Tanium Sensor App Config Dot Net Map

```
# Tanium Sensor: App Config Dot Net Runtime
# Returns tab-delimited rows: ConfigFile | AppName | RuntimeVersion | RuntimeSKU | Source |
Publisher
# Deduped via HashSet — each config file parsed exactly once
#
# NOTE: Configure this sensor's Result Type as "Multiple Text Columns" (or an
# equivalent Column Set) with 6 columns in this order: ConfigFile, AppName,
# RuntimeVersion, RuntimeSKU, Source, Publisher — otherwise the tab-delimited
# row will render as a single blob column instead of sortable fields.
#
# Publisher is read from the embedded Authenticode signer on the companion
# .exe (only applies to *.exe.config rows — web.config/app.config rows without
# a 1:1 exe have no single binary to check and are left blank). This is a
# fast local identity read (no chain validation, no revocation check, no
# network call) — a triage heuristic, not a verified trust decision. Values:
#   "Microsoft"          - signer CN contains Microsoft; likely patchable via
#                          Windows/SQL Update
#   "<Signer CN>"        - signed by a named third-party vendor; needs a
#                          vendor upgrade
#   "Unsigned/Unknown"   - no embedded signature found; likely custom/internal app
#
# In addition to app-declared rows, this sensor also emits HOST-level rows
# (ConfigFile = "HOST") reporting what's actually installed on the machine —
# .NET Framework 4.x/3.5 (registry) and every installed .NET Core/5+ and
# ASP.NET Core shared runtime (filesystem). Source for these rows is prefixed
# "InstalledRuntime:" so they can be filtered separately from app-declared
# rows, and are what you compare app declarations against to find the actual
# gap. Because these rows are always present, the "No runtime declarations
# found" fallback below will now rarely fire — check Source for
# "InstalledRuntime:" vs everything else, not $results.Count, to know whether
# any app-level declarations were found.

# --- Debug toggle -----------------------------------------------------------
```

```
# Flip to $true for manual/console testing to see elapsed time and file count
# on stderr (does not pollute the Write-Output stream Tanium parses as
# results). Leave $false for production deployment.
$Debug = $false
# ------------------------------------------------------------------------------

$results        = [System.Collections.Generic.List[string]]::new()
$processedFiles = [System.Collections.Generic.HashSet[string]]::new(
                    [System.StringComparer]::OrdinalIgnoreCase)
$publisherCache = [System.Collections.Generic.Dictionary[string,string]]::new(
                    [System.StringComparer]::OrdinalIgnoreCase)
$sw             = [System.Diagnostics.Stopwatch]::StartNew()
$timeCap        = 25    # seconds — exits cleanly before sensor timeout
$maxHits        = 500

$searchRoots = @(
    "$env:SystemDrive\inetpub",
    "$env:ProgramFiles",
    "${env:ProgramFiles(x86)}",
    "$env:SystemDrive\apps",
    "$env:SystemDrive\websites",
    "$env:SystemDrive\services"
) | Where-Object { $_ -and (Test-Path $_) }

$skipPattern    = '\\(' + (@(
    # OS / runtime internals
    'Windows'
    'Microsoft\.NET'
    'dotnet\\host'
    '\$Recycle'
    'Tanium'
    'node_modules'
    '\.git'
    # Dev tooling — not a deployed app dependency
    'Microsoft SQL Server Management Studio[^\\]*'
    'Microsoft Visual Studio\\Installer'
    'Microsoft Visual Studio\\Shared'
    'Windows Kits'
    # SQL Server setup/installer payloads — cached installers, not the running engine
    'Setup Bootstrap\\Update Cache'
```

```
    'Setup Bootstrap\\SQL\d+'
    # SCCM site-server staging internals — not live services
    'EasySetupPayload'
    'CMUStaging'
    'cd\.latest'
) -join '|') + ')\\'
$configExtensions = @('.config')   # covers web.config, app.config, *.exe.config
$configFileNames  = @('web.config', 'app.config')

