

Forum:[Hack Forums](#)

Thread: [Device Fingerprinting & Signature Matching System](#)

itcorp - July 4, 2026, 10:11 am

I want to share our device fingerprinting system that can identify hundreds of different devices based on their network signatures. This system is designed for security monitoring, asset discovery, and vulnerability assessment, and is also intended as a small addendum to the previous thread to encourage the HF JULY CONTEST.

Core Architecture

The system uses a comprehensive signature-based approach to identify devices by analyzing:

- Banner Grabbing - Service banners from various ports
- HTTP Headers - Web server responses
- HTML Content - Web interface fingerprints
- Endpoints - Specific paths that indicate device type
- Port Combinations - Open ports patterns

DeviceSignatureMatcher Implementation

```
Code public class DeviceSignatureMatcher : IDeviceSignatureMatcher, IDisposable
{
    private readonly ILogger<DeviceSignatureMatcher> _logger;
    private readonly IDeviceAnalysisConfigProvider _configProvider;
    private readonly CancellationTokenSource _internalCts = new CancellationTokenSource();
    private bool _disposed;

    public DeviceSignatureMatcher(
        ILogger<DeviceSignatureMatcher> logger,
        IDeviceAnalysisConfigProvider configProvider)
    {
        _logger = logger ?? throw new ArgumentNullException(nameof(logger));
        _configProvider = configProvider ?? throw new
ArgumentNullException(nameof(configProvider));
    }

    public async Task<DeviceSignatureMatchResult> MatchSignaturesAsync(
        AnalysisResult analysisResult,
        CancellationToken ct)
    {
        var result = new DeviceSignatureMatchResult();

        if (string.IsNullOrEmpty(analysisResult.Banner) &&
            (analysisResult.HttpHeaders == null || !analysisResult.HttpHeaders.Any()))
        {
            _logger.LogInformation("No banner or headers available for signature matching");
            return result;
        }
    }
```

```

try
{
    var headerText = string.Join("\n",
        analysisResult.HttpHeaders?.Select(h => $"{h.Key}: {h.Value}") ??
Array.Empty<string>());

    var combinedText =
$"{analysisResult.Banner}\n{analysisResult.HtmlContent}\n{headerText}".ToLowerInvariant();

    _logger.LogInformation("Matching against combined text");

    // Create a linked token source with a timeout
    using var linkedCts = CancellationTokenSource.CreateLinkedTokenSource(ct,
_internalCts.Token);
    linkedCts.CancelAfter(TimeSpan.FromSeconds(30));

    var matches = await Task.Run(() =>
    {
        var foundMatches = new List<(string Type, DeviceSignatureConfig Signature)>();

        foreach (var (deviceType, signature) in _configProvider.DeviceSignatures)
        {
            // Check if port matches
            if (!signature.Ports.Contains(analysisResult.Port))
            {
                continue;
            }

            try
            {
                linkedCts.Token.ThrowIfCancellationRequested();

                // Check each pattern
                foreach (var pattern in signature.Patterns)
                {
                    try
                    {
                        if (string.IsNullOrEmpty(pattern))
                        {
                            _logger.LogWarning("Empty pattern found for device {DeviceType}",
deviceType);
                            continue;
                        }

                        var isMatch = Regex.IsMatch(combinedText, pattern,
RegexOptions.IgnoreCase);
                        if (isMatch)
                        {
                            _logger.LogDebug("Pattern matched - Device: {DeviceType}, Pattern:
{Pattern}",
                                deviceType, pattern);
                            foundMatches.Add((deviceType, signature));
                        }
                    }
                    catch { }
                }
            }
            catch { }
        }
    });
}

```

```

        break;
    }
}
catch (RegexParseException ex)
{
    _logger.LogWarning(ex, "Invalid regex pattern {Pattern} for device
{DeviceType}",
        pattern, deviceType);
}
}
}
catch (OperationCanceledException)
{
    _logger.LogWarning("Device signature matching timed out for {DeviceType}",
deviceType);
    throw;
}
catch (Exception ex)
{
    _logger.LogWarning(ex, "Error matching device signature {DeviceType}",
deviceType);
}
}
return foundMatches;
}, linkedCts.Token);

if (matches.Any())
{
    // Find the best match
    var bestMatch = matches
        .OrderByDescending(m => m.Signature.Patterns.Max(p => p?.Length ?? 0))
        .First();

    result.DeviceType = bestMatch.Type;
    result.Categories.AddRange(bestMatch.Signature.Categories);
    result.Confidence = CalculateConfidence(combinedText, bestMatch.Signature);

    result.Metadata["MatchedPatterns"] = bestMatch.Signature.Patterns
        .Where(p => !string.IsNullOrEmpty(p) &&
            Regex.IsMatch(combinedText, p, RegexOptions.IgnoreCase))
        .ToList();

    _logger.LogInformation("Matched device: {DeviceType} with confidence {Confidence}",
        result.DeviceType, result.Confidence);
}
else
{
    _logger.LogInformation("No device signatures matched");
}
}
catch (OperationCanceledException)
{
    _logger.LogWarning("Device signature matching timed out for {IP}:{Port}",

```

```

        analysisResult.IpAddress, analysisResult.Port);
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, "Device signature matching failed");
    }

    return result;
}

private float CalculateConfidence(string text, DeviceSignatureConfig signature)
{
    var matchingPatterns = signature.Patterns
        .Where(p => !string.IsNullOrEmpty(p) &&
            Regex.IsMatch(text, p, RegexOptions.IgnoreCase))
        .ToList();

    if (matchingPatterns.Count == 0) return 0;

    // Calculate score based on pattern matches and port
    var patternScore = matchingPatterns.Sum(p => p.Length) / (float)matchingPatterns.Count;
    var portScore = signature.Ports.Any(p => p < 1024) ? 0.7f : 1.0f;

    // Weighted average (60% pattern score, 40% port score)
    var confidence = patternScore * 0.6f + portScore * 0.4f;

    return Math.Min(confidence, 1.0f);
}

public void Dispose()
{
    Dispose(true);
    GC.SuppressFinalize(this);
}

protected virtual void Dispose(bool disposing)
{
    if (!_disposed)
    {
        if (disposing)
        {
            _internalCts?.Cancel();
            _internalCts?.Dispose();
        }
        _disposed = true;
    }
}
}

