Using Code Inspection to Detect Security Vulnerabilities and Manage Potential Ones

A Case Study With ASP.NET and Custom FxCop Rules

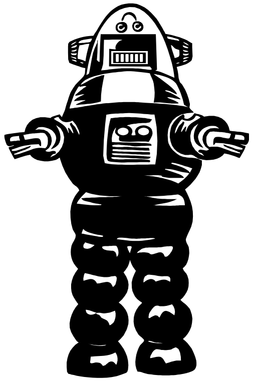
Chris Shaffer

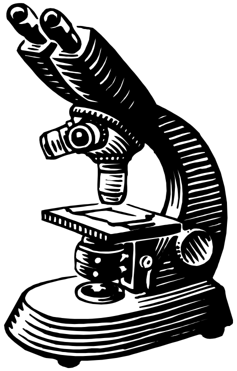
Director of Engineering, Relationship Science

Relationship Science REL SCI

We’ll be addressing…

- Cross-Site Request Forgery
- Cross-Site Scripting
- JSON Request Hijacking
- SQL Injection
- Management and inventory of application’s attack surface
- Enforcing organization-specific best practices





Back up … What is it, anyway?

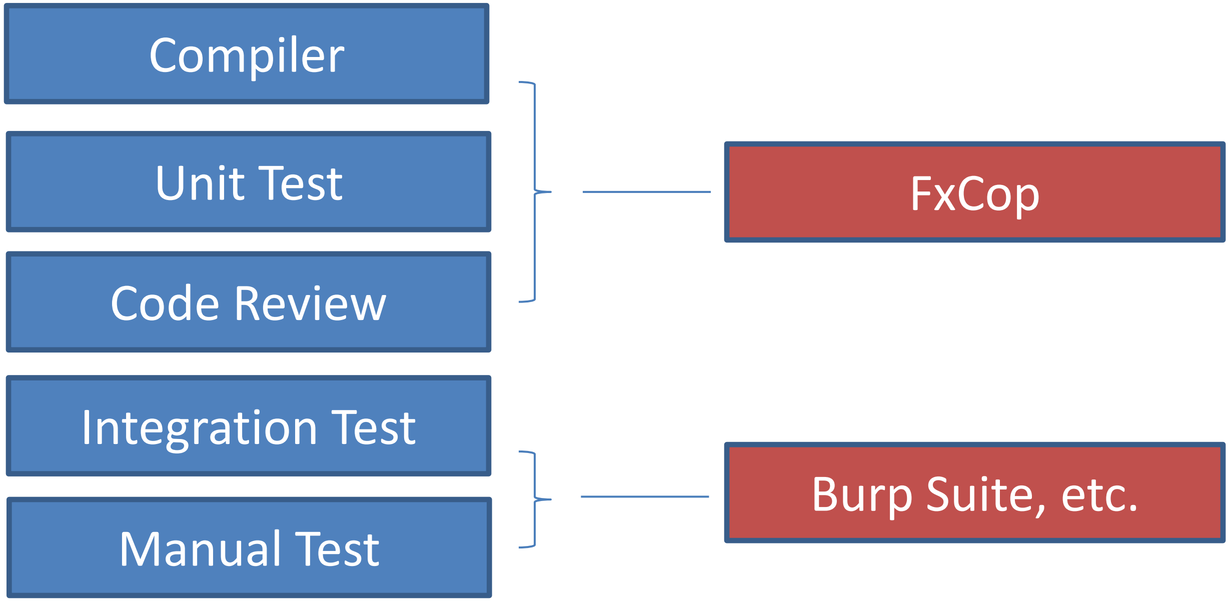
STATIC CODE ANALYSIS

Static Code Analysis

- FxCop is a static analysis program used to find design and usage flaws, and failure to adhere to best practices.
- It’s code reading other code and then criticizing it.
- FxCop is just one of many.

Your code is great … for me to poop on.

