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 checks that I perform into some part of my application framework so that they can be automatically applied across all of my features. This lets me keep my brain in “feature mode”, while still delivering 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 you to begin thinking about ways of decoupling them.

**(click)**

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.

**(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 and that’s all I have time to cover. 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.

**(click for “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 exposes the concept of Action Filters that let you run code at specific points in the MVC pipeline

jQuery exposes some global AJAX events that let you run code at the start or end of every single AJAX call that gets made.

ORMs like Entity Framework and NHibernate expose hooks that let you run code when a connection is opened or when a result set is processed.

Tools like PostSharp, which we’ll discuss in a minute, let you create *new* hooks in existing code.

And of course, framework code could also refer to things that you put into base classes that your business objects derive from.

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 for a given page or API.

**(click for “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 busted out my mind-blowing Visio skills to give you this example.

**(click to example)**

Let’s say we have this feature that is an “Order List” page on our website. It gives us a list of orders that the user is allowed to see.

This page has three requirements:

1. First, the user must be logged in. Any unauthenticated users should be redirected to the login screen.
2. Second, a User with the “Manage Orders” permission can see *all* Orders in the database
3. Lastly, users without that permission can only see the Orders they created. Bob can’t see Alice’s data, and vice versa.

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

The naïve approach is to implement those requirements directly within the feature code. If you’re writing an MVC app for instance, you might implement these right in the body of the controller action. This red square represents the security code being added directly to the feature code.

**(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 and they forgot to add those security checks.

**(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 how those requirements were implemented the first time 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, 3 out of 4 parts of this feature implement the security checks, but one doesn’t. Unless your QA department is exhaustively testing *every single endpoint* they could easily overlook this. And even if they are able to test every single endpoint for these rules, it’s going to be enormously expensive to do so. They’re either manually repeating the same test against multiple endpoints, or they’re writing 4 automated tests to do it. Either way, it’s a lot of duplicative and wasteful effort.
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 that implements that change doesn’t realize that the same rules are duplicated in other features, they might end up just one of the features.

Even if that developer searches for all places using the “square” implementation, they might miss the features using 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.

**(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. Every action that calls the method will have a consistent implementation, but nothing prevents a developer from just forgetting to call it. It’s also difficult to audit the codebase to determine which actions call one of the security method and which don’t.

**(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 way the implementation is managed in a single place and it’s also easier to audit, which we’ll talk about later, but still nothing prevents the developer from forgetting to include the attribute.

**(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.

**(click for “show me the codez” 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.

**~~(click for SQL Injection)~~**

~~The “half example” is SQL Injection. This is something I’m assuming you’re all familiar with so I’m not going to spend much time on this, other than to say that it’s a great example of the type of problem that can be solved with a framework-level decision.~~

~~The issue of course is that if you create SQL commands by concatenating untrusted input into your query, you’re at risk. And all you need to do to eliminate that risk is parameterize your queries.~~

**~~(click for example)~~**

~~Just about any modern data access library or framework will do this for you automatically. For example, ORMs like Entity Framework and NHibernate provide a LINQ API that automatically escapes the values and create pamaraterized queries.~~

~~If you pick a data access framework that does this, then your data access code is basically instantly SQL-injection free.~~

**(click for CSRF Transition)**

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 primer.

**(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 redirects the user to the bank’s website using a form POST, 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 already includes some framework level stuff to protect against CSRF. All you need to do is call a helper inside the body of the form, and add an 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 solution is easy and it works well, but it’s not secure by default. It requires the developer to remember these two things or else the form is vulnerable.

Also, this only works if you’re submitting a form. If you’re doing an AJAX POST, you have to manually include the token.

**(click for CSRF – feature (AJAX))**

This is how to make a secure AJAX form post. First, you create a div and call the helper, which renders a hidden form field and sends a cookie to the browser.

Second, you include the value of that hidden field field in your AJAX payload.

The backend doesn’t change, you’d still need to use the [ValidateAntiForgeryToken] attribute on the endpoint.