# --- Safe, non-lazy recursive walk ----------------------------------------
# Directory.EnumerateFiles()/EnumerateDirectories() are lazy: exceptions
# (e.g. UnauthorizedAccessException on a locked-down ACL folder) are thrown
# mid-foreach, outside any try/catch wrapped around the initial call. That
# silently aborts the entire enumeration for that root. GetFiles()/
# GetDirectories() are eager — wrapping each call per-directory lets us
# prune a single bad branch instead of losing the whole tree.
function Get-ConfigFilesSafe {
    param(
        [string]$Root,
        [string]$SkipPattern,
        [System.Diagnostics.Stopwatch]$Stopwatch,
        [int]$TimeCapSeconds
    )

    $stack = [System.Collections.Generic.Stack[string]]::new()
    $stack.Push($Root)

    while ($stack.Count -gt 0) {
        if ($Stopwatch.Elapsed.TotalSeconds -ge $TimeCapSeconds) { return }

        $dir = $stack.Pop()

        # Queue subdirectories (skip known noise up front to save I/O)
        try {
            $subDirs = [System.IO.Directory]::GetDirectories($dir)
        } catch { $subDirs = @() }

        foreach ($d in $subDirs) {
            if ($d -notmatch $SkipPattern) { $stack.Push($d) }
        }
```

```
        # Get candidate files in this directory only
        try {
            $files = [System.IO.Directory]::GetFiles($dir, '*.config')
        } catch { continue }

        foreach ($f in $files) {
            if ($f -match $SkipPattern) { continue }

            $name = [System.IO.Path]::GetFileName($f)
            $isMatch =
                $configFileNames -contains $name.ToLowerInvariant() -or
                $f.EndsWith('.exe.config', [System.StringComparison]::OrdinalIgnoreCase)

            if ($isMatch) { $f }   # yield
        }
    }
}
# ------------------------------------------------------------------------

# --- Publisher tagging ----------------------------------------------------
# Only meaningful for *.exe.config (the companion .exe is a real signable
# binary). web.config/app.config sit next to a whole app folder, not a single
# exe, so there's nothing specific to check the signature of.
#
# Uses X509Certificate::CreateFromSignedFile() rather than
# Get-AuthenticodeSignature. That cmdlet's chain validation can perform an
# online CRL/OCSP revocation check per file, which can hang for several
# seconds with no bound this script controls, on a host with no outbound
# access. CreateFromSignedFile reads the embedded signer certificate straight
# off disk — no network call, no chain build, no revocation check, so the
# time cap stays enforceable. This is a fast identity heuristic, not a
# validated trust decision — fine for inventory/triage, not a security check.
function Get-PublisherTag {
    param(
        [string]$ConfigPath,
        [System.Collections.Generic.Dictionary[string,string]]$Cache
    )

    if ($ConfigPath -notmatch '\.exe\.config$') { return '' }
```

```
    $exePath = $ConfigPath.Substring(0, $ConfigPath.Length - '.config'.Length)

    if ($Cache.ContainsKey($exePath)) { return $Cache[$exePath] }

    $tag = 'Unsigned/Unknown'
    try {
        if (Test-Path -LiteralPath $exePath) {
            $cert =
[System.Security.Cryptography.X509Certificates.X509Certificate]::CreateFromSignedFile($exePath
)
            if ($cert -and $cert.Subject -match 'CN=("[^"]+"|[^,]+)') {
                $cn  = $matches[1].Trim('"')
                $tag = if ($cn -match 'Microsoft') { 'Microsoft' } else { $cn }
            }
        }
    } catch {
        # CreateFromSignedFile throws on unsigned files — leave as Unsigned/Unknown
    }

    $Cache[$exePath] = $tag
    return $tag
}
# ---------------------------------------------------------------------------