When Testing Happens



Advantages to Static Analysis

- Runs at earlier stage of life cycle
- Can run scans much faster
- Can detect unsafe functions before code that calls them is even written, poor design that doesn’t leave external evidence
- Of course, it works best in parallel, not as a replacement

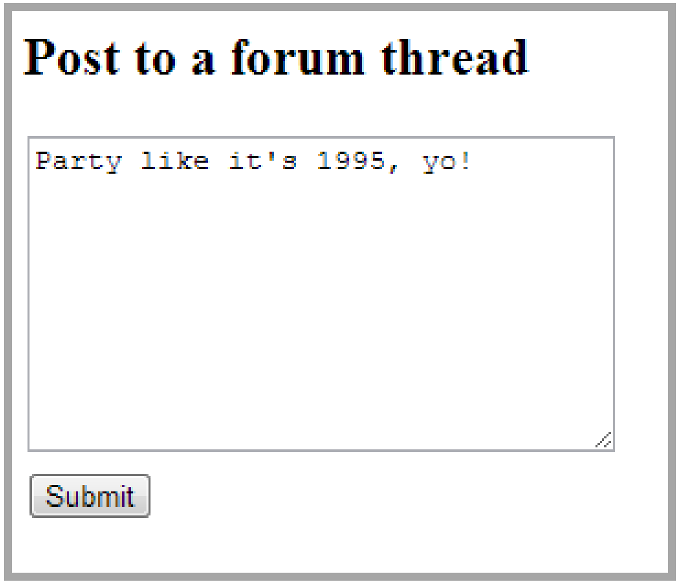
Some Examples of Rules Microsoft Gives Us

- Declare Types in Namespaces
- Do Not Catch General Exception Types
- URI Properties Should Not Be Strings
- Avoid Uncalled Private Code
- Review Unused Parameters
- And many more …

First, let's review how CRSF attacks work

CROSS-SITE REQUEST FORGERY

So, a victim is logged into a site …

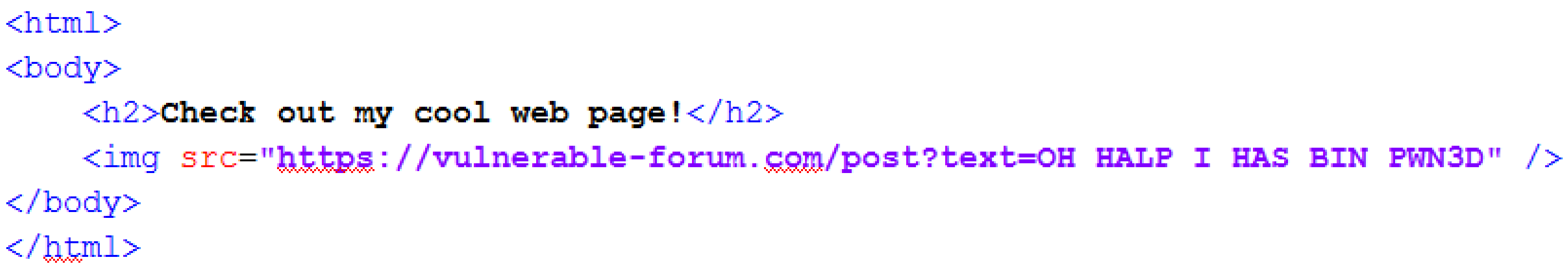


Post to a forum thread

Party like it's 1995, yo!

For this example, let’s pretend the authentication of this site is secure …

An attacker tricks them into clicking a link …



```
<html>
<body>
    <h2>Check out my cool web page!</h2>
    <img src="https://vulnerable-forum.com/post?text=OH HALP I HAS BIN PWN3D" />
</body>
</html>
```

… or maybe they use a cross-site scripting vulnerability on a trusted site to stick that image in an ad on a news site …

If the victim user is logged into that site, and opens this link in the same browser, it's going to use their login cookie and execute as if they clicked the 'post a message' button themselves.

Or maybe the attacker is even more brazen …

“Hey, click this link!”

https://nostalgia-forum.com/post?somecrap=look-nontechnical-users-arent-even-going-to-read-this-seriously-its-techie-computer-nerd-stuff&nonsense=30ijrf3j09fij3ef09j3f0i3j2390fj30i4h4309u3409hf30f9h340&text=O%20HALP%20I%20STILL%20HAS%20BIN%20PWN3D

“Oh, I use the Nostalgia Forum all the time, of *course* I can trust a link to an article on it.”

An application-wide solution to not being that guy

CSRF MITIGATION

Generate a token, and add it to every form

- Require that the token be submitted with any form.
- This will ensure that a form submission to your site came *from* your site.
- Fortunately, the .NET Framework gives us a built-in method for this …

Built-In Anti-Forgery in .NET

To add a token to the page's markup:

```
@Html.AntiForgeryToken()
```

To check the token's validity:

```
AntiForgery.Validate();
```

That's It!

