# Intro

Hello, and welcome to “Don’t Write Secure Code”. I’m Seth Petry-Johnson, and unlike some of the other speakers in this track, I am not a security professional. I’m just a normal programmer, although I do have a security related confession to make.

**(click)**

That confession is that I hate writing secure code.

I’m not proud of it, but when I’m building a feature and I’m elbow deep in complex business logic or functional requirements, I don’t want to think about security. I want to be totally focused on that problem, but these ever-present security issues keep stealing my focus. The user has to be logged in to do this. They have to have some permission to do that. Alice shouldn’t be able to see Bob’s data. Bob shouldn’t be able to get all ticked off about something and drop the user table through SQL Injection. Yadda, yadda, yadda.

The constant need to pay attention to those same security concerns, over and over, feature after feature, bums me out.

So, maybe a better way of expressing myself is that

**(click)**

I hate writing secure *feature* code. I actually don’t mind thinking about security, and I enjoy making my system secure, I just want to separate the security code from my features. I want my features to be clean and simple and elegant, not sullied up with a bunch of duplicate security checks copied and pasted between features.

So I guess what I’m *really* trying to say is that

**(click)**

I hate implementing *cross-cutting security concerns* by repeating the same patterns over and over again in my *feature-level code*.

That’s a mouthful, but it’s exactly what this talk is all about. Because I hate doing this, I’ve invested a lot of time and energy looking for ways to avoid it. And the way that I typically avoid it is by extracting the security code into some part of my application framework so that it can be automatically applied across all of my features. This lets me keep my brain in “feature mode”, keeps my feature code clean, and I still get to deliver a secure system.

Over the next 55 minutes I’m going to share some of my techniques with you. I want you to recognize the intermingling of security code and feature code in your own systems, and I want to give you some tools for decoupling them so that you can maximize *both* security AND maintainability.

## Agenda

Here’s our agenda:

First I’ll define what it means for something to be a “cross cutting” security concern and what types of things are best suited to pushed into the framework.

**(click)**

Second, I’m going to show you some examples of cross-cutting concerns that you can 100% completely solve in your framework code, without requiring *any* changes to your feature code. I call this being “secure by default” because once these things are in place, it takes literally zero effort for developers to ship secure features.

If you’re familiar with the phrase “pit of success”, then that applies here because with these things in place it actually takes a developer MORE effort to be insecure, than secure.

**(click)**

Unfortunately, not everything can be made *fully* secure by default, and sometimes you’ll still need to put some sort of security code into your features. I’ll show you how a declarative approach results in better factored code that minimizes the intermingling of concerns.

**(click)**

Finally, I’ll show you some ways that you can use static analysis tools to perform a security audit, and how you could incorporate that audit into your automated testing process.

The code samples in this talk are in .NET and JS, because that’s what I’m familiar with. However, many of the techniques I’ll show you have parallels in other languages and platforms as well, so the general ideas should be portable even if the specific code samples are not.

## Defining “Framework”

The main point of this talk is the idea of pulling cross cutting concerns out of your feature code and pushing them down into your framework.

Before I show any code, let me define those terms a little better.

When I talk about “framework level code”, I’m referring to any code that leverages hooks in an underlying system or library so that it can be automatically executed at a specific time.

**(click for framework examples)**

For example, ASPNET MVC lets you tap into the MVC pipeline using Action Filter attributes.

ASP.NET HTTP Modules let you tap into the entire ASPNET request pipeline.

jQuery lets you tap into the AJAX request pipeline using these global event handlers.

ORMs like Entity Framework and NHibernate let you run code whenever a database connection is opened or a transaction begins.

And if you need a hook that doesn’t already exist, tools like PostSharp let you create your own extension points in existing code. We’ll talk about this in more detail in a bit.

And of course, your custom application code might provide its own framework level hooks. For instance, if you have some base class that everything derives from, then you can provide your extension points that make sense in your own system.

Basically, when I say “framework code”, I’m referring to something that you write that gets executed automatically by some underlying system, as opposed to the specific feature-level code you write at the top of your stack.

## Defining “Cross Cutting”

And when I say “cross cutting concern”, I’m referring to any security requirement that spans multiple features.

Cross cutting concerns can be very low level and unrelated to your business domain, such as SQL injection. The need to sanitize your query inputs applies to every system you build, no matter what business rules it has.

Higher level requirements can be cross-cutting as well. For instance, you might have a group of related pages or features that share the same set of security concerns, such as requiring a similar set of permissions or enforcing a similar set of access control requirements.