# --- Installed runtime inventory ---------------------------------------------
# Reports what's actually installed on the host, so results can be compared
# against what the app-config scan says apps *declare* they need — that
# comparison is the actual dependency-gap question for remediation.
#
# Deliberately filesystem/registry-only, no process spawn (e.g. no
# `dotnet --list-runtimes`) — avoids process-creation overhead and keeps this
# section fast and reliable regardless of PATH state on the host.
#
# Emits rows in the same 6-column shape, tagged via Source so they're
# filterable separately from app-declared rows:
#   Source = InstalledRuntime:NETFramework
#   Source = InstalledRuntime:Microsoft.NETCore.App        (.NET Core/5+ runtime)
#   Source = InstalledRuntime:Microsoft.AspNetCore.App      (ASP.NET Core runtime)
#   Source = InstalledRuntime:Microsoft.WindowsDesktop.App  (WPF/WinForms runtime, if present)
```

```
function Add-InstalledRuntimeRows {
    param([System.Collections.Generic.List[string]]$ResultsList)

    # .NET Framework 4.x — single latest version per machine, keyed off the
    # documented Release DWORD (see Microsoft's "How to: Determine which
    # .NET Framework versions are installed" release-id table).
    try {
        $ndpKey = 'HKLM:\SOFTWARE\Microsoft\NET Framework Setup\NDP\v4\Full'
        if (Test-Path $ndpKey) {
            $release  = (Get-ItemProperty -Path $ndpKey -Name Release -ErrorAction
Stop).Release
            $fxVersion = if     ($release -ge 533320) { '4.8.1' }
                         elseif ($release -ge 528040) { '4.8'   }
                         elseif ($release -ge 461808) { '4.7.2' }
                         elseif ($release -ge 461308) { '4.7.1' }
                         elseif ($release -ge 460798) { '4.7'   }
                         elseif ($release -ge 394802) { '4.6.2' }
                         elseif ($release -ge 394254) { '4.6.1' }
                         elseif ($release -ge 393295) { '4.6'   }
                         elseif ($release -ge 379893) { '4.5.2' }
                         elseif ($release -ge 378675) { '4.5.1' }
                         elseif ($release -ge 378389) { '4.5'   }
                         else                         { "Unknown" }

$ResultsList.Add("HOST`t`t$fxVersion`tRelease=$release`tInstalledRuntime:NETFramework`tMicroso
ft")
        }
    } catch { }

    # .NET Framework 3.5 (and by extension 2.0/3.0) — legacy apps still
    # sometimes require this as a separate, independently-installed feature.
    try {
        $legacyKey = 'HKLM:\SOFTWARE\Microsoft\NET Framework Setup\NDP\v3.5'
        if (Test-Path $legacyKey) {
            $installed = (Get-ItemProperty -Path $legacyKey -Name Install -ErrorAction
SilentlyContinue).Install
            if ($installed -eq 1) {
                $ResultsList.Add("HOST`t`t3.5`t`tInstalledRuntime:NETFramework`tMicrosoft")
            }
        }
```

```
    } catch { }

