BinSkim: A Windows PE (Portable Executable) Security Correctness Tool.

Contents

[Description of Tool 1](#_Toc468456534)

[Source and Drop Location 2](#_Toc468456535)

[Running BinSkim from the Command-Line 2](#_Toc468456536)

[Quick Start 2](#_Toc468456537)

[Command-Line Argument Reference 2](#_Toc468456538)

[Help command 2](#_Toc468456539)

[Analyze Command 2](#_Toc468456540)

[Project Files 6](#_Toc468456541)

[CommandLine Property 7](#_Toc468456542)

[Authoring Custom Loggers 7](#_Toc468456543)

[IMessageLogger 8](#_Toc468456544)

[CSV Custom Logger Example 8](#_Toc468456545)

[FAQs 9](#_Toc468456546)

[How does ModernCop raise its ASTs? Can I consume them? 9](#_Toc468456547)

[General Notes 9](#_Toc468456548)

[Acquiring/Generating the AST 9](#_Toc468456549)

[Performance and completeness 10](#_Toc468456550)

# Description of Tool

BinSkim is a checker that examines portable executables and their associated PDBs to identify various security problems. These include:

* **Use of outdated compiler toolsets.** Binaries should be compiled against the most recent compiler toolsets wherever possible to maximize use of current compiler-level and OS-provided security mitigations.
* **Insecure compilation settings.** Binaries should be compiled with the most secure settings possible, to enable OS-provided security mitigations, maximize compiler error and warnings reporting, etc.
* **Signing issues**. Signed binaries should be signed with cryptographically strong algorithms.

# Source and Drop Location

BinSkim is a Github-hosted [open-source project](https://github.com/Microsoft/binskim/). The latest version of the tool is always published as a [NuGet package](https://www.nuget.org/packages/Microsoft.CodeAnalysis.BinSkim/).

# Running BinSkim from the Command-Line

## Quick Start

The primary function of BinSkim is to analyze Windows portable executables (.dlls, .exes, etc). To analyze a file, pass one or more arguments that resolve to references to portable executables.

Note: the colon separator is option for nearly all ModernCop multi-part command-line arguments, as the mixed examples indicate below.

binskime.exe analyze MyProjectFile.dll --verbose  
binskim analyze \*.exe \*.dll --recurse  
binskim analyze \*.dll --output MyReport.sarif

## Command-Line Argument Reference

### Help command

The following command-lines invoke the general BinSkime help message. This message will display all the built-in ModernCop commands (help, analyze, capture, et al) for which more detailed help can be requested:

binskim.exe --help

To request detailed help for specific commands, invoke ‘binskim.exe help [command]’, eg:

binskim.exe help analyze  
binskim.exe help exportRules  
binskim.exe help exportConfig  
binskim.exe help dump  
binskim.exe help version

### Analyze Command

The ‘analyze’ command supports the following additional arguments:

|  |  |
| --- | --- |
| Argument (short form, long form) | Meaning |
| --sympath | Symbols path value, e.g., SRV\*http://msdl.microsoft.com/download/symbols or Cache\*d:\symbols;Srv\*http://symweb |
| -o, --output | File path to which SARIF-formatted analysis output will be written. |
| -v, --verbose | Emit verbose output. The resulting comprehensive report is designed to provide appropriate evidence for compliance scenarios. |
| -r, --recurse | Recurse into subdirectories when evaluating file specifier arguments. |
| -c, --config | (Default: ‘default’) Path to policy file that will be used to configure analysis. Passing value of 'default' (or emitting the argument) invokes built-in settings. |
| -q, --quiet | Do not log results to the console. |
| -s, --statistics | Generate timing and other statistics for analysis session. |
| -h, --hashes | Output SHA-256 hash of analysis targets when emitting SARIF reports. |
| -e, --environment | Log machine environment details of run to output file. WARNING: This option records potentially sensitive information (such as all environment variable values) to any emitted log. |
| -p, --plug-in | Path to plug-in that will be invoked against all targets in the analysis set. |
| --help | Displays this table of argument information. |
| --version | Displays BinSkim version details. |

In addition to the named arguments above, BinSkim accepts one or more specifiers to a file, directory, or filter pattern that resolves to one or more binaries to analyze. arguments can include wildcards, relative paths (in which case the file or directory path is resolved relative to the current working directory), and environment variables. All these arguments can be applied one or more times on the command-line. For analysis to occur, at least one specifier must be passed that resolves to one or more files.