This is still relatively simple, but it’s error prone. And in this case, this is something that *can* be made 100% secure by default.

**(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 generally do this in my global layout file. 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.

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

To handle AJAX posts, I run a second bit of JQuery that defines a global “prefilter” handler. The prefilter handler provides a hook so that you can modify the AJAX options *before* the request is sent. In this case, we use it to add the CSRF token to every single POST call.

**(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, overriding the OnActionExecuting method, and executing the token validation logic when necessary.

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”.

**(click for Authentication transition)**

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.

**(click)**

In ASPNET MVC, for instance, you can add the [Authorize] attribute to an Action and it will automatically require a valid Forms Authentication token to access that page.

I don’t like using this though, because it represents a “public by default” model where any given resource 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 endpoints to be private unless the dev explicitly makes them public.

You might think that you could just add a [Public] attribute or something to the public endpoints, but that won’t necessarily work. As of IIS 7, it’s possible for ALL requests, even native requests for static resources, to be routed through ASP.NET. We need some way of indicating that certain stylesheets and scripts can be obtained anonymously, but there’s no way to put an Attribute on a static resource.

**(click)**

Implementing a “private by default” model requires two things.

First, I create a base class that all of my controllers will inherit from, and I override the “OnActionExecuting” method. This ensures my code is going to run for every single request.

**(click**)

Secondly, since every request gets routed through this logic, I need to figure out which URLs are allowed to be public.

**(click for Authentication – web config)**

My solution looks like this.

This is a custom section of web.config where I specify a series of regexes representing the URL patterns that are allowed without a login.

Managing this list in a centralized place makes it really easy to maintain, and using regexes gives me a great deal of flexibility in exchange for only a tiny bit of complexity. For example, a single rule lets me expose my entire scripts folder for anonymous access so that I don’t have to modify this file every time I add a new jquery plugin.

**(click for Authentication – code sample v2)**

Going back to my base controller, to figure out if a request requires authentication or not I just compare the path against the regexes in web.config. The implementation of this method is pretty simple and you can get it from my github repo if you’re interested.

Once these things are in place, all a controller needs to do is inherit from this base class and it becomes “private by default”. Every endpoint will require a login unless the endpoint matches one of the patterns in web.config.

At the end of the talk I’ll show a way to generate a report of which MVC controller actions are public and which aren’t, so that this is easy to audit.

**(click for Access Control transition)**

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. Otherwise, if he can guess an order ID, he would be able to modify the URL to directly access it, even if that order wasn’t shown on the list.

**(click for Access Control feature)**

Here’s one way to implement that requirement in feature level code. This feature gets all orders and then, if the user lacks the permission, it filters that result set by user ID.

If we’re enforcing this logic on the List page, we should also enforce it on the Order Detail page. Otherwise the user could manually modify the URL and access someone else’s order.

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

This is what that same requirement looks like on the Order Detail page. In this case we call the GetById method to retrieve the order and then we implement the permission check.

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. This rule might apply to a bunch of places in the code, and without a centralized implementation it’s going to be very difficult to keep them in sync as requirements change over time.

There are two ways that we could reduce that duplication by handling this as a cross-cutting concern.

**(click for Access Control – framework #1)**

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.

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.

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

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.

**(click for RLS graphic)**

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 basically 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.

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. I spent about 30 minutes trying to figure it out before I realized that I’d created a security policy that was so secure, it prevented ANYONE from ever seeing any records from the Orders 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!

**(click for Authorization transition)**

I just showed you a few security concerns that you can implement completely in your framework, with no feature level code whatsoever.

In many cases, though, the business rules governing access to a feature can’t be *fully* swept under the covers. For instance, you can implement a permission check in the framework, but you still need something in the feature code 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.

**(click for Auth – feature)**

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.

**(click for Auth – framework)**

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.

**(click for Property-level transition)**

Page level authorization is great for course-grained control over your app, but sometimes you may need more granular control to further protect specific pieces of sensitive data, such as credit card or social security numbers, that may be displayed within your app.

