    def calculate_minimum_position_size(self, min_profit_margin=0.05):
        """Calculate minimum position size to justify gas costs"""
        exit_gas_cost = self.calculate_exit_gas_cost()
        return exit_gas_cost * (1 + min_profit_margin)

# Example usage
gas_exit = GasAdjustedExit(eth_price=2000, gas_limit=150000,
base_gas_price=50)  # High gas period

position_values = [100, 500, 1000, 2000, 5000]

print("Gas-Adjusted Exit Example:")
print(f"Exit gas cost: ${gas_exit.calculate_exit_gas_cost():.2f}")
print(f"Minimum position size: $
{gas_exit.calculate_minimum_position_size():.2f}")
print()

for value in position_values:
    should_exit = gas_exit.should_exit_due_to_gas(value)
```

```
    print(f"Position value: ${value}, Should exit due to gas:
{should_exit}")
```

2. Slippage-Aware Exit

Account for slippage when executing exits.

```
class SlippageAwareExit:
    def __init__(self, slippage_factor=0.005):
        self.slippage_factor = slippage_factor  # e.g., 0.5% slippage

    def calculate_slippage_cost(self, position_size, current_price):
        """Calculate cost of exit due to slippage"""
        slippage_amount = position_size * self.slippage_factor
        slippage_cost = slippage_amount * current_price
        return slippage_cost

    def calculate_effective_exit_price(self, target_price,
direction='market'):
        """Calculate effective exit price after slippage"""
        if direction == 'market':
            return target_price * (1 - self.slippage_factor)
        else:
            return target_price  # Limit order, no slippage

    def should_use_limit_order(self, position_size, current_price,
min_slippage_cost=50):
        """Determine if limit order should be used instead of market
order"""
        slippage_cost = self.calculate_slippage_cost(position_size,
current_price)
        return slippage_cost > min_slippage_cost

# Example usage
slippage_exit = SlippageAwareExit(slippage_factor=0.003)  # 0.3%
slippage

position_size = 1000  # $1000 position
current_price = 100

print("Slippage-Aware Exit Example:")
print(f"Position size: ${position_size}")
print(f"Current price: ${current_price}")
print()

slippage_cost = slippage_exit.calculate_slippage_cost(position_size,
current_price)
effective_price =
slippage_exit.calculate_effective_exit_price(current_price, 'market')
```

```
should_use_limit = slippage_exit.should_use_limit_order(position_size,
current_price)

print(f"Slippage cost: ${slippage_cost:.2f}")
print(f"Effective exit price: ${effective_price:.2f}")
print(f"Should use limit order: {should_use_limit}")
```

Comprehensive Exit System

Unified Exit Manager

```
class UnifiedExitSystem:
    def __init__(self, config):
        """
        Config example:
        {
            'stop_loss': {'type': 'percentage', 'value': 0.02},
            'take_profit': {'type': 'fixed', 'targets': [0.05, 0.10,
0.20]},
            'trailing_stop': {'enabled': True, 'percentage': 0.03},
            'time_limit': {'hours': 6},
            'gas_threshold': {'min_position': 100},
            'slippage_threshold': {'max_slippage': 0.005}
        }
        """
        self.config = config
        self.exit_triggers = {}

        # Initialize exit systems
        if config.get('stop_loss'):
            self._init_stop_loss(config['stop_loss'])

        if config.get('take_profit'):
            self._init_take_profit(config['take_profit'])

        if config.get('trailing_stop', {}).get('enabled'):
            self._init_trailing_stop(config['trailing_stop'])

        if config.get('time_limit'):
            self._init_time_exit(config['time_limit'])

    def _init_stop_loss(self, config):
        """Initialize stop-loss system"""
        if config['type'] == 'percentage':
            self.stop_loss = PercentageStopLoss(config['value'])

    def _init_take_profit(self, config):
        """Initialize take-profit system"""
        if config['type'] == 'fixed':
            self.take_profit = FixedTakeProfit(config['targets'])

    def _init_trailing_stop(self, config):
        """Initialize trailing stop"""
```

```
        self.trailing_stop = TrailingStopLoss(
            trail_percentage=config['percentage']
        )

    def _init_time_exit(self, config):
        """Initialize time-based exit"""
        self.time_exit = TimeBasedExit(
            max_holding_period_hours=config['hours']
        )

    def analyze_exit_conditions(self, current_data):
        """
        Analyze all exit conditions

        current_data = {
            'price': current_price,
            'entry_price': entry_price,
            'position_size': position_size,
            'direction': 'long'/'short',
            'entry_time': datetime_object,
            'price_history': pandas_series
        }
        """
        exit_signals = {}
        current_time = datetime.now()

        # Stop-loss check
        if hasattr(self, 'stop_loss'):
            should_stop = self.stop_loss.should_exit(
                current_data['price'],
                current_data['entry_price'],
                current_data['direction']
            )
            exit_signals['stop_loss'] = should_stop