#### --sympath

The --sympath argument is used to provide a path to a symbol server. The syntax for this argument is identical to the symbol path provided to Windows debuggers, as documented at <https://msdn.microsoft.com/en-us/library/windows/hardware/ff558829(v=vs.85).aspx>. As per this documentation, the symbol path can also be used to specify a directory location to cache any downloaded symbols.

NOTE: BinSkim requires PDBs to complete a significant subset of its analysis (see list below) which generally should be located alongside a target .dll or .exe. BinSkim explicitly clears any symbol path configured in the environment via %\_NT\_SYMBOL\_PATH% to prevent unexpected network activity or slowdowns related to PDB acquisition during analysis.

When BinSkim cannot properly load a PDB, because it is missing, corrupted, etc., the tool will emit an instance of ERR97. This message will report the problem including information on the specific HRESULT (and its meaning) error code returned by the PDB loading API, e.g.:

error ERR997.ExceptionLoadingPdb : BA2013 : 'symsrv.dll' was not evaluated for check 'InitializeStackProtection' because its PDB could not be loaded. ‘(E\_PDB\_NOT\_FOUND (File not found))’

|  |  |  |
| --- | --- | --- |
| Id | Name | Data Examined |
|  | BuildWithSecureTools |  |
|  | DoNotDisableStackProtectionForFunction |  |
|  | DoNotIncorporateVulnerableDependencies |  |
|  | EnableCriticalCompilerWarnings |  |
|  | EnableStackProtection |  |
|  | InitializeStackProtection |  |

The **/ref** argument specifies a JS or HTML file which is not a literal target for analysis but which is parsed in order to provide information required to analyze other files. NOTE: Reference files for JavaScript are typically discovered by parsing in-lined XML doc comments that refer to them (these annotations are also used to drive IntelliSense in Visual Studio). See the [section on annotations](#_Specifying_Dependency_References) for more information. Files specified by the /ref command are added as references to all files specified (directly or indirectly) for analyze via the /dir and /file arguments.

#### /enable and /disable

The /**enable** and /**disable** arguments are one (of several) mechanisms for enabling and disabling individual analysis checks. These arguments are mutually exclusive (that is, you cannot specify both /enable and /disable on the same command-line). The reason is the resulting behavior when specifying each argument kind. If the /enable arguments is encountered, all checks discovered at runtime are first disabled and then each check specified via /enable is enabled once more. The result is an analysis that is excluded to the checks that are explicitly provided. If the /disable argument is used, all checks discovered at runtime are enabled with the exception of those explicitly specified.

Individual checks for HTML are prefixed with WEB, eg, WEB1001. JavaScript checks are prefixed with JS, eg, JS3092. Both /enable and /disable accept a comma-separated list of checks. Check ids can be further qualified with the human readable descriptor for a rule (‘FollowKeywordWithSpace’) as well as its general id (referred to as its ‘opaque’ id):

// Analyze myfile.js with only rules JS3092 and JS2030 enabled  
mod50.exe /file:MyFile.JS /enable:JS3092,JS2076.IdentifierIsMiscased

// Analyze myfile.js with all rules except for JS3092 and JS2076  
mod50.exe /file:MyFile.JS /enable:JS3092,JS2076.IdentifierIsMiscased

#### /groups

The /**groups** argument provides another way to enable or disable analysis checks. Individual rules can have one or more group names associated with them. A check to insure that all ALT attributes have meaningful content is grouped as both ‘seo’ and ‘accessibility’ (since the check applies to both general analysis concerns). Currently, the group of most interest is ‘seo’, as the most significant HTML analysis is a comprehensive search engine optimization report. For Windows Web Applications, it is appropriate to disable this analysis. ModernCop will do this automatically on discovering a file named AppManifest.xml that lives alongside any specified file or directory. Otherwise, see below for an example of how to disable the SEO report during analysis.