In both cases, though, the point is that separating the cross-cutting concerns from the feature or page-specific concerns makes your code easier to maintain and your system more secure.

To illustrate this, I used my world-class Visio skills to bring you this example.

**(click to example)**

This beautiful green square represents a feature on our website. It’s the “Order List” feature that displays to a user the list of orders that user is allowed to see.

This page has three requirements:

1. First, user must be logged in to see the page.
2. Second, users can only see their own data,
3. Unless they have a specific permission that grants them access to all records

**(click for addition of security box)**

The naïve approach is to implement those requirements directly within the feature code. This beautiful red square represents the security code being intermingled with the feature code. If this was an MVC app, for instance, this red square might represent a couple of lines of code in the body of a controller action.

**(click for Order Detail)**

The next feature we build is the Order Detail page so that the user can click on an Order and get more information about it. This page probably has the exact same security requirements for obvious reasons.

For the sake of this example, let’s assume those requirements are implemented exactly the same way on this feature as the first one.

**(click for Cancel Order)**

Next we build a feature to Cancel the order. Now, this feature *should* have the same requirements as the first two, but maybe it was built by a different developer that wasn’t security conscious or was under some deadline pressure. For whatever reason, this feature isn’t secure at all.

**(click for Refund Order)**

Finally, we build a fourth feature to Refund the order. Again, this has same requirements as the other features, but maybe the developer was unfamiliar with those implementations and they implement the same rules, but in a different manner. Maybe instead of doing the check in the controller action, they move it into the model or something.

The red diamond here represents the same basic rules, but implemented in a different way.

The problems with this approach might be obvious.

1. First, one of these 4 features is insecure. If your QA department primarily tests through the UI, then as long as the List and Detail page implement the security rules, they might never discover that the Cancel feature allows anonymous access. With this architecture, the only way to guarantee that QA would find this issue is if they do exhaustive, black-box testing of every security rule against every endpoint. That can be enormously expensive.
2. Second, maintaining the security code will be difficult. What happens when we add a new user role that allows a user to see all Orders placed by other users within the same organization?

**(click for triangle icon)**

If the developer making that change doesn’t realize that the same rules are duplicated in other features, they might end up changing just one of them. Now we have *three* implementations: triangle, square, and diamond.

Even if that developer does replace all places using the “square” implementation, they might still miss the Cancel feature with no security and the Refund feature with the “diamond” implementation.

This is how security defects creep into software. Well-meaning developers either forget to implement the security check, or they inconsistently maintain the security checks over time. Either way, we end up with a confusing and inconsistent mess of security code intermingled with feature code, or missing altogether.

**(click for Cross Cutting)**

My approach would be to extract those business rules into something reusable so that we can implement the requirements only once.

There’s multiple ways that you could do this.

**(click for ex #1)**

The simplest would be to extract this logic into a helper method and call it from all 4 controller actions or something. This would standardize on a single “square” implementation, but nothing prevents a developer from forgetting to call this method and leaving the feature insecure.

And again, the only way to *guarantee* that we’ll find those insecure features is by doing black-box testing of every single endpoint against the security requirements.

**(click for ex #2**)

Another approach would be to extract the logic into some sort of attribute that injects that logic into the appropriate part of the processing pipeline.

This also results in a consistent implementation, but it’s just as easy for a developer to forget. However, this approach is easier to audit. In a little bit I’ll talk about using static analysis to generate a report of secure and insecure endpoints based on attribute usage.

**(click for ex #3)**

In a perfect world, the developer wouldn’t have to do anything at all except write their business logic, and the security stuff would be handled automagically *and* in a way that was easily audited and tested. And that’s the point of this talk.

It isn’t always possible to get it *quite* this magical, but we can probably get a lot closer than you’d expect.

## Code sample transition

For the rest of this talk I’ll be showing you a bunch of code samples from a demo app I wrote. This app, which you can get from my Github page, shows the same features implemented three ways: first, with no security at all, then using “secure feature” code where the security requirement is handled directly within the business logic, and finally with the security concerns extracted into some part of the framework.

I will move through my slides pretty quickly, but everything you’ll see up here comes straight from that project that you can reference later. It’s fully functional and heavily commented so I hope it’s a useful resource for you.

To start, I’m going to show you 3 examples of what I call a “secure by default” system. Each of these examples demonstrates how you can solve a problem once, and then more or less forget about it. Once it’s been solved in the framework, no additional developer effort is needed on a feature-by-feature basis.

