Outline:

Intro – 5 min

SQL injection protection via parameterized sql (canonical example)

Output encoding (manual in ASPX, automatic in Razor)

[x] Authentication: “secure by default” vs [Authorize]

[x] Authorization: Permission checks via attributes

[x] Authorization: AOP for masked values

[x] Access control: in data access code

[x] Access control: Row-level security in SQL Server

[x] Anti CSRF tokens

[ ] Testing: static analysis

[x] API bearer token check via Attributes (Web API)

~~[x] AOP for encryption~~

Static analysis

* Controllers w/out permission check
* UI testing?
* HP Fortify

Anti CRSF Tokens

* ASP.NET
* ?

<https://github.com/jkuemerle/EncryptedType>

IL merge?

Build security into the process (case template w/ security section)

OAuth token management in the news recently

Conclusion – 5 min

“Census XML Gateway” = a proxy. “Oracle XML Gateway” (gives external access to ESB)

* Access control in data access code (C#)
* Permission checks via attributes (C#)
* API bearer token check via Attributes (Web API)
* Permission checks via Annotations (Java?)
* Static analysis
  + Find all controllers without a permission check

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, security concerns feel like a distraction to me. They just aren’t interesting because it’s always the same thing over and over again: 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. Etc, etc. When I’m really engaged in a business problem, I want to be “all in” on that problem, and the constant need to implement security requirements, feature after feature, bums me out.

I guess a better way of expressing this concept is that

**(click)**

I hate writing “secure features”. 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 feature code. 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 the next 50 minutes are all about. When you leave here, I want you to recognize the problems and the duplication that occur when you intermingle security concerns with your feature code. I’m going to show you how to remove that duplication so that you can address your security requirements *once* in your application framework and not over and over again in each feature. That’s going to help you write less code, that’s secure by default, that’s easier to audit, and easier to maintain.

.

**(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 3 and a half ways you can make your feature code “secure by default” by handling specific concerns in the framework. Once you implement these framework features you can completely stop thinking about certain types of security issues, and still be protected.

**(click)**

Unfortunately, not everything can be made *fully* secure by default, so I’ll show you how a declarative approach to security concerns results in cleaner, easier code than a naieve imperative approach.

**(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 “Cross Cutting”)**

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

The easiest way to describe a cross-cutting concern is any security requirement that spans multiple features.

An obvious example is SQL Injection: the need to sanitize input before passing it to the database engine is a global requirement, it’s not feature-specific. Other low-level examples are HTML encoding your outputs and preventing cross site request forgery attacks.

Higher level examples can be cross-cutting as well. For instance, you might have a whole group of pages on your site that require the same level of permissions to access them. This isn’t a *global* requirement, but it still spans multiple features or pages.

I have an example that illustrates what can happen when these higher level rules are implemented as feature-level concerns instead of cross-cutting concerns.

**(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 “what makes cross cutting”)**

So what makes a specific security rule a “cross cutting” concern?

The first thing is if it’s orthogonal to your feature-specific business rules. Examples of things like this are SQL Injection and Cross Site Request Forgery protection. There’s nothing feature-specific about the requirement to sanitize input before using it in a SQL query, and this requirement exists for *every* feature, so that’s an obvious candidate for a cross cutting concern.

Your feature-specific logic can be cross-cutting as well, if it applies to multiple features. Access control is a good example; if you have complex rules about when and how Bob is allowed to see Alice’s orders, then those concerns are cross cutting for all features dealing with Orders.

Basically, anything that “cuts across” multiple MVC actions, API calls, page views, etc could be a cross cutting concern.

**(click for “what makes framework”)**

In addition to cross cutting concerns, your framework level code is a great way to address anything that can be made “secure by default”. What I mean by that is that sometimes it’s possible to solve a problem once and have that solution *automatically* apply to all features you write from that point forward.

And finally, if you want to be able to audit your software and generate a report about which features handle security issues in what way, you probably want to look at framework-level solutions. I’ll talk more about this later.

**(click for “show me the codez” transition)**

At this point, you probably want to see some code. To indulge you, I’ve put together a sample app that 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.

All of the screenshots I’ll show you come from that app, which you can get in its entirety from my Github page, which is linked here and also at the end of this deck.

To start, I’m going to show you 3 and a half 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, but it’s a good example of a cross cutting concern solved in the framework.

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 next example of a “secure by default” cross cutting concern deals with #8 on the OWASP Top 10, Cross Site Request Forgery. This requires that we write a little bit of code, but once it’s in place it will protect every feature on the site.

**(click for CSRF diagram)**

In case you’re not familiar with CSRF, here’s a quick primer.

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 a page on the bank’s website. Now, the bad guy can’t actually read any data that comes back, but they CAN try to trick the bank into doing something on the user’s behalf. For example, the malicious site might make a form POST to the bank’s “transfer funds” page, requesting that money be transferred into the attacker’s account.

If the bank website hasn’t been properly secured, then all it’s going to see is a request coming in, from a user with a valid session, requesting a transfer. And if it completes that request, the cross-site request forgery attack is successful.

**(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, for two reasons.

1. First, this assumes a “public by default” model where actions are public unless we explicitly make them private. I tend to work on applications where the vast majority of our pages are private, and only a specific few are public.
2. Second, this attribute only requires a valid Forms Auth token. I tend to store information about the user in session state after they log in, and in some cases it’s possible to have a valid Forms Auth token but have an unpopulated session. This can lead to errors when code assumes that it can get certain data from the session.