    # .NET Core / .NET 5+ shared runtimes — every installed runtime version
    # lives as its own folder under dotnet\shared\<RuntimeName>\<Version>.
    $dotnetRoots = @(
        "$env:ProgramFiles\dotnet\shared",
        "${env:ProgramFiles(x86)}\dotnet\shared"
    ) | Where-Object { $_ -and (Test-Path $_) }

    foreach ($sharedRoot in $dotnetRoots) {
        try {
            $runtimeDirs = [System.IO.Directory]::GetDirectories($sharedRoot)
        } catch { continue }

        foreach ($runtimeDir in $runtimeDirs) {
            $runtimeName = [System.IO.Path]::GetFileName($runtimeDir)

            try {
                $versionDirs = [System.IO.Directory]::GetDirectories($runtimeDir)
            } catch { continue }

            foreach ($vDir in $versionDirs) {
                $ver = [System.IO.Path]::GetFileName($vDir)
                $ResultsList.Add("HOST`t`t$ver`t`tInstalledRuntime:$runtimeName`tMicrosoft")
            }
        }
    }
}
# ---------------------------------------------------------------------------

:rootLoop foreach ($root in $searchRoots) {
    if ($sw.Elapsed.TotalSeconds -ge $timeCap -or $results.Count -ge $maxHits) { break }

    foreach ($fp in (Get-ConfigFilesSafe -Root $root -SkipPattern $skipPattern -Stopwatch $sw
-TimeCapSeconds $timeCap)) {

        if ($sw.Elapsed.TotalSeconds -ge $timeCap) { break rootLoop }
        if (-not $processedFiles.Add($fp))         { continue }   # dedup

        try {
```

```
            [xml]$xml = Get-Content -LiteralPath $fp -Raw -ErrorAction Stop
        } catch { continue }

        $app       = [System.IO.Path]::GetFileNameWithoutExtension($fp) -replace
'\.exe$|\.dll$', ''
        $publisher = Get-PublisherTag -ConfigPath $fp -Cache $publisherCache

        # <startup><supportedRuntime version="" sku="" />
        $xml.SelectNodes('//startup/supportedRuntime') | ForEach-Object {
            $v = $_.GetAttribute('version'); $s = $_.GetAttribute('sku')
            if ($v -or $s) { $results.Add("$fp`t$app`t$v`t$s`tsupportedRuntime`t$publisher") }
        }

        # <runtime><supportedRuntime> (alternate placement)
        $xml.SelectNodes('//runtime/supportedRuntime') | ForEach-Object {
            $v = $_.GetAttribute('version'); $s = $_.GetAttribute('sku')
            if ($v -or $s) {
$results.Add("$fp`t$app`t$v`t$s`tsupportedRuntime(runtime)`t$publisher") }
        }

        # SDK-style: <TargetFramework>net6.0</TargetFramework>
        $xml.SelectNodes('//*[local-name()="TargetFramework" or local-
name()="TargetFrameworks"]') |
        ForEach-Object {
            $tf = $_.InnerText.Trim()
            if ($tf) { $results.Add("$fp`t$app`t$tf`t`tTargetFramework`t$publisher") }
        }

        # web.config: <compilation targetFramework="4.7.2">
        $xml.SelectNodes('//compilation[@targetFramework]') | ForEach-Object {
            $tf = $_.GetAttribute('targetFramework')
            if ($tf) { $results.Add("$fp`t$app`t$tf`t`tweb.config/compilation`t$publisher") }
        }

        # web.config: <httpRuntime targetFramework="4.7.2">
        $xml.SelectNodes('//httpRuntime[@targetFramework]') | ForEach-Object {
            $tf = $_.GetAttribute('targetFramework')
            if ($tf) { $results.Add("$fp`t$app`t$tf`t`tweb.config/httpRuntime`t$publisher") }
        }
```

```
        if ($results.Count -ge $maxHits) { break rootLoop }
    }
}

# Host installed-runtime inventory — run regardless of whether the app-config
# scan above hit its time cap or maxHits, since this section is a handful of
# registry reads and directory listings (negligible cost) and is essential
# context even on a partial config scan.
Add-InstalledRuntimeRows -ResultsList $results

if ($Debug) {
    [Console]::Error.WriteLine(
        "[DEBUG] Elapsed: $([math]::Round($sw.Elapsed.TotalSeconds, 2))s | " +
        "Files processed: $($processedFiles.Count) | " +
        "Hits: $($results.Count) | " +
        "TimeCapHit: $($sw.Elapsed.TotalSeconds -ge $timeCap) | " +
        "MaxHitsHit: $($results.Count -ge $maxHits)"
    )
}

if ($results.Count -eq 0) {
    Write-Output "No runtime declarations found`t`t`t`t`t"
} else {
    $results | ForEach-Object { Write-Output $_ }
}
```

Revision #1
Created 21 July 2026 23:21:28 by Christian Kaba
Updated 21 July 2026 23:23:33 by Christian Kaba