# CSRF

My first example deals with #8 on the OWASP Top 10, Cross Site Request Forgery. In case you’re not familiar with CSRF, here’s a quick overview.

**(click for CSRF diagram)**

First, a user logs into a site they trust, such as their bank.

Second, while the session with the bank is active, they visit a malicious website. This could be in a different tab but it doesn’t have to be.

Third, the bad guy website tricks the user into submitting a form post TO the bank’s website, for example to the “transfer money” endpoint. If the bank website hasn’t been properly secured, all it’s going to see is a request coming in, from a logged in user with a valid session, requesting a fund transfer into the bad guy’s account.

In order to protect itself from this sort of thing, the bank needs a way to differentiate between a form post that initiated from its own domain versus a post that initiated from the bad guy’s website. The HTTP Referrer can help with this, but it’s insufficient on its own because that value is spoofable.

**(click for CSRF – feature)**

ASP.NET gives us some tools to protect against CSRF. You have to do two things: call the AntiForgeryToken helper inside the body of your form, and add the ValidateAntiForgeryToken attribute to the action it posts to.

The helper does three things. First, it creates a cryptographic token based on the user identity. Second, it outputs that token into a hidden form field. Third, it sets a cookie with the same token value.

When the form is submitted, the ValidateAntiForgeryToken attribute checks to see if the token value submitted with the form is the same as the cookie value, and rejects the post otherwise. The browser’s security model prevents a malicious website from reading or writing the cookies for the friendly site, so the only way the tokens will match is if the form being submitted is coming from the friendly site itself.

This is easy to do, but it’s not secure by default. It requires the developer to make corresponding changes in two files. And if the developer remembers the helper, but forgets the attribute, then the form post will be accepted. The dev might think they’re secure, but actually the endpoint is vulnerable.

Also, this only works if you’re submitting a form. This helper doesn’t really help if you’re doing an AJAX post because there’s no form to write the hidden field into.

**(click for CSRF – feature (AJAX))**

If you’re making an AJAX form post, you have to do something like create a div, call the helper to create the hidden form field, and then manually copy that hidden form field value into your AJAX payload.

This really sucks. It’s ugly, it’s error prone, and it’s not a pattern that I want to be repeating every single time I need to make an AJAX post.

**(click for CSRF – framework #1)**

To handle this in a cross-cutting way we need to do a few things.

First, we need to create one, global anti-CSRF token. I do this in my global layout file so that it applies to every page on my site. Remember that this creates a hidden text field AND creates a cookie.

**(click for CSRF – framework #2 – clone field)**

Second, on document ready I run a tiny bit of jQuery that loops through every form on the page, looks to see if it already has a token field, and if not, clones the global one and adds it to the form.

This ensures that every single form that I use on my site will end up with a hidden field containing that security token.

**(click for CSRF – framework #3 - ajax)**

To handle AJAX posts, I run a second bit of JQuery that defines a global jquery “ajax prefilter” handler.

Hooking into the prefilter event lets you can modify the AJAX options *before* the request is sent. In this case, I add that CSRF token to every single AJAX POST.

**(click for CSRF – framework #3)**

Finally, we need to run the token validation logic automatically for all form post actions. We can do that by creating a custom controller class and overriding the OnActionExecuting method. The code that we put here will run on every single request, so if that request is a form post then we run the token verification logic.

The *only thing* the developer has to do is derive their controller from the correct base class. As long as they do that, everything is handled automatically.

**(click for CSRF – framework #4 – form post clean)**

Here’s what the feature level code looks like when we’re done. This is 100% business logic, and yet every single form post is still protected from cross site request forgery attacks.

**(click for CSRF – framework #5 – AJAX clean)**

And here’s what it looks like for an AJAX POST. Again, it’s 100% business logic, yet still protected.

CSRF defense is a great example of a cross-cutting concern because it’s orthogonal to individual feature requirements, it applies globally to the whole system, and it’s pretty easy to make it “secure by default”.

# Authentication

My next example of making a feature “secure by default” deals with Authentication, or specifically with preventing anonymous access to protected areas of your site.

Most web frameworks make this fairly easy to do.

## Auth – feature code

In ASPNET MVC, for instance, you can add the [Authorize] attribute to an Action and it will automatically redirect users to the login page if they aren’t logged in.

I don’t like using this though, because it represents a “public by default” model where any given MVC endpoint is accessible anonymously unless it is explicitly flagged as private. I tend to work on applications where the vast majority of resources are private, and only a specific few are public, so I want a “private by default” model instead.