```

Supported Device Categories

Networking Infrastructure

Code "Cisco_IOS_Device": {

```

"Patterns": [
  "Cisco IOS",
  "Cisco Catalyst",
  "Cisco Nexus",
  "Cisco Router",
  "Cisco Switch",
  "Cisco ASA"
],
"Ports": [22, 23, 80, 443, 161, 2000, 4000, 5060, 8000, 8080, 8443],
"Categories": ["Networking", "Router", "Switch", "Firewall"],
"Vendors": ["Cisco"],
"Models": [
  "Catalyst 9200",
  "Nexus 9000",
  "ISR 4000",
  "ASA 5500"
]
}

```

Other Network Vendors: Juniper - MX Series, EX Series, SRX Series

MikroTik - RouterOS, RB Series

Ubiquiti - EdgeRouter, UniFi, airOS

TP-Link - Archer Series, TL-WR Series

Netgear - Nighthawk, Orbi

ASUS - RT Series, ASUSWRT

Fortinet - FortiGate, FortiOS

Palo Alto - PAN-OS, PA Series

SonicWall - TZ Series, NSA Series

D-Link - DIR Series, DSL Series

Surveillance Systems:

Hikvision

```

Code "Hikvision_Camera": {
  "Patterns": [
    "Hikvision",
    "Hik-Connect",
    "DS-2CD",
    "IPC-",
    "NVR-",
    "Hikvision ISAPI"
  ],
  "Ports": [80, 443, 554, 8000, 37777, 8200],
  "Categories": ["Surveillance", "IP Camera", "NVR"],
  "Vendors": ["Hikvision"],
  "Models": [
    "DS-2CD2342WD-I",
    "DS-2CD2142FWD-I",
    "DS-7608NI-K2"
  ],
  "Vulnerabilities": [
    {
      "CVE": "CVE-2017-7921",

```

```

    "Description": "Authentication bypass",
    "Severity": "Critical"
  }
]
}

```

Camera Vendors Supported: Dahua - Authentication bypass (CVE-2021-33044)

Axis - Hardcoded credentials (CVE-2018-10660)

Bosch - DINION, AUTODOME, FLEXIDOME

Hanwha - Wisenet, Samsung SNO/SND/SNP

Uniview - IPC, NVR Series

Vivotek - CC, FD Series

Pelco - Sarix, Spectra

Avigilon - H4A, H5A Series

Amcrest - IP2M, IP3M, IP4M, IP5M Series

Reolink - RLC Series

Foscam - FI Series

Lorex - LNB Series

TP-Link - NC Series

D-Link - DCS Series

Industrial Control Systems

```

Code "Siemens_S7_PLC": {
  "Patterns": [
    "Siemens",
    "SIMATIC S7",
    "S7-1200",
    "S7-1500",
    "S7Comm",
    "PROFINET"
  ],
  "Ports": [80, 102, 443, 502, 161, 1023],
  "Categories": ["Industrial", "PLC", "SCADA"],
  "Vendors": ["Siemens"],
  "Models": [
    "S7-1200",
    "S7-1500",
    "ET 200SP"
  ]
}

```

Smart Home

```

Code "Samsung_SmartTV": {
  "Patterns": [
    "Samsung SmartTV",
    "Tizen TV",
    "Samsung QLED",
    "Samsung UHD"
  ],
  "Ports": [8001, 8002, 8080, 9197],
  "Categories": ["Consumer", "Television", "Smart Home"],
  "Vendors": ["Samsung"],
  "Models": [

```

```

    "UE55AU8000",
    "QE65QN90B"
]
}

```

Smart Home Supported:

G ThinQ - Smart Fridge, TV, Washer
 Honeywell - Thermostats (T6, Lyric T5)
 Samsung - Smart TVs, SmartThings
 Apple - iPhone, iPad, Apple TV, HomePod
 Tesla - Wall Charger, Powerwall

ETC ETC

Confidence Scoring Algorithm

```

Code private float CalculateConfidence(string text, DeviceSignatureConfig signature)
{
    var matchingPatterns = signature.Patterns
        .Where(p => !string.IsNullOrEmpty(p) &&
            Regex.IsMatch(text, p, RegexOptions.IgnoreCase))
        .ToList();

    if (matchingPatterns.Count == 0) return 0;

    // Pattern score based on length of matched patterns
    var patternScore = matchingPatterns.Sum(p => p.Length) / (float)matchingPatterns.Count;

    // Port score (privileged ports indicate more reliable detection)
    var portScore = signature.Ports.Any(p => p < 1024) ? 0.7f : 1.0f;

    // Weighted average (60% pattern score, 40% port score)
    var confidence = patternScore * 0.6f + portScore * 0.4f;

    return Math.Min(confidence, 1.0f);
}

```

Usage Example

```

Code var analysisResult = new AnalysisResult
{
    IpAddress = "192.168.1.100",
    Port = 80,
    Banner = "HTTP/1.1 200 OK\r\nServer: Cisco-IOS\r\n",
    HttpHeaders = new List<HttpHeader>
    {
        new() { Key = "Server", Value = "Cisco-IOS" }
    },
    HtmlContent = "<html><title>Cisco Device Manager</title></html>"
};

```

```

var matcher = new DeviceSignatureMatcher(logger, configProvider);
var result = await matcher.MatchSignaturesAsync(analysisResult, CancellationToken.None);

```

```

Console.WriteLine($"Device Type: {result.DeviceType}");