In many cases, these requirements can also be treated as cross-cutting concerns, with a little bit of effort.

**(click for code sample)**

This piece of code is 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.

Just like with page-level authorization this is fairly simple to do, but it can result in a lot of copy/pasted code that makes it hard to maintain or modify those rules over time. It also requires that we couple the object model to the concept of a user identity, which might be undesirable.

Since this rule applies to *any* feature that displays SSNs, it meets our definition of a cross cutting concern that could be handled in the framework.

**(click for Property Auth – framework)**

And just like with page-level authorization, that means pushing the *implementation* of the check into an attribute and passing the specific permission to that attribute as an argument. A side benefit of this approach is that we might be able to decouple the object model from the application user class, which you’ll see is NOT represented anywhere in this code.

Unlike with page-level authorization, however, there’s no handy place to hook this logic into. There is no generic “OnPropertyAccessed” type filter built into .NET.

However, we can use a library called PostSharp to give us pretty much that exact thing. PostSharp is what’s called an “Aspect Oriented Programming” framework. I won’t go into all the details, but basically it’s a post-processor that runs *after* you compile your code, and it essentially modifies the IL to “weave” code together in interesting ways.

For example, in my demo app I implemented a custom attribute called MaskedValue, which looks like this:

**(click for PostSharp code**)

I’ve created this base class called UserAwarePropertyInterceptor. It basically talks to the current thread, figures out which user is active, and exposes the user data as a class member. This is what allows us to decouple the view model itself from the application user class.

This property interceptor base class itself inherits from a PostSharp class which gives us this *OnGetValue* method to override. This method will get executed every time code tries to read the property that the attribute is attached to. I do the permission check in the body of this method and, if the user doesn’t have the necessary permission, I return a masked value instead of the raw SSN.

Again, the magic here is PostSharp. After I compile my code, the PostSharp engine re-writes the getter for the SSN property and injects a call into this method instead. It’s a little mind-bending at first, but it’s really cool and enables you to do some amazing things.

**(click for Property Auth – framework (dupe))**

In my demo app I’m assigning this attribute to a view model because it made it easier to compare/contrast the different techniques in different areas of my site.

In a production app, you might be able to assign it directly to your domain model like you see here. This way, every view model that you create would inherit the same permission check.

**(click for Encrypt)**

Another cross-cutting concern that you can handle with PostSharp is encryption. I’m not going to show the details right now, but if you look at my sample code you’ll see an example of this Encrypted Value interceptor that automatically encrypts its values when storing them in the database, and automatically decrypts the values when reading them back out.

This can be a great way to securely store sensitive data on disk while still making it super, super easy to get access to that data in the app code.

**(click for auditing)**

For the final segment of this talk 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’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.

**(click for recap)**

To wrap up, I have a quick recap.

First, any time that you have to implement a security check, consider whether it’s truly unique to the feature you’re implementing or whether it applies more broadly across multiple features. If it is a cross cutting concern, extract as much of the implementation as possible into your framework. This ensures that the implementation is consistent across the entire app.

The holy grail is to make features “secure by default” by completely handling the security requirements in a global way. I showed you how to do this with SQL Injection, CSRF defense, and with row-level security in SQL Server 2016.

Sometimes you’ll still need the developer 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. MVC lets you do this with ActionFilters, but you can also use PostSharp to decorate plain-old C# objects and properties with attributes as well. I showed you how to use this approach to mask out sensitive data unless the active user has a specific permission.

Finally, make life easy for your testers. Instead of making them do exhaustive black-box testing, use Reflection to generate reports showing them which areas of the site implement which security rules. This allows them to provide a 2nd line of defense against mis-configuration. And for bonus points, use the Approval Tests library to further automate that sort of security audit.

**(click for close)**

This slide deck, my speaker notes, and a fully functional working sample of every concept I presented today is available on my GitHub page, and if you have any questions you can contact me through my website or through Twitter.

Thank you so much!