One way to build a “private by default” model is to create a custom HTTP Module that runs on every request and enforces a login, unless the URL matches a whitelist of endpoints that allow anonymous access.

This is actually really easy to do.

## Auth – framework code

First, we need some place to manage that whitelist. Since our goal is to require zero changes to the feature code, I like to use a custom section of web.config to do this. That way I don’t need to make any changes at all to the controllers themselves.

Now, it’s important to note that HTTP Modules run for EVERY request, not just ones that get routed into MVC. That means that our whitelist needs to include static resources, such as stylesheets or scripts, that we want to use on our login page.

For that reason, I like to build my whitelist using regular expressions. That way I can create a single rule that grants access to my entire Scripts folder so that I don’t have to continually modify web.config each time we add a new jquery plugin.

Creating custom web config sections is pretty simple, but I don’t have time to show the code in this session so just check out my demo app to see how it works.

**(click**)

Once we’ve defined the whitelist, we need to create the HTTP module to enforce it.

This is also really easy to do. It’s just a class that implements the IHttpModule interface and then provides an implementation for this OnAcquireRequestState method.

In the body of this method we look to see whether or not the URL matches our whitelist and whether or not the user is already logged in. If either of those things are true then we do nothing. Otherwise, we send the user to the login page.

**(click)**

The last piece of the puzzle is to tell IIS to run that module for all requests, and we can do that with a single line of code in web.config.

And that’s it. Now, every single request that comes into my site will be automatically redirected to the login page unless the user is logged in, OR the URL matches a whitelist that I can control in a single, centralized place. The developers literally need to take zero additional effort to make their endpoints secure, and if they want to make something public it’s just a single adjustment to the whitelist.

Also, in the last segment of this talk, I’ll show you a way to generate a report of all of your MVC controller actions and whether or not they are publicly accessible. That sort of auditing is a useful way of validating that your whitelist is doing what you expect it do.

# Access Control

My final example of a “secure by default” framework deals with access control, which is about preventing users from accessing data that they don’t own or have rights to.

This type of requirement tends to cut across multiple features. For instance, if there’s some code that prevents Bob from seeing Alice’s orders on a list page, then we probably want to apply that same restriction on the order details page. This makes it a good candidate for a cross-cutting concern.

## Access Control – Feature Level code

This is what an access control requirement looks like in the List page feature code.

I call the data access layer, get a list of all orders, and then I explicitly remove ones the user can’t access. Again, this is an example of security logic being intermingled with feature logic.

**(click for Access Control – feature #2)**

That same requirement looks like this on the Order Detail page.

First I call GetById to retrieve the order, then I check the permission and kick the user out if necessary.

This is an example of the situation I showed you at the start of my talk. We have the same logical rule being implemented in two different ways. Without a centralized implementation it’s going to be very difficult to keep them in sync as requirements change.

There are two ways that we could centralize the implementation of this access control logic.

## Access Control – Data access framework

The easiest way to make this a cross-cutting concern is to push the responsibility for access control into your data access code where it can be reused by multiple features.

Here’s an example of a simple data service that provides two different ways to get order data. You can call GetAll and get everything, or you can call GetById and get back a single order.

There are two versions of each method – one that takes the current user as an argument, and one that doesn’t. If you call the version that takes a user argument, then it returns only the data that user can access. If you call the version that doesn’t take a user, then it does no extra filtering and you’re responsible for doing your own access control in your feature code.

Why have two versions? Well, there might be scenarios where there is no “current user”, such as an automated maintenance program running on a schedule.

**(click for Access Control – framework, with “secure” names)**

If you can avoid it, I’d recommend NOT having the insecure versions at all. But if you *do* have to provide two versions of your data access methods, I recommend a naming convention like you see here.

In this case, I’ve added the suffix “Insecure” to the methods that do NOT do any access control. The idea here is to remind programmers of their obligations when calling these methods.

For instance, if I type “OrderService.GetById”, it isn’t explicitly obvious whether or not access control is being handled. But if I type “GetByIdInsecure”, that’s a pretty clear reminder that I’m on the hook for security in my feature code.

This approach is better than nothing, and it does push the access control logic down from the top level feature code and into the data access layer. But this is far from “secure by default”; the security code needs to be manually added to every data access method, and that can result in a lot of duplication. It can also result in a lot of inconsistency if each method implements those rules in a haphazard way.

## Access Control – Row Level Security

Another way to handle access control is through a technique called Row Level Security. The idea here is that instead of filtering out data in our application code, we create a security policy *in the database itself* that does the filtering. Then, the application can simply ask for the data it needs, and the database will only return the data that the user is allowed to access. This essentially makes the access control transparent to the application code and makes the entire data access layer “secure by default”.