```

```
Console.WriteLine($"Confidence: {result.Confidence:P2}");
Console.WriteLine($"Categories: {string.Join(", ", result.Categories)}");
Console.WriteLine($"Matched Patterns: {string.Join(", ", result.Metadata["MatchedPatterns"])}");
Code using System;
using System.Collections.Generic;
```

```
namespace SOC.Agent.Models
```

```
{
    // Configuration Models
    public class DeviceSignatureConfig
    {
        public string[]? Patterns { get; set; }
        public int[]? Ports { get; set; }
        public string[]? Categories { get; set; }
        public string[]? Vendors { get; set; }
        public HttpHeaders[]? HttpHeaders { get; set; }
        public Endpoint[]? Endpoints { get; set; }
        public string[]? BannerPatterns { get; set; }
        public string[]? Models { get; set; }
        public string[]? FirmwarePatterns { get; set; }
        public Vulnerability[]? Vulnerabilities { get; set; }
        public SNMPConfig? SNMP { get; set; }
        public SSHConfig? SSH { get; set; }
        public TelnetConfig? Telnet { get; set; }
        public UPnPConfig? UPnP { get; set; }
        public Dictionary<string, object>? Metadata { get; set; }
    }

    public class HttpHeader
    {
        public string? Key { get; set; }
        public string? Value { get; set; }
    }

    public class Endpoint
    {
        public string? Path { get; set; }
        public string? Method { get; set; }
        public string? Description { get; set; }
        public Dictionary<string, string>? Parameters { get; set; }
        public bool IsVulnerable { get; set; }
    }

    public class Vulnerability
    {
        public string? CVE { get; set; }
        public string? Description { get; set; }
        public string? Severity { get; set; }
        public string? CVSS { get; set; }
        public bool IsExploitable { get; set; }
    }

    public class SNMPConfig
```

```

{
    public string[]? Olds { get; set; }
    public string[]? CommunityStrings { get; set; }
}

public class SSHConfig
{
    public string? Banner { get; set; }
    public string[]? AuthMethods { get; set; }
}

public class TelnetConfig
{
    public string? LoginPrompt { get; set; }
    public string? PasswordPrompt { get; set; }
}

public class UPnPConfig
{
    public string[]? DeviceTypes { get; set; }
    public string[]? Services { get; set; }
}

public class VendorProfileConfig
{
    public string? Name { get; set; }
    public string? Website { get; set; }
    public string[]? MacOuiPrefixes { get; set; }
    public string[]? HttpHeaders { get; set; }
    public string[]? DhcpFingerprints { get; set; }
    public string[]? UpnpFingerprints { get; set; }
    public Dictionary<string, object>? Metadata { get; set; }
}

// Analysis Input Models
public class AnalysisResult
{
    public string? IpAddress { get; set; }
    public int Port { get; set; }
    public string? Protocol { get; set; }
    public string? Banner { get; set; }
    public string? HtmlContent { get; set; }
    public List<HTTPHeader>? HttpHeaders { get; set; }
    public Dictionary<string, object>? AdditionalData { get; set; }
    public DateTime Timestamp { get; set; } = DateTime.UtcNow;
}

// Result Models
public class DeviceSignatureMatchResult
{
    public string? DeviceType { get; set; }
    public string? DeviceModel { get; set; }
    public string? FirmwareVersion { get; set; }
}

```

```

    public float Confidence { get; set; }
    public List<string> Categories { get; set; } = new();
    public List<string> Vendors { get; set; } = new();
    public List<MatchedPattern> MatchedPatterns { get; set; } = new();
    public List<DetectedVulnerability> DetectedVulnerabilities { get; set; } = new();
    public Dictionary<string, object> Metadata { get; set; } = new();
    public DeviceFingerprint? Fingerprint { get; set; }
    public DateTime Timestamp { get; set; } = DateTime.UtcNow;
}

```

```

public class MatchedPattern
{
    public string? Pattern { get; set; }
    public string? Source { get; set; }
    public float Confidence { get; set; }
    public string? MatchedText { get; set; }
}

```

```

public class DetectedVulnerability
{
    public string? CVE { get; set; }
    public string? Description { get; set; }
    public string? Severity { get; set; }
    public string? CVSS { get; set; }
    public bool IsExploitable { get; set; }
    public float Confidence { get; set; }
    public string? AffectedComponent { get; set; }
}

```

```

public class DeviceFingerprint
{
    public string? DeviceType { get; set; }
    public string? Vendor { get; set; }
    public string? Model { get; set; }
    public string? Firmware { get; set; }
    public string? Hardware { get; set; }
    public string? SerialNumber { get; set; }
    public string? MacAddress { get; set; }
    public Dictionary<string, string> AdditionalAttributes { get; set; } = new();
}
}

```

Code using System.Collections.Generic;
using SOC.Agent.Models;

```

namespace SOC.Agent.Interfaces
{
    public interface IDeviceAnalysisConfigProvider
    {
        DeviceAnalysisConfig AnalysisConfig { get; }
        ProtocolAnalyzerConfig ProtocolAnalyzerConfig { get; }
        IReadOnlyDictionary<string, DeviceSignatureConfig> DeviceSignatures { get; }
        IReadOnlyDictionary<string, VendorProfileConfig> VendorProfiles { get; }
        IReadOnlyDictionary<string, Dictionary<string, List<string>>> DeviceModels { get; }
    }
}

```

```

        IReadOnlyDictionary<string, SoftwareSignature> WebSoftwareSignatures { get; }
        IReadOnlyDictionary<string, TechnologySignature> TechnologySignatures { get; }
        IReadOnlyDictionary<string, VulnerabilitySignature> VulnerabilitySignatures { get; }
        DeviceDetectionSettings DeviceDetectionSettings { get; }

        void ReloadConfiguration();
        bool TryGetDeviceSignature(string deviceType, out DeviceSignatureConfig signature);
        bool TryGetVendorProfile(string vendor, out VendorProfileConfig profile);
    }
}

Code using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using Microsoft.Extensions.Configuration;
using SOC.Agent.Interfaces;
using SOC.Agent.Models;

namespace SOC.Agent.Providers
{
    public class JsonDeviceAnalysisConfigProvider : IDeviceAnalysisConfigProvider
    {
        private readonly IConfigurationRoot _configuration;
        private readonly string _configPath;

        public DeviceAnalysisConfig AnalysisConfig { get; private set; }
        public ProtocolAnalyzerConfig ProtocolAnalyzerConfig { get; private set; }
        public IReadOnlyDictionary<string, DeviceSignatureConfig> DeviceSignatures { get; private
set; }
        public IReadOnlyDictionary<string, VendorProfileConfig> VendorProfiles { get; private set; }
        public IReadOnlyDictionary<string, Dictionary<string, List<string>>> DeviceModels { get;
private set; }
        public IReadOnlyDictionary<string, SoftwareSignature> WebSoftwareSignatures { get; private
set; }
        public IReadOnlyDictionary<string, TechnologySignature> TechnologySignatures { get; private
set; }
        public IReadOnlyDictionary<string, VulnerabilitySignature> VulnerabilitySignatures { get;
private set; }
        public DeviceDetectionSettings DeviceDetectionSettings { get; private set; }

        public JsonDeviceAnalysisConfigProvider(string configPath = "devicesettings.json")
        {
            _configPath = configPath;

            if (!File.Exists(_configPath))
            {
                throw new FileNotFoundException($"Configuration file not found: {_configPath}");
            }

            _configuration = new ConfigurationBuilder()
                .SetBasePath(Directory.GetCurrentDirectory())
                .AddJsonFile(_configPath, optional: false, reloadOnChange: true)
                .Build();
        }
    }
}