Sort of.

You’d still have to add that code to every sensitive form post …

That’s a pain.
Let’s not do that.

- Add that token to our _Layout file
- Have JavaScript append to it every form post
- Have an HttpModule that runs on every request
  - If the request is a POST and is from a logged-in user, look for a token
  - If the token is missing, throw an error, otherwise validate it
- Add `[HttpPost]` attributes to those sensitive form posts to make sure an attacker can’t hit them with a GET request (which would bypass the above)

We’re Done!

Hold up.
Do you have any idea how fast you were going, son?

```
public class DerpDerpImNewAtWebDevelopmentController : Controller
{
    [HttpGet]
    public ActionResult TransferAllOfMyMonies(int destinationAccount)
    {
        //didn't plan for me, didja, punk?
    }
}
```

FxCop to the rescue!

HOW CODE ANALYSIS FITS IN

First Rule

- First rule of CSRF: ~~Don’t~~ talk about CSRF

- Looks for any Action without [`HttpPost`]
- If the name of the Action contains any words that indicate editing of data…
  - Such as … Save, Update, Edit, Delete, etc.
- Or if it *calls* any functions named like that
- Flag it as a violation

Second Rule

- We flag anything that's missing an `[HttpGet]` or `[HttpPost]` tag.
  - FxCop allows us to differentiate between "warnings" and "errors", so we can treat these as a lower priority.
- To catch cases in which a data-altering function doesn't contain one of those keywords
  - Maybe we didn't think of a keyword we should have
  - Maybe someone made a typo
- Basically, this ends up being a way of making sure new code that adds to our attack surface is tracked and code reviewed.

If you have to support crappy browsers…

JSON HIJACKING

JSON Hijacking

- It’s similar to CSRF, except it uses JSON GETs to eavesdrop on the contents of those requests or even convince your browser to forward your session cookie to the attacker.
- Details: http://haacked.com/archive/2009/06/25/json-hijacking.aspx
- Fixed in most modern browsers.
- But, it’s a great example.

```
public class JSONAllowGetRule : BaseIntrospectionRule
{
   public JSONAllowGetRule()
     : base("JSONAllowGetRule", "RelSciCustomRules.RelSciCustomRules", typeof(RelSciBaseRule).Assembly) {}

   public override TargetVisibilities TargetVisibility { get { return TargetVisibilities.All; } }

   public override ProblemCollection Check(Member member)
   {
      var m = member as Method;
      if (m != null)
      {
         VisitStatements(m.Body.Statements);
      }
      return Problems;
   }

   public override void VisitMethodCall(MethodCall call)
   {
      if (call.Method() != null)
      {
         if (call.Method().Parameters.Any(a => a.Type.FullName == "System.Web.Mvc.JsonRequestBehavior"))
         {
            this.Problems.Add(new Problem(this.GetResolution(), (Node)call));
         }
      }

      base.VisitMethodCall(call);
   }
}
```

The point of the next few rules is to identify high-risk code to make sure it’s first in line to be code reviewed.

Policing to ensure the proper checks are in place.