Of course, the devil is in the details, and this approach depends heavily on making this security policy thing aware of who the current user is. Unless you want to give each of your users a dedicated database login, this is typically easier said than done.

However, SQL Server 2016 added a new feature that makes this much, much easier to do.

**(click for RLS)**

This new feature is called the “session context”, and it’s basically a key/value collection that’s scoped to the database connection. This gives us a global dictionary that is shared by all queries within a connection.

You put a value into the collection like this, and you can select it back out like this. And you can read this value anywhere in the connection: inside a view, inside a stored procedure, etc.

This now gives us a really easy way to tell that security policy who the current user is. Here’s how it all works.

**(click for RLS – security policy)**

First, we need to create what’s called a predicate function. This function will get executed against each row of a result set containing Order records. If the function returns TRUE then the Order gets returned, and if it returns FALSE the order is not returned.

The function assumes that the current user ID has been added to the session context. It accepts as input the user ID associated with an Order.

If the user ID that’s passed in is equal to the value in the session context, then it means that the given order record is owned by the current user, and the function returns TRUE.

If the user ID that’s passed in is NOT equal to the value in the session context, we do a permission lookup for the currently logged in user. If they are allowed to manage all orders, we return TRUE, and otherwise we return FALSE.

You can think of this like a lambda expression that you might pass into a LINQ query, it’s basically a filter function that is used to reduce a result set.

There is one gotcha here: when we’re done, the security policy is going to execute this predicate whenever ANYONE tries to access the Orders table. It’s important that you check to see if the user ID is in the session context or not. If not, don’t filter anything. Otherwise, you won’t be able to see the data in SSMS without setting a user id.

I learned this the hard way. When I was writing the sample code for this section I couldn’t figure out why my test data scripts were successful, but the Orders table was empty. It took me about 30 minutes to realize that my security policy was working flawlessly and was successfully hiding every record in the table. Good times.

Once we have this predicate defined, we need to tell SQL Server where to use it.

**(click for RLS – policy)**

To do that, we create a security policy on the Orders table that contains a FILTER PREDICATE referencing the function we just created.

Once this is in place, any attempt to read from the Orders table will be subject to the access control logic we defined.

The last piece of the puzzle is to actually set the user ID into the session context so that it’s available to the security policy.

**(click for RLS – EF interceptor)**

To do that, we need some sort of hook that we can tap into and run custom code at the start of every database connection.

In Entity Framework, for example, we can create a class that implements the IDbConnectionInterceptor interface and implement the “Opened” method. The code we put here gets run each time Entity Framework opens a new connection.

All we need to do is figure out who the current user is, and then add that value to the session context.

That’s it.

**(click for RLS – clean)**

Once those things are in place, our feature code can be 100% focused on business logic because all of the access control is taking place automatically. Pretty sweet!

This is a very new feature in SQL Server and there are a couple of restrictions that you need to be aware of, so definitely do your own research before you totally replace your existing security code. Based on my preliminary analysis though it looks promising.

# Page-level Authorization

I just showed you three things that you can implement completely in your framework, with no feature level code whatsoever.

In many cases, though, the security concerns can’t be *fully* swept under the covers. Authorization is a good example of why. You can move the code that implements a permission check into the framework, but you still need *something* at the feature level to indicate which permissions are required.

These next two examples are going to show you how to take a declarative approach to those concerns so that you can decouple the implementation of the security check from the declaration that it’s actually needed.

Let’s start with a basic example where we want to ensure that only users with a specific permission are allowed to access a certain MVC endpoint.

## Page-level auth, feature code

If you implement this rule in feature code it will look something like this: somewhere in the body of each page or action you’ll check to see if the user has the necessary permission and, if not, you’ll kick them out.

This is simple, but it results in a lot of duplicative code. If you change how the permission check works, or if you decide you want to do something different than return an HttpUnauthorizedResult, you’re going to have a lot of places to modify.

**(click for Auth – feature #2)**

The only thing about this piece of code that will change between features is the specific permission that is required; the rest of this code is boilerplate, and that makes it a good candidate for being solved in the framework.

## Page-level auth, framework code

To make this a cross-cutting concern, extract the *implementation* of the permission check into an Action Filter attribute, and then provide the *feature-specific* data as an argument to that attribute.