```

```

        LoadConfiguration();
    }

    private void LoadConfiguration()
    {
        AnalysisConfig = _configuration.GetSection("DeviceAnalysis")
            .Get<DeviceAnalysisConfig>() ?? throw new InvalidOperationException("Missing
DeviceAnalysis config");

        ProtocolAnalyzerConfig = _configuration.GetSection("ProtocolAnalyzer")
            .Get<ProtocolAnalyzerConfig>() ?? throw new InvalidOperationException("Missing
ProtocolAnalyzer config");

        DeviceSignatures = _configuration.GetSection("DeviceSignatures")
            .Get<Dictionary<string, DeviceSignatureConfig>>() ?? new Dictionary<string,
DeviceSignatureConfig>();

        VendorProfiles = _configuration.GetSection("VendorProfiles")
            .Get<Dictionary<string, VendorProfileConfig>>() ?? new Dictionary<string,
VendorProfileConfig>();

        DeviceModels = _configuration.GetSection("DeviceModels")
            .Get<Dictionary<string, Dictionary<string, List<string>>>>() ?? new();

        WebSoftwareSignatures =
        _configuration.GetSection("ContentAnalysis:WebSoftwareSignatures")
            .Get<Dictionary<string, SoftwareSignature>>() ?? new Dictionary<string,
SoftwareSignature>();

        TechnologySignatures =
        _configuration.GetSection("ContentAnalysis:TechnologySignatures")
            .Get<Dictionary<string, TechnologySignature>>() ?? new Dictionary<string,
TechnologySignature>();

        VulnerabilitySignatures =
        _configuration.GetSection("ContentAnalysis:VulnerabilitySignatures")
            .Get<Dictionary<string, VulnerabilitySignature>>() ?? new Dictionary<string,
VulnerabilitySignature>();

        DeviceDetectionSettings = _configuration.GetSection("DeviceDetection")
            .Get<DeviceDetectionSettings>() ?? new DeviceDetectionSettings();
    }

    public void ReloadConfiguration()
    {
        _configuration.Reload();
        LoadConfiguration();
    }

    public bool TryGetDeviceSignature(string deviceType, out DeviceSignatureConfig signature)
    {
        return DeviceSignatures.TryGetValue(deviceType, out signature);
    }

```

```

    }

    public bool TryGetVendorProfile(string vendor, out VendorProfileConfig profile)
    {
        return VendorProfiles.TryGetValue(vendor, out profile);
    }
}
} Code using System.Threading;
using System.Threading.Tasks;
using SOC.Agent.Models;

namespace SOC.Agent.Interfaces
{
    public interface IDeviceSignatureMatcher
    {
        Task<DeviceSignatureMatchResult> MatchSignaturesAsync(AnalysisResult analysisResult,
CancellationTokens ct);
        Task<DeviceSignatureMatchResult> MatchByPatternsAsync(string text, int port,
CancellationTokens ct);
        Task<DeviceSignatureMatchResult> MatchByHeadersAsync(List<HttpHeader> headers, int
port, CancellationTokens ct);
        Task<List<DeviceSignatureMatchResult>> MatchAllAsync(AnalysisResult analysisResult,
CancellationTokens ct);
        float CalculateConfidence(string text, DeviceSignatureConfig signature);
        bool IsSignatureMatch(string text, DeviceSignatureConfig signature);
    }
}
Code using System;
using System.Collections.Concurrent;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text.RegularExpressions;
using System.Threading;
using System.Threading.Tasks;
using Microsoft.Extensions.Logging;
using SOC.Agent.Interfaces;
using SOC.Agent.Models;

namespace SOC.Agent.Services.Matchers
{
    public class DeviceSignatureMatcher : IDeviceSignatureMatcher, IDisposable
    {
        private readonly ILogger<DeviceSignatureMatcher> _logger;
        private readonly IDeviceAnalysisConfigProvider _configProvider;
        private readonly CancellationTokenSource _internalCts = new CancellationTokenSource();
        private readonly ConcurrentDictionary<string, Regex> _regexCache = new();
        private readonly RegexOptions _regexOptions = RegexOptions.Compiled |
RegexOptions.IgnoreCase | RegexOptions.CultureInvariant;
        private readonly TimeSpan _regexTimeout = TimeSpan.FromMilliseconds(500);
        private bool _disposed;

        private static readonly HashSet<string> _privilegedPorts = new()

```

```

{
    22, 23, 25, 53, 80, 110, 143, 443, 465, 587, 636, 993, 995,
    161, 162, 179, 389, 443, 514, 587, 636, 993, 995, 1194, 3306,
    3389, 5432, 5900, 6379, 8080, 8443
};

public DeviceSignatureMatcher(
    ILogger<DeviceSignatureMatcher> logger,
    IDeviceAnalysisConfigProvider configProvider)
{
    _logger = logger ?? throw new ArgumentNullException(nameof(logger));
    _configProvider = configProvider ?? throw new
ArgumentNullException(nameof(configProvider));
}

#region Public Methods

    public async Task<DeviceSignatureMatchResult> MatchSignaturesAsync(AnalysisResult
analysisResult, CancellationToken ct)
    {
        if (analysisResult == null)
            throw new ArgumentNullException(nameof(analysisResult));

        var result = new DeviceSignatureMatchResult
        {
            Timestamp = DateTime.UtcNow
        };

        if (string.IsNullOrEmpty(analysisResult.Banner) &&
            (analysisResult.HttpHeaders == null || !analysisResult.HttpHeaders.Any()) &&
            string.IsNullOrEmpty(analysisResult.HtmlContent))
        {
            _logger.LogInformation("No data available for signature matching");
            return result;
        }

        try
        {
            // Build combined text for matching
            var combinedText = BuildCombinedText(analysisResult);

            _logger.LogDebug("Matching against combined text of length {Length}",
combinedText.Length);

            // Create linked token source with timeout
            using var linkedCts = CancellationTokenSource.CreateLinkedTokenSource(ct,
_internalCts.Token);

            linkedCts.CancelAfter(TimeSpan.FromSeconds(_configProvider.AnalysisConfig.ScanTime
outSeconds));

            var matches = await MatchSignaturesParallelAsync(combinedText, analysisResult.Port,
linkedCts.Token);