AUTHENTICATION AND AUTHORIZATION

Authentication

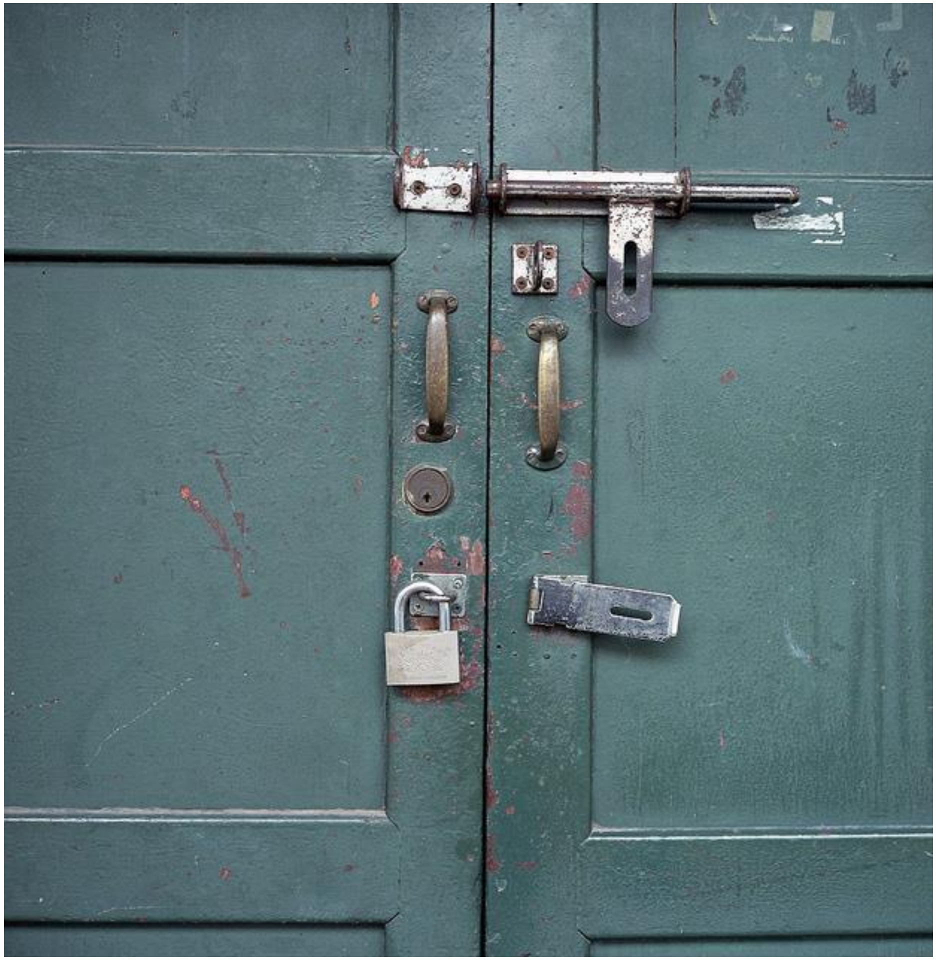
- Our setup: Any request is required to be authenticated by default; any action that a developer wants to be accessed by an un-authenticated user has to be marked with a `[SkipAuthorize]` attribute
- To ensure it's being used correctly, we have FxCop generate a warning for any new `[SkipAuthorize]` to be code reviewed.

Authorization

- .NET provides attributes:

```
[Authorize(Roles, Users, etc.)]
```

- We use these to control access to premium features, and also administrative functions.
- If it’s missing, any authenticated user could hit this page.
- FxCop rule: Flag any Controller that doesn’t have this attribute specified.



What happens when code bypasses the framework’s protections?

CROSS-SITE SCRIPTING

XSS

- Flags functions that return an `IHtmlString` for code review.

- We haven't tried it, yet, but an approach similar to the next one (SQL injection) might work, here.

XSS (side note)

Replace this:

```
public class AViewModel
{
   public IHtmlString AHyperlink { get; set; }

// ... elsewhere
AHyperlink = new HtmlString("<a href='" + Url.Action("doinit") + "'>clicky</a>");
```

With this:

```
public class AViewModel
{
   public UrlHelper Url { get; set; }
   public IHtmlString Ahyperlink {
      get {
         return new HtmlString("<a href='" + Url.Action("doinit") + "'>clicky</a>");
```

Also, use a TagBuilder.

The final frontier.

SQL INJECTION

SQL Injection – Attempt 1

- Having a strictly enforced 3-layer architecture allows us to limit checks to just libraries with database access.
- Follow "string-ish" method parameters

string, StringBuilder, IEnumerable<string>, user-defined types with public string properties, string cheese.

- Flag unsafe concatenations, appends, etc. – anything with non-numeric arguments

SQL Injection – Failures of Attempt 1

- False alarm: String concatenation for display logic in the data layer.

