1. IDSCore - The Main Engine 🚀

Purpose:

- **Central coordinator** that manages the entire IDS lifecycle
- **Orchestrates all components** and handles packet flow
- **Manages resources** and system state

Key Methods & How They Work:

```
public void Start()
```

What it does:

- Initializes the rule engine and loads detection rules
- Creates worker threads for parallel packet processing
- Starts packet capture (live interface or PCAP file)
- Starts maintenance timers for cleanup and statistics

```
private void ProcessPackets(CancellationToken token)
```

How it works:

- Each worker thread runs this method continuously
- Takes packets from `_packetQueue` using `_packetQueue.Take(token)`
- Processes each packet through `ProcessPacket(wrapper)`
- Implements **backpressure**: if queue is 70% full, workers delay briefly
- Logs performance metrics every 1000 packets

Why We Need Workers: 🛠️

```
_workerCount = Math.Max(2, Environment.ProcessorCount / 2);
```

- **Parallel Processing**: Multiple threads process packets simultaneously
- **Load Distribution**: Prevents packet loss during traffic bursts
- **CPU Utilization**: Uses multiple cores efficiently
- **Fault Isolation**: If one worker fails, others continue

2. Packet Queue System 📦

What is the Queue?

```
private BlockingCollection<PacketCaptureWrapper> _packetQueue;
```

- **Thread-safe collection** that holds packets waiting for processing
- **Fixed capacity** (configurable, default 5000 packets)
- **Blocks when full** to prevent memory overflow

Why We Need the Queue: 🎯

```
// Without queue: Packet loss during traffic spikes
// With queue: Smooth processing with buffering
if (!_packetQueue.TryAdd(wrapper, 50))
{
    Interlocked.Increment(ref _errorCount);
    OptimizedLogger.LogQueue($"Packet dropped - queue full");
}
```

Benefits:

- **Decouples Capture from Processing**: Capture thread isn't blocked by slow processing
- **Handles Traffic Bursts**: Absorbs sudden traffic increases
- **Provides Backpressure**: When queue fills, capture slows down
- **Enables Load Balancing**: Multiple workers share the load

3. Packet Processing Pipeline 🔄

Step-by-Step Flow:

```
private void ProcessPacket(PacketCaptureWrapper wrapper)
{
    // 1. Parse raw packet
    var packet = Packet.ParsePacket(wrapper.LinkLayerType, wrapper.Data);

    // 2. Route by link layer type
    switch (wrapper.LinkLayerType)
    {
        case LinkLayers.Ethernet:
            ProcessEthernet(packet as EthernetPacket);
            break;
        case LinkLayers.Ieee80211:
            ProcessWiFi(packet);
            break;
    }
}
```

Detailed Processing Chain:

A. Ethernet Processing:

```
private void ProcessEthernet(EthernetPacket ethPacket)
{
    if (ethPacket.PayloadPacket is IPPacket ip)
    {
        // Handle IP fragmentation reassembly
        var reassembledPacket = _ipReassemblyManager.ProcessFragment(ip);
        if (reassembledPacket != null)
        {
            ProcessIPPacket(reassembledPacket);  // Process complete packet
        }
    }
}
```

B. IP Packet Processing:

```
csharp
private void ProcessIPPacket(IPPacket ipPacket)
{
    string srcIp = ipPacket.SourceAddress.ToString();
    string dstIp = ipPacket.DestinationAddress.ToString();

    // TCP Stream Reassembly for stateful protocols
    if (ipPacket.PayloadPacket is TcpPacket tcp)
    {
        var completeMessages = _tcpStreamReassembler.ProcessTcpSegment(ipPacket
, tcp);
        foreach (var message in completeMessages)
        {
            ProcessCompleteTcpMessage(ipPacket, tcp, message, srcIp, dstIp, src
Port, dstPort);
        }
    }

    ProcessPacketData(ipPacket, payload, srcIp, dstIp, srcPort, dstPort, protoc
olName, tcpFlags);
}
```

4. TCP Stream Reassembler 🔗

Why We Need It:

- **Protocol Understanding**: Some protocols (SMB, HTTP) require complete messages
- **Fragmentation Handling**: TCP segments may arrive out-of-order
- **Stateful Analysis**: Detect attacks that span multiple packets

How It Works:

```
csharp
public List<byte[]> ProcessTcpSegment(IPv4Packet ip, TcpPacket tcp)
{
    // 1. Identify TCP connection
    string key = $"{ip.SourceAddress}:{tcp.SourcePort}-{ip.DestinationAddress}:
{tcp.DestinationPort}";

    // 2. Store fragments in sorted order
    if (!conn.Fragments.ContainsKey(tcp.SequenceNumber))
        conn.Fragments.Add(tcp.SequenceNumber, tcp.PayloadData);

    // 3. Merge contiguous fragments
    MergeFragments(conn);
    // 4. Extract complete messages
    return ExtractMessages(conn, key);
}
```

5. Rule Engine & Fast Path Optimization ⚡

The Optimization Strategy:

```
csharp
public bool ShouldCheckRules(string protocol, int dstPort, int payloadLength, s
tring ruleGroup)
{
    // 1. Quick rejection of obviously non-matching packets
    if (payloadLength == 0 && !IsProtocolWithZeroLengthValid(protocol))
        return false;

    // 2. Bloom Filter check - probabilistic fast path
    if (_bloomFilters.TryGetValue(protocol, out var protoFilter))
    {
        if (protoFilter.MightMatch(protocol, dstPort, payloadLength))
            return true; // Possible match, need detailed check
    }

    return false; // Definitely no match, skip rule checking
}
```

Bloom Filter Magic: 🎩

```
csharp
public class RuleBloomFilter
{
    private readonly BitArrayWrapper _bits;

    public void AddRule(Entity.Signatures rule)
    {
        // Convert rule characteristics to hash positions
        var hashes = GetRuleHashes(rule);
        foreach (var hash in hashes)
```

```
            {
                int idx = Math.Abs(hash % _filterSize);
                _bits.Set(idx);  // Mark position as "might match"
            }
        }
}
```

Why This is Fast:

- **Memory Efficient**: 1 bit per rule characteristic vs storing full rules
- **Constant Time**: O(1) checks regardless of rule count
- **False Positives OK**: Says "might match" when no match exists (safe)
- **No False Negatives**: Never says "no match" when match exists (critical)

6. Protocol-Specific Analysis 🔍

How Protocol Parsing Works:

```
csharp
private void ProcessAllProtocols(int logId, IPPacket ip, byte[] payload, string
srcIp, string dstIp, int srcPort, int dstPort)
{
    // Dispatch to specialized parsers based on port/protocol
    if (tcp != null && (dstPort == 80 || srcPort == 80))
    {
        ProcessHttpHttps(logId, payload, srcIp, dstIp, srcPort, dstPort);
    }
    else if (udp != null && (dstPort == 53 || srcPort == 53))
    {
        ProcessDns(logId, udp, srcIp, dstIp);
    }
    else if (tcp != null && (dstPort == 445 || srcPort == 445))
    {
        ProcessEnhancedSmbProtocol(logId, payload, srcIp, dstIp, srcPort, dstPo
rt);
    }
    // ... and 10+ other protocols
}
```

Example: SMB Processing:

csharp

```
private void ProcessEnhancedSmbProtocol(int logId, byte[] smbPayload, string sr
cIp, string dstIp, int srcPort, int dstPort)
{
    // 1. Validate SMB structure
    if (SmbDetectionHelper.ValidateSmbStructure(smbPayload))
    {
        // 2. Parse SMB commands and files
        var result = _smbParser.Parse(smbPayload);

        // 3. Security analysis
        var securityAnalysis = _enhancedSmbParser.AnalyzeSecurity(result, srcIp
, dstIp);

        // 4. Brute force detection
        bool isBruteForce = _smbBruteForceDetector.DetectBruteForce(srcIp, resu
lt);

        // 5. Log and alert
        SmbLogBLL.Insert(logId, result.Command, result.Filename, result.Share);
    }
}
```