        # Take-profit check
        if hasattr(self, 'take_profit'):
            should_tp = self.take_profit.should_partial_exit(
                current_data['price'],
                current_data['entry_price'],
                current_data['direction']
            )
```

```
            exit_signals['take_profit'] = should_tp

        # Trailing stop check
        if hasattr(self, 'trailing_stop'):
            self.trailing_stop.update_position(
                current_data['price'],
                current_data['direction']
            )
            should_trail = self.trailing_stop.should_exit(
                current_data['price'],
                current_data['direction']
            )
            exit_signals['trailing_stop'] = should_trail

        # Time-based exit
        if hasattr(self, 'time_exit'):
            self.time_exit.set_entry_time(current_data['entry_time'])
            should_time =
self.time_exit.should_exit_by_time(current_time)
            exit_signals['time_exit'] = should_time

        return exit_signals

    def get_recommended_action(self, exit_signals):
        """Get recommended action based on exit signals"""
        # Priority order: stop_loss > take_profit > trailing_stop >
time_exit
        if exit_signals.get('stop_loss'):
            return {'action': 'full_exit', 'reason':
'stop_loss_triggered'}
        elif exit_signals.get('take_profit'):
            return {'action': 'partial_exit', 'reason':
'profit_target_reached'}
        elif exit_signals.get('trailing_stop'):
            return {'action': 'full_exit', 'reason':
'trailing_stop_triggered'}
        elif exit_signals.get('time_exit'):
            return {'action': 'full_exit', 'reason':
'time_limit_reached'}
        else:
            return {'action': 'hold', 'reason': 'no_exit_signals'}
```

```
# Example usage
exit_config = {
    'stop_loss': {'type': 'percentage', 'value': 0.02},
    'take_profit': {'type': 'fixed', 'targets': [0.03, 0.07, 0.15]},
    'trailing_stop': {'enabled': True, 'percentage': 0.03},
    'time_limit': {'hours': 6}
}

exit_system = UnifiedExitSystem(exit_config)

# Simulate position monitoring
current_data = {
    'price': 108,
    'entry_price': 100,
    'position_size': 1000,
    'direction': 'long',
    'entry_time': datetime.now() - timedelta(hours=2),
    'price_history': pd.Series([100, 102, 105, 108])
}

exit_signals = exit_system.analyze_exit_conditions(current_data)
recommended_action = exit_system.get_recommended_action(exit_signals)

print("Exit System Analysis:")
print(f"Current signals: {exit_signals}")
print(f"Recommended action: {recommended_action}")
```

Emergency Exit Procedures

Automated Emergency Exits

```
class EmergencyExitSystem:
    def __init__(self):
        self.emergency_triggers = {
            'protocol_hack': False,
            'extreme_volatility': False,
            'liquidation_cascade': False,
            'network_outage': False
        }
        self.escape_routes = {
            'rapid_exit': {'gas_multiplier': 5, 'slippage_tolerance':
0.1},
            'gradual_exit': {'gas_multiplier': 2, 'slippage_tolerance':
0.05}
        }

    def detect_emergency_conditions(self, market_data):
        """
        Detect emergency conditions that require immediate exit
        """
        alerts = []

        # Check for extreme volatility (price change > 20% in short
period)
        if market_data.get('price_change_1h', 0) > 0.20:
            alerts.append('extreme_volatility_up')
        elif market_data.get('price_change_1h', 0) < -0.20:
            alerts.append('extreme_volatility_down')

        # Check for rapid volume spikes
        if market_data.get('volume_ratio_1h', 1) > 10:
            alerts.append('liquidation_cascade')

        # Check for protocol-specific alerts
        if market_data.get('protocol_alerts'):
            for alert in market_data['protocol_alerts']:
                if alert['severity'] == 'critical':
                    alerts.append('protocol_hack')

        return alerts

    def execute_emergency_exit(self, portfolio_positions,
emergency_type):
```

```
        """Execute emergency exit procedures"""
        if emergency_type in ['protocol_hack', 'liquidation_cascade']:
            # Rapid exit - prioritize speed over price
            exit_params = self.escape_routes['rapid_exit']
            strategy = 'immediate_exit'
        else:
            # Gradual exit - try to minimize losses
            exit_params = self.escape_routes['gradual_exit']
            strategy = 'controlled_exit'

        return {
            'exit_strategy': strategy,
            'gas_multiplier': exit_params['gas_multiplier'],
            'slippage_tolerance': exit_params['slippage_tolerance'],
            'positions_to_close': len(portfolio_positions),
            'estimated_time': '5-15 minutes' if strategy ==
'immediate_exit' else '30-60 minutes'
        }