The content of the /groups argument is a comma-separated list of group names. The hyphen character is used as a prefix to indicate that a group should be disabled. The absence of a prefix (or the optional presence of a plus sign prefix) indicates the group should be enabled.

The current supported groups are ‘seo’ (a general SEO report for HTML), ‘html’ (a set of general correctness checks for html), ‘js’ (correctness checks for JS) and ‘spelling’ (which is restricted to the two rules that spellcheck comments and identifiers). These groups are likely to be refined over time.

#### /js and /html

The **/js** and **/html** arguments provide the ability to remap file extensions that are associated with JavaScript and HTML analysis, respectively. Both arguments accept a comma-separated list of extensions (which have an optional “.” prefix. By default, all files with the “.js” extension are regarded as JavaScript (“/js:js”). By default, all files with either a “.htm” or “.html” extension is regarded as HTML (“/html:htm,html”).

#### /jslib

The **/jslib** argument is used to remap the location of the directory that contains supporting JS reference code consumed by the Coffeemaker semantic checker. This argument can also be set to a special value of ‘none’ (/jslib:none) with the result that semantic analysis will be skipped entirely. ModernCop will emit a warning and disable semantic analysis if the specified jslib directory cannot be located. NOTE: this argument cannot be used to point ModernCop to a new location containing semantic checker extensions. Due to .NET assembly probing path requirements, all semantic checker extensions need to be located in the ModernCop install directory or a directory named ‘jslib’ within that location.

#### /out and /in

The **/out** argument is a general output command. Currently, this argument can be used to specify the following built-in formats (specified by extension):

* : .**modcop** Outputs a ModernCop project file that contains all the analysis configuration details specified on the command-line (and discovered at analysis time). See the ‘[Project Files’](#_Project_Files) section in this document for more information. In the future, ModernCop will output other formats as well (such as the XML output used with the PREfast defect viewer).
* .**csv**: Outputs a CSV file of all reported messages. (Note: this functionality is implemented as a custom logger in Loggers.dll).
* .**wtl**: Outputs a WTT log file to the current working directory (or to the location specified in the %WTTRunWorkingDir% environment variable, if it is defined). If a file of identical name exists at this location, analysis results will be appended to that location (Note: this functionality is implemented as a custom logger in Loggers.dll).

The /out argument also provides a mechanism for binding to a custom logger at runtime. The expanded syntax for invoking this functionality is to provide a comma-separated string that provides the output file name, the assembly containing the custom logger, and the type name of the class that implements the ModernCop IMessageLogger interface. See ‘[Authoring Custom Loggers’](#_Authoring_Custom_Loggers) for more information.

This expanded syntax can be used to specify an output format that is compatible with the standard defined by the PREfast and PREfix tools. This XML standard is commonly used for all OACR analysis. To invoke it, suffix the /out argument file name with “Loggers,PrefastLogger” (with no quotes), eg:

mod50.exe /file:MyFile.JS /out:OacrOutput.xml,Loggers,PrefastLogger

The **/in** argument accepts a .ModernCop project file and uses the information in it to drive analysis. The output project file from a previous command-line generated by use of the /out argument should result in precisely the same analysis executing when subsequently passed to the tool via /in, eg:

// Analyzes myfile.js with only rule JS3092 enabled  
mod50.exe /file:MyFile.JS /enable:JS3092 /out:MyProject.ModernCop

// Also analyzes myfile.js with only rule JS3092 enabled  
mod50.exe /in:MyProject.ModernCop

The /in argument can be combined with additional command-line arguments, in which case, new command-line details will be applied after ModernCop first configures itself from the project file:

// Analyzes according to project file and also enables rule JS3092  
mod50.exe /in:MyProject.ModernCop /enable:JS3053

The ‘[Project Files’](#_Project_Files) section contains more detailed information on the ModernCop project file format and how it can be modified in order to configure analysis in more sophisticated ways.