This is bad practice, but it’s not SQL injection …

- This:

```
//the procedure comes from a value in a drop-down
public IEnumerable<int> RunSomeSelect(string procedureName) {
```

It’s not part of a dangerous *concatenation*…

- And this:

```
var uhOh = repo.Query<string>(
   "select Value from SomeCrapTheUserTypedIn;").Single();

string sql = string.Format(
   "select * from SomeTable where Name = '{0}';", uhOh);
```

SQL Injection – Attempt 2

- Every declared or assigned variable…
  - Mark parameters “dirty”
  - Mark constants and literals “clean”
  - Mark the target “clean” if *all* of the components in a format or concatenation are “clean”
    - Otherwise, mark it “dirty”
  - If the source is unknown, it’s “dirty”
- If a SQL-executing function is called, and the SQL parameter is dirty, throw an error

SQL Injection – Attempt 2

- Fixes the issues with the first one
- Still has weaknesses:
  - It’s hard (false alarms)
  - Functions like these give errors that aren’t particularly informative; we can mitigate that with attributes.

```
[ExecutesSql]
private void RunMySQLAndDoSomethingWithTheResults(string sql)
{
```

```
var data = Query<int>(BuildCrazySearchSQL());
//... elsewhere
[BuildsDynamicSql]
private string BuildCrazySearchSQL()
```

SQL Injection – Other Ideas

- SafeSQLBuilder class – prevent string inputs, flag any SQL call that doesn’t use it
- The obvious …
  - Use procedures
  - Use parameters
  - Use enums when databases, tables, and/or structural pieces of queries are determined by parameters

Insert witty subtitle here.

ENFORCING BEST PRACTICES

Rules to Enforce Best Practices

- Declaring a static IDisposable
- Emails sent directly through the system library rather than our company's library
- Using a .Context property on the data layer

- Lots of possibilities for more rules

Integrating it into our build process.

PUTTING IT ALL TOGETHER

Integrating Into a Build Process

- TeamCity: Fail if the number of warnings and/or errors reaches a certain threshold.
- FxCop projects live in a repository that everyone has *read* access to, but only a few people have *write* access to.

So punks can't whitelist their own stuff.

- Command-line utility: Add every DLL in a given directory to an FxCop project file.

No forgetting to set up scanning for new code.

Relationship Science REL SCI

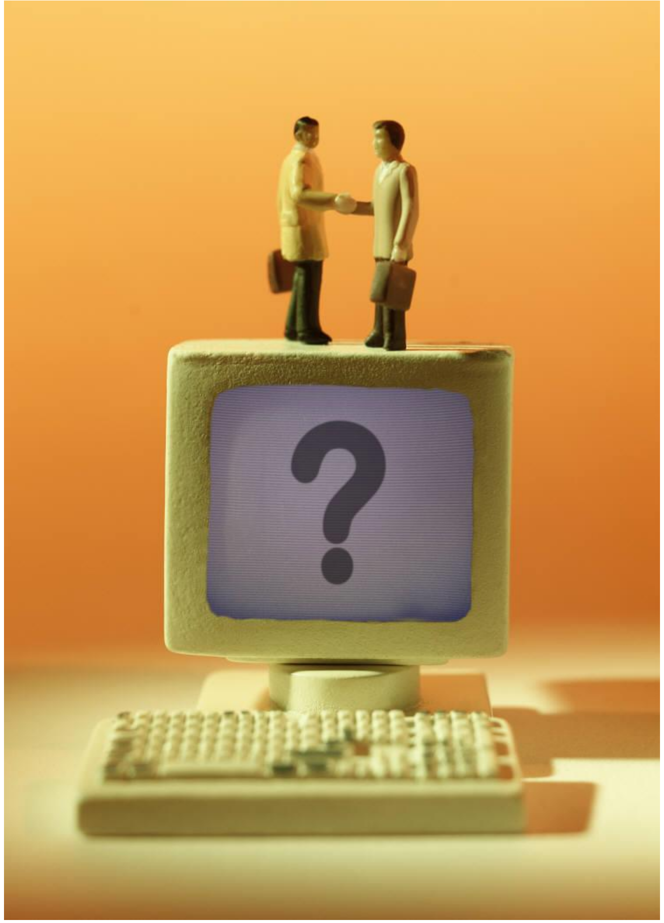
Next Steps

- Code that's not complied into an assembly …
  - Html.Raw() in .cshtml files
  - Dynamic SQL *inside* stored procedures
  - JavaScript
    - Example: Find code that makes a GET request to one of the actions turned up by the CSRF or JSON rules.

    (so they don't turn into page breaks when you fix the vulnerability)
- Mass Assignment attacks
- Make that SQL injection rule actually work
- And many more …

Ask them.

QUESTIONS



TEH CODEZ: https://github.com/DrShaffopolis/FxCop