7. Performance Optimization System 🏎️

Worker Thread Management:

csharp

```
// During startup:
_workerTasks.Clear();
_workerCount = Math.Max(2, Environment.ProcessorCount / 2);

for (int i = 0; i < _workerCount; i++)
{
    _workerTasks.Add(Task.Run(() => ProcessPackets(_cts.Token)));
}
```

Worker Lifecycle:

1. **Start**: Each worker begins consuming from `_packetQueue`
2. **Process**: Applies full analysis pipeline to each packet
3. **Monitor**: Tracks processing time and performance
4. **Backpressure**: Slows down when queue is overloaded
5. **Graceful Shutdown**: Responds to cancellation token

Queue Management Strategy:

```csharp
// Producer side (packet capture)
void OnPacketArrival(object sender, PacketCapture e)
{
    var wrapper = new PacketCaptureWrapper(rawCapture);
    if (!_packetQueue.TryAdd(wrapper, 50)) // Try for 50ms
    {
        // Queue full - increment error counter
        Interlocked.Increment(ref _errorCount);
    }
}

// Consumer side (worker threads)
void ProcessPackets(CancellationToken token)
{
    while (!token.IsCancellationRequested && !_packetQueue.IsCompleted)
    {
        var wrapper = _packetQueue.Take(token); // Blocks until packet availabl
e

        // Backpressure: slow down if queue is getting full
        if (_packetQueue.Count > _packetQueue.BoundedCapacity * 0.7)
        {
            await Task.Delay(1, token);
        }
    }
}
```

8. Detection & Alerting System 🚨

Rule Matching Process:

csharp
```
public List<Entity.Signatures> CheckPacket(IPPacket ipPacket, byte[] payload, string srcIp, string dstIp, int srcPort, int dstPort, string protocolName)
{
    // 1. Fast path optimization
    if (!_fastPathOptimizer.ShouldCheckRules(protocolName, dstPort, payload?.Length ?? 0, "default"))
        return new List<Entity.Signatures>(); // Skip detailed checking

    // 2. Get relevant rules for this traffic
    var relevantRules = GetRelevantRulesForTraffic(protocolName, dstPort, srcIp, dstIp, payload?.Length ?? 0);

    // 3. Apply each rule
    foreach (var rule in relevantRules.Take(_perfSettings.MaxRulesPerPacket))
    {
        if (MatchesRuleFast(rule, ipPacket, payload, srcIp, dstIp, srcPort, dstPort, protocolName))
        {
            matches.Add(rule);
            GenerateAlert(logId, srcIp, dstIp, payload.Length, rule.AttackName, "RuleMatch", rule.SignatureId, "High");
        }
    }

    return matches;
}
```

Alert Generation:

csharp
```
private void GenerateAlert(int logId, string src, string dst, double size, string sig, string method, int? sigId, string sev)
{
    // Deduplication to prevent alert flooding
    string alertKey = $"{method}-{src}-{dst}-{sig}";
    if (_recentAlertsCache.TryGetValue(alertKey, out var lastAlertTime) &&
        DateTime.Now - lastAlertTime < TimeSpan.FromMinutes(2))
    {
        return; // Skip duplicate alert
    }

    // Insert alert into database
    if (AlertBLL.Insert(logId, sig, type, severity, src, dst, "", DateTime.Now,
"New"))
    {
        OptimizedLogger.LogImportant($"ALERT: {method} from {src} to {dst} - {t
ype} ({severity})");
    }
}
```

9. System Architecture Benefits 🏗️

Why This Design Works:

Scalability:

- **Horizontal**: Add more workers for higher traffic
- **Vertical**: Tune queue size and processing limits

Resilience:

- **Graceful Degradation**: When overloaded, slows down rather than crashing
- **Fault Isolation**: Worker failures don't affect entire system
- **Resource Management**: Automatic cleanup prevents memory leaks

Performance:

- **Parallel Processing**: Multiple workers handle packets simultaneously
- **Memory Efficiency**: Bloom filters and caching reduce overhead
- **Intelligent Filtering**: Fast path avoids unnecessary work

Maintainability:

- **Modular Design**: Each component has single responsibility
- **Configurable**: Performance settings adjustable without code changes
- **Monitorable**: Comprehensive logging and statistics