#### /setting

The /setting command is used to pass arbitrary key/value pairs on the ModernCop command-line. This mechanism is used to control internal variables and settings (which aren’t generally a formal part of the ModernCop command-line for users). It can also be used to pipe additional parameters to extensions (such as custom loggers). As an example, ModernCop supports a special internal setting to generate a Coffeemaker command-line that mirrors the settings used by ModernCop to drive its analysis:

## Project Files

Below is a sample ModernCop project file that has been reduced to its essential components, which are:

* A ‘CommandLine’ node that recapitulates the command-line arguments specified for the run. This element also includes a ‘RootDirectory’ property which can be used to assist with constructing relative paths for analysis targets.
* A ‘Dictionary’ node that holds the ModernCop custom dictionary information.
* A ‘Files’ node that holds that actual file list that was generated at analysis time, based on information specified via /dir and /file arguments (which might include wildcards).
* A ‘RuleSettings’ node that specifies analysis settings relevant to individual checks or (in some cases) to the overall run. This section is likely to be broken out in the future to distinguish general analysis settings from the individual checks.

More information on each section is included after the example XML.

<?xml version="1.0" encoding="utf-8"?>  
<Properties Key="ModernCopProject" Type="WebContentProject" Version="1.0.0.0">  
 <Properties Key="CommandLine" Type="CommandLine" Version="1.0.0.0">  
 <Properties Key="Files" Type="PropertyBag" Version="1.0.0.0">  
 <Property Key="c:\temp\\*.js" Value="Analyze" />  
 </Properties>  
 <Property Key="RootDirectory" Value="c:\tools" />  
 </Properties>  
 <Properties Key="Dictionary" Type="WebDictionary" Version="1.0.0.0">  
 <Properties Key="Recognized" Type="PropertyBag" Version="1.0.0.0">  
 <Property Key="async" Value="All" Type="WordContext" />  
 </Properties>  
 </Properties>  
 <Properties Key="Files" Type="PropertyBag" Version="1.0.0.0">  
 <Property Key="c:\temp\test.js" Value="Analyze" />  
 </Properties>  
 <Properties Key="RuleSettings" Type="PropertyBag" Version="1.0.0.0">  
 <Property Key="ComprehensiveDom" Value="False" Type="System.Boolean" />  
 <Property Key="JS2001.DoNotUseEval.IssueTypeOverride" Value="Default" Type="IssueType" />  
 </Properties>  
</Properties>

### CommandLine Property

<Properties Key="CommandLine" Type="CommandLine" Version="1.0.0.0">  
 <Properties Key="Files" Type="PropertyBag" Version="1.0.0.0">  
 <Property Key="c:\temp\\*.js" Value="Analyze" />  
 </Properties>  
 <Property Key="RootDirectory" Value="c:\tools" />  
 </Properties>

# Authoring Custom Loggers

ModernCop provides a mechanism for authoring custom loggers, which can be invoked on the command-line via the [/out](#_/out_and_/in) argument by passing a colon-separated list that specifies the output file, followed by the assembly name of the custom logger dll (which must exist in the ModernCop install directory), followed by the fully-qualified type name of a class that implements the IMessageLogger interface. The following command shows an /out argument that is configured to emit a CSV format provided by a standard ModernCop logger:

/out:MyResults.csv;Loggersl;CsvLogger

Note: as a convenience, ModernCop does not require users to add the assembly and type name for the .csv or .wtl extensions. The fully-qualified syntax above, however, is supported if provided.

To create a custom logger:

* Create a new managed project and add a reference to Microsoft.BrowserTools.Analysis.Diagnostics.dll.
* Create a new class in the project and add the following using statement to it.

Using Microsoft.BrowserTools.Analysis.Diagnostics;

* Implement the IMessageLogger interface defined in this namespace.

