Extending and Adding Checks

# WatcherCheck Class

New checks can be added by writing a class that inherits from WatcherCheck. From here, there are only a few methods required for you to implement:

## Required virtual methods

public override void Check(Watcher watcher, Session session, WatcherEngine.UtilityHTMLParser htmlparser)

The WatcherCheck class defines Check() and uses it as the entry point for running whatever logic the check implements. The Check() is run asynchronously in a thread and must handle all data structures in a thread safe manner.

public override String GetName()

Watcher interrogates checks to provide their name for display in the configuration list. You will need to implement a GetName() function which returns the name of your check.

public override String GetDescription()

Watcher also attempts to display a description of the check to users. This should be implemented by checks to allow users to know what the check does.

## Optional virtual methods

public override System.Windows.Forms.Panel GetConfigPanel()

This is an optional method that allows you to provide a UI control for configuration of your check. The control will be placed in the lower configuration panel on the Config tab when the check is selected. WatcherCheckLib provides a few predesigned configuration panels that can be used or you can design your own.

public override void Clear()

Some checks need to store state across requests, but not across running sessions of the tool. Watcher has a notion of clearing the alert history or the history checks may be maintaining as well. When a clear is triggered, Watcher calls the Clear() method of each WatcherCheck. If you have state to clear, implement Clear().

# Watcher Class

Your check will be passed a watcher object (Watcher class) and a session object (Fiddler.Session class). These objects will allow you to access application functionality as well as the data Fiddler has intercepted. Some of the important functions exposed by the watcher object are:

public static string Base64Encode(string data)

public static string Base64Decode(string data)

base64Encode() and base64Decode()- Use these methods in your UI to encode all data before storing it using the configuration method below.

public static Icon GetEmbeddedIcon(string strName, Assembly a)

public static Image GetEmbeddedImage(string strName, Assembly a)

GetEmbeddedIcon() and GetEmbeddedImage() – These methods allow you to retrieve Icons or Images you may be using for display.

public String GetOriginDomain()

GetOriginDomain() – Retrieves the current Origin domain for filtering and trust purposes. (**Please note** - Watcher calls every enabled check on every request. It is up to checks to properly filter based on Origin and Trusted domains).

public bool IsOriginDomain(String hostname)

IsOriginDomain() – Similar to the method above but will handle an Origin which has been specified as a regex.

public static String GetCheckConfig(WatcherCheck check, String configoption, string defaultval)

public static void SetCheckConfig(WatcherCheck check, String configoption, String value)

GetCheckConfig() and SetCheckConfig – These methods allow checks to store configuration settings as part of the application settings. Check configurations are isolated and checks cannot see each other's config items. Since the data is stored as a string it should be Base64 encoded if it contains any non-printable characters. GetCheckConfig() can also take a default value which allows the check to initialize the config setting the first time it is loaded. (**Please note** - Configuration is read once at load time. Updates will not be visible until the user clicks save. Checks should store config state in the UI, updating the config whenever the UI changes. State will not be preserved across Fiddler sessions unless the user clicks save.)

public bool IsTrustedDomain(String domain)

IsTrustedDomain() – This allows checks to determine if a host is listed as a trusted domain.

public static void SaveConfig()

SaveConfig() – Though exposed, this should not be called by checks.

Finally, checking wouldn't be useful without reporting, so most checks will implement something like:

private void AddAlert(Watcher watcher, Session session, String cookie)

Inside AddAlert() our checks call **Watcher.AddAlert(int level, int ID, String URL, String typex, String description)** as a way to report when a security issue has been identified. This is demonstrated in the following sample check, which reports when weak authentication methods such as Basic and Digest are used.

using System;

using Fiddler;

public class HeaderWeakAuthWatcherCheck : WatcherCheck

{

private void AddAlert(Watcher watcher, Session session, String context)

{

string name = "Weak authorization method";

string text =

name +

"\r\n\r\n" +

"Risk: Medium\r\n\r\n" +

"The server issued a weak Basic or Digest authorization challenge:\r\n\r\n" +

session.url +

"\r\n\r\n" +

"The context was:\r\n\r\n" +

context;

watcher.AddAlert(Watcher.Medium, session.id, session.url, name, text);

}

public override void Check(Watcher watcher, Session session)

{

bool authBasic = session.oResponse.headers.ExistsAndContains("WWW-Authenticate", "Basic");

bool authDigest = session.oResponse.headers.ExistsAndContains("WWW-Authenticate", "Digest");

string headers = session.oResponse.headers.ToString();

if (watcher.IsOriginDomain(session.hostname))

if (session.responseCode == 401)

if (!session.HTTPMethodIs("CONNECT"))

if (authBasic || authDigest)

AddAlert(watcher, session, headers);

}

}