```

```

        if (matches.Any())
        {
            ProcessMatches(matches, result, combinedText);
        }
        else
        {
            _logger.LogInformation("No device signatures matched for port {Port}",
analysisResult.Port);
        }

        // Try matching by headers if available
        if (analysisResult.HttpHeaders != null && analysisResult.HttpHeaders.Any())
        {
            var headerMatches = await MatchByHeadersAsync(analysisResult.HttpHeaders,
analysisResult.Port, ct);
            if (headerMatches.Confidence > result.Confidence)
            {
                result = headerMatches;
            }
        }
    }
    catch (OperationCanceledException)
    {
        _logger.LogWarning("Device signature matching timed out for {IP}:{Port}",
analysisResult.IpAddress, analysisResult.Port);
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, "Device signature matching failed for {IP}:{Port}",
analysisResult.IpAddress, analysisResult.Port);
    }

    return result;
}

```

```

public async Task<DeviceSignatureMatchResult> MatchByPatternsAsync(string text, int port,
Cancellation token ct)
{
    var result = new DeviceSignatureMatchResult();

    if (string.IsNullOrEmpty(text))
        return result;

    try
    {
        var matches = await MatchSignaturesParallelAsync(text, port, ct);
        if (matches.Any())
        {
            ProcessMatches(matches, result, text);
        }
    }
    catch (Exception ex)
    {
    }
}

```

```

    {
        _logger.LogError(ex, "Pattern matching failed");
    }

    return result;
}

public async Task<DeviceSignatureMatchResult> MatchByHeadersAsync(List<HttpHeader>
headers, int port, CancellationToken ct)
{
    var result = new DeviceSignatureMatchResult();

    if (headers == null || !headers.Any())
        return result;

    try
    {
        var text = string.Join("\n", headers.Select(h => $"{h.Key}: {h.Value}"));

        var matches = await MatchSignaturesParallelAsync(text, port, ct);
        if (matches.Any())
        {
            ProcessMatches(matches, result, text);
        }
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, "Header matching failed");
    }

    return result;
}

public async Task<List<DeviceSignatureMatchResult>> MatchAllAsync(AnalysisResult
analysisResult, CancellationToken ct)
{
    var results = new List<DeviceSignatureMatchResult>();

    if (analysisResult == null)
        return results;

    var combinedText = BuildCombinedText(analysisResult);

    try
    {
        using var linkedCts = CancellationTokenSource.CreateLinkedTokenSource(ct,
_internalCts.Token);

        linkedCts.CancelAfter(TimeSpan.FromSeconds(_configProvider.AnalysisConfig.ScanTime
outSeconds));

        var matches = await Task.Run(() =>
        {

```

```

var foundMatches = new ConcurrentBag<(string Type, DeviceSignatureConfig
Signature, float Confidence)>();

var options = new ParallelOptions
{
    CancellationToken = linkedCts.Token,
    MaxDegreeOfParallelism = Environment.ProcessorCount * 2
};

Parallel.ForEach(_configProvider.DeviceSignatures, options, item =>
{
    try
    {
        var signature = item.Value;
        if (!signature.Ports.Contains(analysisResult.Port))
            return;

        var confidence = CalculateConfidence(combinedText, signature);
        if (confidence > 0.3f) // Minimum threshold
        {
            foundMatches.Add((item.Key, signature, confidence));
        }
    }
    catch (OperationCanceledException)
    {
        throw;
    }
    catch (Exception ex)
    {
        _logger.LogWarning(ex, "Error matching signature {DeviceType}", item.Key);
    }
});

return foundMatches.ToList();
}, linkedCts.Token);

foreach (var match in matches.OrderByDescending(m => m.Confidence))
{
    var result = new DeviceSignatureMatchResult
    {
        DeviceType = match.Type,
        Confidence = match.Confidence,
        Categories = match.Signature.Categories?.ToList() ?? new List<string>(),
        Vendors = match.Signature.Vendors?.ToList() ?? new List<string>(),
        Timestamp = DateTime.UtcNow
    };

    // Add matched patterns
    foreach (var pattern in match.Signature.Patterns ?? Array.Empty<string>())
    {
        if (!string.IsNullOrEmpty(pattern) &&
            Regex.IsMatch(combinedText, pattern, RegexOptions.IgnoreCase))
        {

```

```

        result.MatchedPatterns.Add(new MatchedPattern
        {
            Pattern = pattern,
            Source = "PatternMatch",
            Confidence = 0.8f,
            MatchedText = GetMatchedText(combinedText, pattern)
        });
    }
}

results.Add(result);
}
}
catch (Exception ex)
{
    _logger.LogError(ex, "MatchAll failed");
}

return results.OrderByDescending(r => r.Confidence).ToList();
}

public float CalculateConfidence(string text, DeviceSignatureConfig signature)
{
    if (string.IsNullOrEmpty(text) || signature == null || signature.Patterns == null)
        return 0;

    try
    {
        var matchingPatterns = new List<string>();
        foreach (var pattern in signature.Patterns)
        {
            if (!string.IsNullOrEmpty(pattern) &&
                Regex.IsMatch(text, pattern, RegexOptions.IgnoreCase))
            {
                matchingPatterns.Add(pattern);
            }
        }

        if (!matchingPatterns.Any())
            return 0;

        // Pattern confidence
        var patternMatchCount = matchingPatterns.Count;
        var totalPatterns = signature.Patterns.Length;
        var patternRatio = (float)patternMatchCount / totalPatterns;
        var patternScore = matchingPatterns.Average(p => p.Length) / 100f;

        // Port confidence
        var portScore = signature.Ports?.Any(p => _privilegedPorts.Contains(p)) == true ? 0.7f :

1.0f;

        // Category confidence (some categories are more specific)
        var categoryScore = signature.Categories?.Contains("Networking") == true ? 1.0f : 0.9f;