## IMessageLogger

The IMessageLogger is a simple interface that consists of three methods.

namespace Microsoft.BrowserTools.Analysis.Diagnostics

{

public interface IMessageLogger

{

bool Initialize(string outputFile, bool force, PropertyBag settings);

void WriteMessages(Uri uri, IEnumerable<Message> messages);

void Close();

}

}

* **Initialize**: this method is called before analysis occurs. It is passed the output file name as specified on the command-line and a ‘force’ argument that indicates whether the user provided the /force switch on the ModernCop command-line. ModernCop does not enforce any policy around overwriting or merging to output files if they already exist at the specified location: this is left to the logger. The ‘settings’ command is a dictionary of any command-line options specified via the ModernCop [/setting](#_/setting) argument.
* **WriteMessages**: After analysis is complete, this method will be called once for each uri for which one or more messages fired. The ‘messages’ argument holds an enumerable collection of all messages relevant to the specified uri. This method will not be called at all if no messages fired during analysis.
* **Close**: Once all messages are processed, the Close() method is called. Loggers should flush all outstanding output at this point, close open resources, etc.

## CSV Custom Logger Example

using System;

using System.Collections.Generic;

using System.IO;

using Microsoft.BrowserTools.Analysis.Diagnostics;

public class CsvLogger : IMessageLogger

{

private StreamWriter writer;

public bool Initialize(string outputFile, bool force, PropertyBag settings)

{

this.writer = new StreamWriter(outputFile);

this.writer.WriteLine("SourceUri,Id,HumanReadableId,"+

"IssueType,DetailedDiagnosis");

return true;

}

public void WriteMessages(Uri uri, IEnumerable<Message> messages)

{

foreach (Message message in messages)

{

string diagnosis;

diagnosis = LoggerUtilities.BuildDetailedDiagnosis(

message,

enquote:

true);

writer.Write(uri.LocalPath + ",");

writer.Write(message.Rule.Id + ",");

writer.Write(message.Rule.HumanReadableId + ",");

writer.Write(message.IssueType.ToString() + ",");

writer.WriteLine(diagnosis);

}

}

public void Close()

{

this.writer.Dispose();

this.writer = null;

}

}

# FAQs

* [How does ModernCop raise its ASTs? Can I consume them?](#_How_does_JSCOP)

## How does ModernCop raise its ASTs? Can I consume them?

### General Notes

ModernCop consumes an AST raised by the JS runtime. This AST corresponds precisely to that provided at runtime when executing code. The runtime also exposes its tokenizer (used in the editor for colorization) and provides additional hooks for generating completion lists at arbitrary points in a document (used for intellisense).

### Acquiring/Generating the AST

In order to get access to the AST, ModernCop locates a recompiled version of jscript9.dll (named jscriptls.dll, for ‘lang service’). Recompilation is done in order to reguid certain constructs and to provide a tooling story that doesn’t access system bits. After locating the dll, ModernCop initiates a COM instantiation process entirely via PInvoke (ie, we pinvoke to LoadLibrary, we pinvoke to GetProcAddress to acquire the DllGetClasObject entry point, and we call this directly to instantiate). (Parenthetically, this routine has been reviewed for vulnerability to the recent SSIRP dll pre-loading issue, which can be provoked by [DllImport]-provided native interop in managed code).

ModernCop sends a string representation to the runtime bits and gets back a string representation of the file, entirely in JSON. ModernCop uses a simple, hand-written JSON parser to deconstruct this information into a managed representation of the tree. The managed classes that represent call expressions, bitwise operations, etc, are auto-generated from a table pulled from jscript header files. Generally, the managed view maps closely to the JSON data (with some friendly renaming). Member data is mostly weakly typed as either an AST node or a collection of them.

### Performance and completeness

Performance is good especially after replacing the original JSON deserialization mechanism provided by .NET. The ASTs themselves have been comprehensively tested. Most read scenarios have reasonable coverage and the ASTs are usable for general visitor patterns. The AST has been augmented in some cases to assist w/inspecting data that is, by design, not expressed in abstract trees, such as parentheses, the addition of a virtual semi-colons to a statement, etc. Comments are entirely absent from the AST, ModernCop folds these into its general visit by means of information acquired from the tokenizer.  This mechanism needs to be revisited.