# Example usage
emergency_system = EmergencyExitSystem()

# Simulate emergency detection
market_data = {
    'price_change_1h': -0.25,  # 25% drop in 1 hour
    'volume_ratio_1h': 15,     # 15x normal volume
    'protocol_alerts': []
}

emergency_alerts =
emergency_system.detect_emergency_conditions(market_data)
if emergency_alerts:
    emergency_exit_plan = emergency_system.execute_emergency_exit(
        portfolio_positions=['ETH/USD', 'BTC/USD', 'LINK/USD'],
        emergency_type='extreme_volatility_down'
    )

    print("Emergency Exit Plan:")
    for key, value in emergency_exit_plan.items():
        print(f"{key}: {value}")
```

Integration with Position Sizing

Dynamic Exit-Based Position Sizing

```
class ExitAwarePositionSizer:
    def __init__(self, base_position_size, risk_percentage=0.02):
        self.base_position_size = base_position_size
        self.risk_percentage = risk_percentage
        self.exit_systems = {}

    def calculate_position_size_with_exit(self, entry_price,
stop_loss_price, opportunity_data):
        """Calculate position size considering exit strategy"""

        # Calculate risk amount
        risk_amount = self.base_position_size * self.risk_percentage

        # Calculate distance to stop-loss
        stop_distance = abs(entry_price - stop_loss_price)

        # Calculate maximum position size based on stop-loss
        max_position_by_stop = risk_amount / stop_distance if
stop_distance > 0 else 0

        # Factor in opportunity quality
        quality_multiplier =
self._assess_opportunity_quality(opportunity_data)

        # Final position size
        position_size = min(max_position_by_stop * quality_multiplier,
self.base_position_size)

        return position_size

    def _assess_opportunity_quality(self, opportunity_data):
        """Assess opportunity quality and return multiplier"""
        score = 1.0

        # Factor in win rate if available
        if 'win_rate' in opportunity_data:
            score *= (1 + opportunity_data['win_rate'] * 0.5)

        # Factor in expected profit
        if 'expected_profit' in opportunity_data:
            profit_multiplier =
min(opportunity_data['expected_profit'] / 100, 2.0)
```

```
            score *= profit_multiplier

        # Factor in time sensitivity
        if 'time_sensitivity' in opportunity_data:
            score *= opportunity_data['time_sensitivity']

        # Cap the multiplier to prevent over-sizing
        return min(score, 3.0)

    def update_exit_system(self, opportunity_id, exit_config):
        """Update exit system for a specific opportunity"""
        self.exit_systems[opportunity_id] =
UnifiedExitSystem(exit_config)

# Example usage
position_sizer = ExitAwarePositionSizer(base_position_size=10000,
risk_percentage=0.02)

# Opportunity data
opportunity = {
    'win_rate': 0.75,
    'expected_profit': 150,  # $150 expected profit
    'time_sensitivity': 1.2  # 20% bonus for time sensitivity
}

entry_price = 100
stop_loss_price = 97

position_size = position_sizer.calculate_position_size_with_exit(
    entry_price, stop_loss_price, opportunity
)

print(f"Base position size: ${position_sizer.base_position_size}")
print(f"Calculated position size: ${position_size:.2f}")
print(f"Risk per trade: ${position_size * 0.02:.2f}")
print(f"Stop distance: ${abs(entry_price - stop_loss_price):.2f}")
```

Key Takeaways

1. **Multiple exit triggers provide comprehensive protection** - Don't rely on a single exit method.

2. **Dynamic exits adapt to market conditions** - Fixed exits may be too rigid for volatile MEV markets.
3. **Time-based exits prevent capital lockup** - MEV opportunities often have limited lifespans.
4. **Gas costs affect exit decisions** - High gas periods may justify holding losing positions.
5. **Emergency procedures save capital** - Pre-planned responses to market stress are crucial.
6. **Exit systems should be tested extensively** - Backtest your exit logic with historical data.
7. **Automation reduces emotional decisions** - Manual exits often lead to poor outcomes.
8. **Integration with position sizing improves risk management** - Exit costs should influence position sizing.

Next Steps

In the next module, **Strategy Diversification**, you'll learn to:

- Build diversified MEV strategy portfolios
- Analyze correlation between strategies
- Optimize allocation across different approaches
- Create dynamic rebalancing systems
- Implement correlation monitoring

This will complete your foundational risk management knowledge before moving to advanced portfolio-level techniques.

Summary

Effective exit systems are the cornerstone of risk management in MEV trading. By implementing comprehensive exit strategies that combine fixed stops, trailing stops, profit targets, and emergency procedures, you can protect your capital while maximizing the potential of your strategies.

Remember that the best exit system is one that you can trust and follow consistently. The key is to balance protection with opportunity - too tight exits may limit profits, while too loose exits may allow small losses to become large ones. Regular testing and refinement of your exit systems will help you find the optimal balance for your specific strategies and risk tolerance.