```

```
        // Weighted confidence
        var confidence = (patternRatio * 0.5f + patternScore * 0.3f + portScore * 0.1f +
categoryScore * 0.1f);
```

```
        return Math.Min(Math.Max(confidence, 0), 1.0f);
    }
    catch (Exception ex)
    {
        _logger.LogDebug(ex, "Confidence calculation failed");
        return 0;
    }
}
```

```
public bool IsSignatureMatch(string text, DeviceSignatureConfig signature)
{
    if (string.IsNullOrEmpty(text) || signature == null || signature.Patterns == null)
        return false;

    foreach (var pattern in signature.Patterns)
    {
        if (!string.IsNullOrEmpty(pattern) &&
            Regex.IsMatch(text, pattern, RegexOptions.IgnoreCase))
        {
            return true;
        }
    }

    return false;
}
```

```
#endregion
```

```
#region Private Methods
```

```
private string BuildCombinedText(AnalysisResult analysisResult)
{
    var parts = new List<string>();

    if (!string.IsNullOrEmpty(analysisResult.Banner))
        parts.Add(analysisResult.Banner);

    if (!string.IsNullOrEmpty(analysisResult.HtmlContent))
        parts.Add(analysisResult.HtmlContent);

    if (analysisResult.HttpHeaders != null && analysisResult.HttpHeaders.Any())
    {
        parts.Add(string.Join("\n", analysisResult.HttpHeaders
            .Where(h => !string.IsNullOrEmpty(h.Key) && !string.IsNullOrEmpty(h.Value))
            .Select(h => $"{h.Key}: {h.Value}")));
    }

    return string.Join("\n", parts).ToLowerInvariant();
}
```

```

}

private async Task<List<(string Type, DeviceSignatureConfig Signature, float Confidence)>>
    MatchSignaturesParallelAsync(string text, int port, CancellationToken ct)
{
    return await Task.Run(() =>
    {
        var matches = new ConcurrentBag<(string Type, DeviceSignatureConfig Signature, float
Confidence)>();

        var options = new ParallelOptions
        {
            CancellationToken = ct,
            MaxDegreeOfParallelism = Environment.ProcessorCount * 2
        };

        Parallel.ForEach(_configProvider.DeviceSignatures, options, item =>
        {
            try
            {
                var signature = item.Value;

                // Check if port matches
                if (!signature.Ports.Contains(port))
                    return;

                // Check for pattern matches
                if (!IsSignatureMatch(text, signature))
                    return;

                var confidence = CalculateConfidence(text, signature);
                if (confidence > 0.3f) // Minimum threshold
                {
                    matches.Add((item.Key, signature, confidence));
                }
            }
            catch (OperationCanceledException)
            {
                throw;
            }
            catch (Exception ex)
            {
                _logger.LogWarning(ex, "Error matching signature {DeviceType}", item.Key);
            }
        });

        return matches.ToList();
    }, ct);
}

private void ProcessMatches(
    List<(string Type, DeviceSignatureConfig Signature, float Confidence)> matches,
    DeviceSignatureMatchResult result,

```

```

string text)
{
    // Find best match
    var bestMatch = matches
        .OrderByDescending(m => m.Confidence)
        .ThenByDescending(m => m.Signature.Patterns?.Length ?? 0)
        .First();

    result.DeviceType = bestMatch.Type;
    result.Confidence = bestMatch.Confidence;
    result.Categories = bestMatch.Signature.Categories?.ToList() ?? new List<string>();
    result.Vendors = bestMatch.Signature.Vendors?.ToList() ?? new List<string>();

    // Add matched patterns
    if (bestMatch.Signature.Patterns != null)
    {
        foreach (var pattern in bestMatch.Signature.Patterns)
        {
            if (!string.IsNullOrEmpty(pattern) &&
                Regex.IsMatch(text, pattern, RegexOptions.IgnoreCase))
            {
                result.MatchedPatterns.Add(new MatchedPattern
                {
                    Pattern = pattern,
                    Source = "PatternMatch",
                    Confidence = 0.8f,
                    MatchedText = GetMatchedText(text, pattern)
                });
            }
        }
    }

    // Detect models
    if (bestMatch.Signature.Models != null)
    {
        foreach (var model in bestMatch.Signature.Models)
        {
            if (text.Contains(model.ToLowerInvariant()))
            {
                result.DeviceModel = model;
                break;
            }
        }
    }

    // Detect firmware
    if (bestMatch.Signature.FirmwarePatterns != null)
    {
        foreach (var pattern in bestMatch.Signature.FirmwarePatterns)
        {
            var match = Regex.Match(text, pattern, RegexOptions.IgnoreCase);
            if (match.Success && match.Groups.Count > 1)
            {

```

```

        result.FirmwareVersion = match.Groups[1].Value;
        break;
    }
}

// Detect vulnerabilities
if (bestMatch.Signature.Vulnerabilities != null)
{
    foreach (var vuln in bestMatch.Signature.Vulnerabilities)
    {
        result.DetectedVulnerabilities.Add(new DetectedVulnerability
        {
            CVE = vuln.CVE,
            Description = vuln.Description,
            Severity = vuln.Severity,
            CVSS = vuln.CVSS,
            IsExploitable = vuln.IsExploitable,
            Confidence = 0.7f,
            AffectedComponent = bestMatch.Type
        });
    }
}

// Add metadata
result.Metadata["MatchedSignatures"] = matches.Count;
result.Metadata["MatchScore"] = bestMatch.Confidence;
result.Metadata["Timestamp"] = DateTime.UtcNow;

if (bestMatch.Signature.Metadata != null)
{
    foreach (var kvp in bestMatch.Signature.Metadata)
    {
        result.Metadata[kvp.Key] = kvp.Value;
    }
}

private string GetMatchedText(string text, string pattern)
{
    try
    {
        var match = Regex.Match(text, pattern, RegexOptions.IgnoreCase);
        if (match.Success)
        {
            return match.Value.Length > 100 ? match.Value[..100] + "..." : match.Value;
        }
    }
    catch (Exception ex)
    {
        _logger.LogDebug(ex, "Error extracting matched text");
    }
    return string.Empty;
}