**(click for implementation)**

The implementation is really straightforward. MVC will automatically execute the code at the right time, so the only tricky thing is figuring out what permissions the user has. If your controllers all derive from that base class I keep talking about, then just add a “CurrentUser” property to that base controller and then you can access it with a little bit of casting.

Once we know who the user is, we can enforce the permission check from a centralized place.

# API call authorization

That covers authorization of MVC endpoints. You can do the same thing with your API calls.

Since APIs generally don’t maintain state, each request has to carry with it whatever data is necessary to handle authentication and authorization. One simple way is to use what’s called “bearer tokens”. A bearer token is some secret value, like a password, that’s passed to the API with every request. This is typically provided as a header, but it doesn’t have to be. It can also be provided as a querystring or form argument.

In my system, the secret value that gets passed around is an API Key. An API key is generated by our system and uniquely identifies a user of our system, and each key is associated with one or more permissions stating what it’s allowed to do.

So when an API request comes in, that value tells us both WHO is making the call and what they are allowed to do.

**(click)**

Using those same techniques, we can decorate the API endpoint with an attribute stating which permissions are required, and then handle the authorization in a centralized place.

**(click)**

The difference is that, using this custom API key approach, that attribute needs to handle both authentication AND authorization. The way we do that is to ensure that every API argument class derives from a base class called ApiRequestBase. This base class defines the API key properties that we use for authorization and provides the mechanism for extracting those values from the header or querystring or whatever.

Once we have these things in place, we can both verify that an API key is valid AND that it has the necessary permissions to perform the given API request entirely in framework code. The only thing that appears in feature code is the declaration of which permission is needed.

# Property-level Authorization

In some cases, you might need to apply permission checks at a more granular level than you get with page-level authorization. For instance, certain parts of a page might be hidden or locked based on the user’s permissions, or specific pieces of data might be hidden for certain types of users.

With a little extra effort, you can handle these requirements with framework level code as well.

## Property-level auth, feature code

This is a feature-level code sample from a view model in my sample app. It implements a business rule that a user must have a specific permission in order to see plain-text social security numbers. If the user doesn’t have the permission, the value gets masked for display.

Just like with page-level authorization, this is simple to do in the feature code, but it can result in a lot of duplication. It also requires that we couple the object model to the concept of a user identity, which might be undesirable.

And since this rule applies to *any* feature that displays SSNs, it meets our definition of a cross cutting concern that should be extracted from feature level code.

## Property-level auth, framework code

This is what we’d *like* to do: just put an attribute on the property, declare the necessary permission, and be done with it.

If this worked, it would be awesome. There’s no reference in this class to my Application User object, which is a cleaner design, and it would be really easy to use this approach on multiple properties of multiple classes without duplicating any code.

Unfortunately, this is easier said than done. Putting attributes on MVC actions is easy because the MVC framework provides specific hooks for those attributes to plug into. By default, however, .NET doesn’t provide any hooks for property access. There’s no way to *automatically* run this code whenever someone tries to read the SSN property.

Fortunately, we can use a really neat tool called PostSharp to *create* those hooks.

PostSharp is an Aspect Oriented Programming tool that is specifically designed to handle cross cutting concerns. It works by modifying the IL that is produced by the C# compiler in order to do things that aren’t natively supported in the language.

**(click for PostSharp #1)**

Here’s an example. This is a standard C# property. Behind the scenes, the C# compiler creates a getter method that returns some instance variable, and any code that reads this property is essentially calling this method. The property syntax is just a syntactic sugar over this getter method.

**(click for PostSharp #2)**

Using PostSharp, I can create what’s called a “property interception aspect”. This is basically a piece of code that we want to “inject into” the property.

When I compile the project, the PostSharp engine basically re-writes the getter method, injecting the code from the aspect into it.

Now, any code that is reading that SSN property is actually calling a method that now includes the security code.

**(click for PostSharp interceptor code**)

This is what that interception aspect actually looks like.

First, I created a base class called a “UserAwarePropertyInterceptor”. That class is responsible for talking to the current thread and figuring out who the current user is. This is what allows us to decouple the view model itself from the application user class.

PostSharp gives us this *OnGetValue* method to override. This method basically provides the code that will get injected into that property getter method. This is where I put the permission check and, if the user doesn’t have the necessary privileges, I return a masked value instead of the raw SSN.

**(click for view model code again)**

And as a result of that PostSharp magic, I’m able to do exactly what I want to do. All I need to do is put this “MaskedValue” attribute on a view model property, and the raw value of that property will be automatically hidden if the user lacks the correct permissions.