**(click)**

I generally flip the script and make my pages protected by default, and only require the developer to take special action if they want a page to be public.

There are two parts to this solution.

First, I create a base class that all of my controllers will inherit from. This is a really useful thing to do for lots of reasons, security one of them. Within this controller I override the “OnActionExecuting” method and put in my global security check.

Secondly, I specifically identify only those endpoints that need to be public. One simple way to do this is to create a custom “[Public]” attribute that identifies the pages that bypass the login requirement.

Once implemented, developers don’t need to think about this any longer. Every new endpoint they create will be secure by default, and they will only need to do anything extra if they specifically want it to be *insecure*.

**(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 using feature level code. This page lists all orders the user is allowed to see. If they have the Manage Orders permission then they can see everything, otherwise they can only see their own orders.

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

This is what that same requirement looks like on the Order Detail page. In this case, we have the same *business rule* being implemented in *two different ways*. One page is applying the requirement as a filter against a list of orders, and the other page is applying the requirement against a specific order. If you think back to the start of my talk, one of these pages was represented by the square, and the other by the diamond. Same business rule, but different implementation.

There are two ways that we can handle this in a cross-cutting fashion, each with their own trade-offs.

**(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.

In this case, I’ve created 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 applies the security check using whatever custom business rules you need. If you call the version that doesn’t, then 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 a scheduled task that runs a maintenance program in the middle of the night. If you can avoid it, I’d recommend NOT having the insecure versions at all. But if you *do* have to provide them, I recommend a naming convention like you see here because it reminds the programmer that they need to provide their own access checks. If I’m a programmer and I type “OrderService.GetById”, it isn’t explicit whether or not security rules have been applied. But if I type “GetByIdInsecure”, that’s a pretty clear reminder that I’m on the hook for access control in my app code.

This approach is better than nothing, and it pushes the access control logic down from the UI layer and into the data access layer. But this is far from “secure by default”; every time you add a new data access method, you’ll need to re-implement the security checks. It also leads to a lot of duplicate code if the security logic is implemented differently in each of these methods.

In some cases, you might be able to push this logic even lower, into the database itself, using a technique called Row Level Security. The idea here is that the database, not the application, determines which rows a given user can see. This eliminates the need for access control in the application code and goes a long way towards making things “secure by default”.

**(click for RLS)**

This is an example of how you could implement Row Level Security using Entity Framework and SQL Server 2016.

In the 2016 version, SQL Server added a new feature called the “session context”. This is basically a key/value collection that’s scoped to the connection and it lets us establish essentially a “global variable” for all queries executed within that connection. Here’s how it works.

First, we create a class that implements the IDbConnectionInterceptor interface. This basically tells Entity Framework that we want to run some custom code whenever a database connection is opened.

In our custom code, if no user is logged in, we don’t do anything. This is important or else we prevent the system from running *any* queries for anonymous users, which is probably bad.

Then, if there IS a logged in user, we execute this stored procedure called sp\_set\_session\_context. This is how we add values to the connection scope.

At this point, the connection gets opened and any query that wants to can access the current user ID.

**(click for RLS – SQL)**

The next thing we need to do is to use that ID to restrict access to specific rows. We do that by creating a predicate function that accepts the user ID associated with an order we want to display. The predicate function returns 1 if the user ID that is passed in is equal to the current user, which would mean the user is viewing an order they own, OR if the current user has permission to view all orders.

Think of this like a predicate function in a LINQ statement. SQL Server is going to apply it against every Order in a result set, and this function is going to indicate whether the current user can view that specific record or not.

**(click for RLS – SQL 2)**

The final piece of the puzzle is to tell SQL Server that it should apply that predicate function to the Orders table, which you do like this.

You have to be a little careful with this because this predicate is applied all the time, even if you’re connected through Management Studio. Make sure to disable the filtering if there’s no active User Id set.

**(click for RLS – clean)**

And here’s what the feature code looks like. It’s 100% focused on the business logic, no security stuff in sight.

The EF interceptor automatically adds the user ID to the connection’s session context, the database automatically applies the predicate function against the Order table, and the predicate automatically filters out any orders that the user isn’t allowed to see. Pretty cool!

**(click for Authorization transition)**

I just showed you 3 and a half examples of ways to implement security checks without touching feature level code at all.

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 you can decouple the implementation of the check from the declaration that the check needs to happen.

Let’s start with a basic example where we need to ensure that only users with a specific permission can access a specific endpoint in the application.

**(click for Auth – feature)**

If you implement this rule as a feature level concern 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.

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

**(click for Auth – framework)**

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

**(click for implementation)**

The implementation is really straightforward. The only tricky thing is figuring out how to determine who the current user is. Generally speaking, I create a single base controller that *every* other controller in my site derives from. I can then add a “CurrentUser” property to that base controller, which I can then access from the attribute code 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 SSN property of my view model. 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.

**(click for auditing)**

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

For the final segment of this talk I want to

There’s another benefit to implementing this with an attribute instead of feature-level code, and that’s auditing.

Let’s say we have a large website with lots of different endpoints. It’s reasonable to expect that someone, at some point, is going to want a summary of which endpoints require which permissions. This might be necessary as part of an annual security audit, or maybe your QA department wants to cross check that list against a set of requirements somewhere.

If you’ve implemented these checks as feature level code then it’s really difficult to generate that type of report. It’s not *impossible*, but difficult.

But if you implement these checks as attributes, it’s a bit simpler. You can use a tool like NDepend to generate a report of every Controller action and the permissions it requires.

**TODO – NDEPEND**

**TODO**