```

```

    }

    private Regex GetCachedRegex(string pattern)
    {
        return _regexCache.GetOrAdd(pattern,
            p => new Regex(p, _regexOptions, _regexTimeout));
    }

    #endregion

    #region IDisposable

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing)
    {
        if (!_disposed)
        {
            if (disposing)
            {
                _internalCts?.Cancel();
                _internalCts?.Dispose();
                _regexCache.Clear();
            }
            _disposed = true;
        }
    }

    #endregion
}
} Code using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
using Microsoft.Extensions.Logging;
using SOC.Agent.Interfaces;
using SOC.Agent.Models;

namespace SOC.Agent.Services
{
    public class DeviceAnalyzer : IDeviceAnalyzer
    {
        private readonly ILogger<DeviceAnalyzer> _logger;
        private readonly IDeviceSignatureMatcher _signatureMatcher;
        private readonly IContentAnalyzer _contentAnalyzer;
        private readonly IDeviceAnalysisConfigProvider _configProvider;

        public DeviceAnalyzer(

```

```

ILogger<DeviceAnalyzer> logger,
IDeviceSignatureMatcher signatureMatcher,
IContentAnalyzer contentAnalyzer,
IDeviceAnalysisConfigProvider configProvider)
{
    _logger = logger ?? throw new ArgumentNullException(nameof(logger));
    _signatureMatcher = signatureMatcher ?? throw new
ArgumentNullException(nameof(signatureMatcher));
    _contentAnalyzer = contentAnalyzer ?? throw new
ArgumentNullException(nameof(contentAnalyzer));
    _configProvider = configProvider ?? throw new
ArgumentNullException(nameof(configProvider));
}

public async Task<DeviceAnalysisResult> AnalyzeDeviceAsync(
    string ipAddress,
    int port,
    CancellationToken ct)
{
    var result = new DeviceAnalysisResult
    {
        IpAddress = ipAddress,
        Port = port,
        Timestamp = DateTime.UtcNow
    };

    try
    {
        // Step 1: Gather initial data
        var analysisResult = await GatherDeviceDataAsync(ipAddress, port, ct);

        // Step 2: Match signatures
        var signatureResult = await _signatureMatcher.MatchSignaturesAsync(analysisResult,
ct);

        result.DeviceType = signatureResult.DeviceType;
        result.Confidence = signatureResult.Confidence;
        result.Categories = signatureResult.Categories;
        result.Evidence = signatureResult.MatchedPatterns
            .Select(p => p.Pattern ?? string.Empty)
            .ToList();
        result.Metadata["SignatureMatch"] = signatureResult;

        // Step 3: Analyze content if HTTP
        if (port == 80 || port == 443 || port == 8080 || port == 8443)
        {
            var contentResult = await _contentAnalyzer.AnalyzeContentAsync(
                ipAddress, port, analysisResult.HttpHeaders, ct);

            result.ContentAnalysis = contentResult;
            result.Metadata["ContentAnalysis"] = contentResult;
        }
    }
}

```

```

// Step 4: Try to identify model
if (signatureResult.DeviceModel != null)
{
    result.Model = signatureResult.DeviceModel;
}
else if (signatureResult.Fingerprint?.Model != null)
{
    result.Model = signatureResult.Fingerprint.Model;
}

// Step 5: Try to identify vendor
if (signatureResult.Vendors.Any())
{
    result.Vendor = signatureResult.Vendors.First();
}
else if (_configProvider.VendorProfiles != null)
{
    foreach (var vendor in _configProvider.VendorProfiles.Keys)
    {
        if (analysisResult.Banner?.Contains(vendor, StringComparison.OrdinalIgnoreCase)
== true)
        {
            result.Vendor = vendor;
            break;
        }
    }
}

// Step 6: Detect vulnerabilities
if (signatureResult.DetectedVulnerabilities.Any())
{
    result.Vulnerabilities = signatureResult.DetectedVulnerabilities
        .Select(v => new DetectedVulnerability
        {
            CVE = v.CVE,
            Description = v.Description,
            Severity = v.Severity,
            Confidence = v.Confidence
        })
        .ToList();
}

_logger.LogInformation(
    "Device analysis complete for {IP}:{Port} - {DeviceType} (Confidence:
{Confidence:P2})",
    ipAddress, port, result.DeviceType ?? "Unknown", result.Confidence);
}
catch (Exception ex)
{
    _logger.LogError(ex, "Device analysis failed for {IP}:{Port}", ipAddress, port);
    throw;
}

```

```
    return result;
}
```

```
public async Task<DeviceAnalysisResult> AnalyzeDeviceAsync(
    string ipAddress,
    int port,
    string banner,
    List<HttpHeader>? headers,
    CancellationToken ct)
{
    var analysisResult = new AnalysisResult
    {
        IpAddress = ipAddress,
        Port = port,
        Banner = banner,
        HttpHeaders = headers
    };

    return await AnalyzeDeviceAsync(ipAddress, port, analysisResult, ct);
}
```

```
public async Task<DeviceAnalysisResult> AnalyzeDeviceAsync(
    string ipAddress,
    int port,
    AnalysisResult analysisResult,
    CancellationToken ct)
{
    var result = new DeviceAnalysisResult
    {
        IpAddress = ipAddress,
        Port = port,
        Timestamp = DateTime.UtcNow
    };

    try
    {
        // Match signatures
        var signatureResult = await _signatureMatcher.MatchSignaturesAsync(analysisResult,
ct);

        result.DeviceType = signatureResult.DeviceType;
        result.Confidence = signatureResult.Confidence;
        result.Categories = signatureResult.Categories;
        result.Evidence = signatureResult.MatchedPatterns
            .Select(p => p.Pattern ?? string.Empty)
            .ToList();
        result.Metadata["SignatureMatch"] = signatureResult;

        // Detect vulnerabilities
        if (signatureResult.DetectedVulnerabilities.Any())
        {
            result.Vulnerabilities = signatureResult.DetectedVulnerabilities
                .Select(v => new DetectedVulnerability
```

```

        {
            CVE = v.CVE,
            Description = v.Description,
            Severity = v.Severity,
            Confidence = v.Confidence
        })
        .ToList();
    }

    return result;
}
catch (Exception ex)
{
    _logger.LogError(ex, "Device analysis failed for {IP}:{Port}", ipAddress, port);
    throw;
}
}