# Other Postsharp Uses

PostSharp has tons of other uses as well. Basically, it is purpose built to help you globally implement cross-cutting concerns, which makes it useful not only for security but for general application coding tasks as well.

Instead of just masking values, we could implement an encryption scheme as well. The interception aspect could encrypt a value when it’s being stored and decrypt it when it’s being read.

There's a sample implementation of this EncryptedValue attribute in my demo code, and the comments in that code point to a github repo by somebody else that implements a much more robust implementation that deals with key storage and a bunch of other real-world encryption concerns.

**(click for method interception)**

PostSharp can also intercept method calls, not just property access.

This screenshot is taken from the PostSharp website, which has some great tutorials, and it shows a "method boundary" aspect that can inject code just before or just after a method has been executed.

You could apply this aspect to specific methods, or you could apply this at the class level and it will automatically apply to all methods in that class.

This could be used for logging, or caching, or for additional types of authentication or access control. If you've ever wished "man, I wish I could just automatically do <x> whenever <y> happens", then you really need to check out PostSharp.

PostSharp is available on Nuget, but it is a paid tool. There's a free version if your project is small enough, but given the number of things it can do, the cost is very reasonable.

# Auditing and Testing

For the final segment of this session I want to talk about auditing and testing.

Building a secure system is about more than just writing secure code. It’s also about the overall development process and how effectively it helps your team spot and remove vulnerabilities that might otherwise sneak past the developer.

In my experience, exhaustively testing the security of an application is a moving target. You can spend an obscene amount of time and energy doing a full system test, and then your confidence in the results vanishes with the first non-trivial commit that gets pushed. Every time a dev changes existing code there’s the risk that they inadvertently broke an existing security check, or inadvertently introduced a new vulnerability.

One way to mitigate that risk is to automate as much of the security audit and testing as possible, so that you can continually re-run it as the code changes. And in very general terms, this will be easier to do when you’ve isolated your security code into cross cutting concerns.

**(click for endpoint example)**

Here’s an example:

Let’s say we have a large website with lots of different endpoints. It’s reasonable to expect the QA team to validate that every endpoint implements the correct permission check.

The brute force approach would be for QA to perform black-box testing against every single endpoint, verifying that the authentication and authorization checks are properly implemented. This is a really expensive way to go; even if they automate those tests, they still need to spend a lot of effort granting and removing permissions for each test, and automated browser-level tests can be very brittle.

Another approach would be for the dev to write unit tests for those controller actions, but if that’s *all* we do then we’re not giving QA an opportunity to double-check the developer’s work. The testing is less expensive, but at the cost of creating a single point of failure.

In the best of both worlds we’d have QA involved in the verification process, but they’d have a more efficient way of doing so. And, if you’ve implemented your security code as cross-cutting concerns, then you have some interesting options available to you.

**(click for reflection)**

For instance, if you’ve implemented your authorization checks using attributes, then it’s really easy to write a little bit of reflection code to generate a report like this.

This is a snippet of a report from my demo app showing a couple of endpoints. You can see that a couple of them allow public access, others require a login but do not require any specific permissions, and one of them is only accessible to logged in users that also have the ManageOrders permission.

If you publish this report to your QA team, they can be responsible for cross-referencing this data against their security matrix or requirements documents or whatever, so they can maximize the time they spend looking for mistakes and minimize the time they spend fighting with tooling or automating the browser.

**(click for code sample)**

That report was really easy to create. This is basically all it took.

Using reflection, I look for every class that is a type of MVC Controller, I identify all of the public instance methods that are available as endpoints, I ignore some behind-the-scenes stuff added by the MVC framework itself, and then I return an anonymous data structure summarizing those methods.

**(click for sample #2)**

After that, all it takes is a short loop over the data structure to generate the report. Since both the authentication and authorization data are expressed as attributes, it’s easy to use reflection to determine which endpoints require a login or a permission.

If I had implemented those checks as plain-old feature code, rather than using attributes, this would have been a lot harder to do. Granted, with the introduction of Roslyn it’s now *possible* to do static analysis of method bodies themselves so you *could* do something like this without attributes, but it’s certainly *easier* to use reflection and look for the presence of attributes.

**(click for report again)**

In my example here, I’m showing a report of MVC endpoints and the permissions they require. But it would be just as easy to generate a report showing which properties of which classes are using the MaskedValue or EncryptedValue attributes, or whatever else that you’ve implemented as a cross-cutting concern using Attributes.

However, let’s say you have a large application and these reports contain hundreds or even thousands of rows. We’ve made QA’s job *easier* than the alternative, but it’s still far from “easy”. They still need to examine the report, compare it against their “source of truth”, and identify anything that’s been added, removed, or changed.