```

```

public async Task<List<DeviceAnalysisResult>> AnalyzeNetworkRangeAsync(
    string network,
    int[] ports,
    CancellationToken ct)
{
    var results = new List<DeviceAnalysisResult>();
    var tasks = new List<Task>();

    // Parse network range (simplified)
    var ips = ParseNetworkRange(network);

    var semaphore = new
SemaphoreSlim(_configProvider.AnalysisConfig.MaxConcurrentScans);

    foreach (var ip in ips)
    {
        foreach (var port in ports)
        {
            tasks.Add(Task.Run(async () =>
            {
                await semaphore.WaitAsync(ct);
                try
                {
                    var result = await AnalyzeDeviceAsync(ip, port, ct);
                    lock (results)
                    {
                        results.Add(result);
                    }
                }
            }
            catch (Exception ex)
            {
                _logger.LogWarning(ex, "Failed to analyze {IP}:{Port}", ip, port);
            }
            finally
            {

```

```

        semaphore.Release();
    }
    }, ct));
}

await Task.WhenAll(tasks);
return results.OrderBy(r => r.IpAddress).ToList();
}

private async Task<AnalysisResult> GatherDeviceDataAsync(
    string ipAddress,
    int port,
    CancellationToken ct)
{
    // This would connect to the device and gather banner, headers, etc.
    // Simplified for example
    return new AnalysisResult
    {
        IpAddress = ipAddress,
        Port = port,
        Timestamp = DateTime.UtcNow
    };
}

private List<string> ParseNetworkRange(string network)
{
    // Simplified network range parsing
    // In production, use proper IP range parsing
    var ips = new List<string>();

    // Example: "192.168.1.0/24" -> 192.168.1.1 to 192.168.1.254
    // This is a placeholder - implement proper parsing
    for (int i = 1; i < 255; i++)
    {
        ips.Add($"192.168.1.{i}");
    }

    return ips;
}
}

Code using System.Collections.Generic;
using System.Threading;
using System.Threading.Tasks;
using SOC.Agent.Models;

namespace SOC.Agent.Interfaces
{
    public interface IDeviceAnalyzer
    {
        Task<DeviceAnalysisResult> AnalyzeDeviceAsync(
            string ipAddress,

```

```

    int port,
    CancellationToken ct);

Task<DeviceAnalysisResult> AnalyzeDeviceAsync(
    string ipAddress,
    int port,
    string banner,
    List<HttpHeader>? headers,
    CancellationToken ct);

Task<DeviceAnalysisResult> AnalyzeDeviceAsync(
    string ipAddress,
    int port,
    AnalysisResult analysisResult,
    CancellationToken ct);

Task<List<DeviceAnalysisResult>> AnalyzeNetworkRangeAsync(
    string network,
    int[] ports,
    CancellationToken ct);
}
} Code using System;
using System.Collections.Generic;

namespace SOC.Agent.Models
{
    public class DeviceAnalysisResult
    {
        public string? IpAddress { get; set; }
        public int Port { get; set; }
        public string? DeviceType { get; set; }
        public string? Vendor { get; set; }
        public string? Model { get; set; }
        public string? OperatingSystem { get; set; }
        public string? Category { get; set; }
        public float Confidence { get; set; }
        public List<string> Categories { get; set; } = new();
        public List<string> Evidence { get; set; } = new();
        public List<DetectedVulnerability> Vulnerabilities { get; set; } = new();
        public WebContentAnalysisResult? ContentAnalysis { get; set; }
        public Dictionary<string, object> Metadata { get; set; } = new();
        public DateTime Timestamp { get; set; } = DateTime.UtcNow;
    }
} config

```

Code <https://pastebin.com/abp5XA8U> Feel free to ask questions about project...Duck and I are ready to give you all the answers you need :)

HerrWinston - July 4, 2026, 11:31 am

Wow, it is insane to me that you just post stuff like this because you want to share something HQ.

I truly appreciate every thread you make, there is so much to learn from reading your code and seeing whats possible!

This is true legends high quality and a perfect start into this July contest!

itcorp - July 4, 2026, 12:17 pm

(Jul 4, 2026 11:31 AM)HerrWinston Wrote: [\[CITE\]](#) Wow, it is insane to me that you just post stuff like this because you want to share something HQ.

I truly appreciate every thread you make, there is so much to learn from reading your code and seeing whats possible!

This is true legends high quality and a perfect start into this July contest!

Thank you for the kind words, I'm glad the post was informative for you

NoodleBot™ - July 6, 2026, 11:42 pm

This is actually kind of amazing. I was thinking of something similar but for another use.

itcorp - July 10, 2026, 5:11 pm

(Jul 6, 2026 11:42 PM)NoodleBot™ Wrote: [\[CITE\]](#) This is actually kind of amazing. I was thinking of something similar but for another use.

thank you for the kind words, I'm available for any questions, but I'm also interested in your view of the whole story and idea

NoodleBot™ - July 10, 2026, 10:21 pm

(Jul 10, 2026 05:11 PM)itcorp Wrote: [\[CITE\]](#) thank you for the kind words, I'm available for any questions, but I'm also interested in your view of the whole story and idea

Oops I think you posted several things in a single post, a similar thread, yet somehow I commented wrong thread lol. I'll take a look after to see which of your posts I really meant that for.

But this looks pretty slick too

Edit: maybe I gotta read everything again when I'm not braindead tired. This is the post lol

itcorp - July 11, 2026, 12:36 pm

[Video: <https://www.youtube.com/watch?v=Bpkm-2Fkqs4>]

CL1B - July 13, 2026, 6:41 pm

Woah, I have to read this later