**(click for Approval Tests)**

The very last thing I’m going to show you today is using a library called ApprovalTests to automate the auditing of this report.

Approval Tests is an alternative way of writing assertions in your tests. It works with everything from MSTest to NUnit to RSpec to Cucumber and a bunch of things in the middle. It is freely available on Nuget.

It’s designed for scenarios where you have an automated test that does some work, but where you need a human being to interpret the results. That’s exactly the scenario we’re talking about with this security audit idea: we can run some code to produce a report showing our endpoints, but we specifically WANT a human being to verify it.

Here’s how you could use this for your security audit.

**(click for Approvals – Step 1)**

First, create a plain-text version of the report that you want to audit. In my demo project I wrote a simple console app that produces a report like this. It’s the same data I just showed you in that HTML page, but in plain text.

**(click for Approvals – Step 2)**

Next, write a unit test that generates that report. Instead of making an assertion, however, call Approvals.Verify() and pass the report text. You can also work with files on disk if you have to, but keeping it string based makes things a little easier.

The Approval Test framework keeps track of the “accepted” state of each test. When this test runs, the framework will compare the new version of the report text against that last known accepted state. If they match, the test passes. If they don’t match, Approval Tests automatically opens a diff tool so that a human being can compare the results and make a decision.

**(click for Approvals – Step 3)**

The very first time that QA runs the test, since there is no “accepted state” yet, the test will launch a diff tool, like you see here. On the left is the report text, and on the right is a blank file.

At this point, the tester would manually verify the report contents. Once they are satisfied that everything matches expectations, they’d merge the left contents into the right file and save them. This is what creates the “accepted state”.

From this point forward, as long as the output of the report doesn’t change, the test will pass without manual intervention.

**(click for Approvals – Step 4 (diff))**

In the future, let’s say I make two changes. I add a new endpoint, and I accidently remove the permission attribute from an existing endpoint.

The next time that QA runs the approval tests, the diff tool will automatically open and will show the differences. In this case, the tester might determine that the new endpoint was expected and is configured properly, but the removal of the permission setting for the existing endpoint was not expected. The tester could then open a security ticket or otherwise contact the developer to discuss.

This is all really simple to set up, *if* you’ve written your security code in a way that lends itself to static analysis.

**(click for auditing difficult)**

Generally speaking, these are the things that are easiest to audit using reflection: attributes, class inheritance, and interface implementation. If you use these techniques to implement your cross cutting concerns, then you’ll find it pretty easy to do security audits with a little bit of custom code.

If you don’t do this, for instance if you implement your security concerns as just some random methods that get called from specific places in your code, then it’s going to be harder to audit.

# Recap

Let’s do a quick recap.

The whole point of this is to identity when you have security code intermingled with your feature code that could be decoupled and pushed into your application framework. Separating them will make your system easier to maintain and more secure.

To do that, leverage “hooks” in the underlying system to run your security code across multiple features. Examples of these hooks are

* MVC action filters
* HTTP Modules
* jQuery AJAX events
* Base classes
* ORM interceptors
* SQL Server security policities
* Etc

Remember that if C# or MVC or your custom application framework don’t provide the hooks that you need, you can use PostSharp to create you own. It’s specifically designed to handle cross cutting concerns and is well worth your time to research.

The holy grail of course is to make features “secure by default” by completely handling the security requirements in a global way. Sometimes though you’ll still need to do *something on* a per-feature basis. The best way to do that is to use some sort of attribute or marker interface to declaratively specify what rules should be applied, and them implement those rules in a consistent part of the processing pipeline.

If you do that, then you can make your life easy for testers by using reflection to generate reports showing which areas or classes implement which security rules. This allows QA to better validate those rules with less effort than brute force, black-box testing. And for bonus points, use the Approval Tests library to further automate that sort of security audit.

## Final slide

My examples of those concepts are in my sample app, which you can get from this link here. This github repo contains my slide deck, my speaker notes, and a fully functional sample of everything I showed you today including:

* Global CSRF defense
* MVC authentication that makes actions private, unless explicitly made public
* Row level security using SQL Server 2016 and Entity Framework
* Permission-based authorization at both the MVC action level, and on specific C# properties using PostSharp
* Auditing using reflection and the Approval Tests library

You can get ahold of me through GitHub, my blog, or on Twitter. I’d love to hear from you, and feel free to send a pull request if you add a technique of your own.

Thank you so much!