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 we’re talk about 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.

Second, I’m going to show you a sample application that I put together and I’m going to walk you through <x> different refactorings, each one moving a different type of security concern out of feature code and into the framework code.

Lastly, I’m going to <???>.

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. I like to think of this as a patterns talk and not a platform specific talk.

**(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 a concern”)**

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.

**(click for “multiple features”)**

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.

**(click for “secure by default”)**

Another good indicator is if you have the ability to make all features “secure by default”, meaning that developers have to do literally *nothing* to gain the protection. I have three different examples to show you where I’ve baked the security check so completely into the framework that a developer could totally forget about the security requirements and still ship a secure feature. That’s pretty cool.

**(click for “audit”)**

The last indicator that something would make a good cross-cutting concern is if you need to perform routine auditing around it.

For example, you might need to do quarterly security tests against your site, and during those tests you might want the ability to generate a report of all MVC or API endpoints that allow anonymous access versus those that require a logged-in user with specific credentials. Many of the techniques I’m going to show you are designed so that you could leverage static analysis tools to generate a report like that.

**(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 a couple of simple security requirements implemented two different ways: first, as “secure feature” code where the security requirement is handled directly within the business logic, and secondly where the security concern was extracted into some part of the framework.

I’ll be showing you snippets of that code, but you can get the entire application from my Github page, which is linked here and also at the end of this deck.

**(click for A1 – Injection)**

The number 1 vulnerability in the OWASP Top 10 is injection attacks. Technically the list is referring to all types of injection attacks, but I want to talk about SQL Injection specifically.

I’m assuming that you’re all familiar with SQL Injection so I’m going to zip through this really fast, but it’s a really good example of how to solve problems at the framework level and make your code “secure by default”.

And that’s because you can totally eliminate this issue with one little trick. All you need to do is to parameterize your SQL commands. That’s it. If you never concatenate untrusted input with your query, you’re safe.

**(click for code slide)**

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 for expressing queries, and they automatically escape the values and create pamaraterized queries.

Using a data access library or framework makes it really hard for a developer to create injection vulnerabilities, which makes it a good example of the problem solving approach I’m suggestion.

**(click for CSRF Transition)**

My next example is about how to deal with #8 on the OWASP Top 10, Cross Site Request Forgery, by treating it as a cross cutting concern. In this case, we have to write some actual code – there’s no library that can do this automatically.

**(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 Authorization transition)**

Another area that tends to lend itself well to a framework-level approach is Authorization, which is all about ensuring that the current user is allowed to perform the current request. Authorization appears on the OWASP Top 10 multiple times, including at #4 “Insecure Direct Object References” and #7: “function level authorization”.

A common requirement when displaying a page is to perform a security check, and then hide any links on that page that the current user isn’t allowed to perform. If you fail to ALSO perform those same security checks on those specific pages, then a malicious user could hack the URL and navigate to them directly. That’s what these things are referring to.

I’m going to show you two different ways that you can treat this type of Authorization as a cross-cutting concern. Now, unlike with SQL Injection and cross-site request forgery defense, you usually can’t make your authorization concerns *totally* secure by default. And that’s because knowing whether or not a user is allowed to do something is, by definition, feature specific.

However, it IS possible to implement those permission checks with very little feature-level code. Let’s take a pretty basic example in which we want to require a specific permission in order to access a specific page or action.

**(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, and that makes it a good candidate for pushing this code into the framework.

**(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.

Yes, the developer must remember to include the permission, but there’s really no way around that.

**(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 makes it available to the attributes with a little bit of casting.

**(click for auditing)**

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**

**(click for A6 transition)**

Page level authorization is great for course-grained control over your app, but sometimes you may need more granular control. That brings us to #6 on the top 10, “sensitive data exposure”. This is all about ensuring that sensitive data like social security and credit card numbers are sufficiently protected.

In many cases, these requirements can also be treated as cross-cutting concerns.

**(click for code sample)**

For example, this piece of code is from a view model in my sample app. It implements a business rule that states 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. Since this rule applies to *any* feature that displays SSNs, it meets our definition of a cross cutting concern that benefits from being handled in the framework.

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

And just like with page-level authorization, that means pushing the *implementation* into an attribute and passing the specific permission to that attribute as an argument.

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 the “weave” code together in interesting ways.

For example, this is what the implementation of the MaskedValue attribute looks like

**(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.

My base class 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, it re-writes the getter for the SSN property and injects a call into this method instead. It’s pretty cool stuff.

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

In my demo app I’m assigning this attribute to a view model because it makes it easier to compare/contrast the different techniques, but you could also 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. Once you do this, every single view model becomes “secure by default”.

Also, just like with the controller attribute, using a property access attribute makes it easier to audit this rule. You could use NDepend to run a report of every property, belonging to a domain model, that is protected in this way, and by which permissions.

**(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 Access Control)**

Another common cross-cutting concern is access control, which is the mechanism through which we prevent a user from accessing data they shouldn’t.

The way I treat it here touches multiple items on the OWASP Top 10. This is similar to authorization, but in general I tend to think of authorization like “is Bob allowed to access the Manage Orders page at all”, and access control is like “is Bob allowed to view the *specific order* he’s requested”. This

**(click for Access Control feature)**

Here’s an example of a page that 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.

This type of requirement tends to cut across multiple features. If we’re restricting what items show up in the list, then we probably want to apply that same restriction on the details page. Otherwise the user might start hacking around with the URL and get access to something they shouldn’t.

**(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.

This is 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. If you call the version that doesn’t, then you’re responsible for doing your own access control.

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. You can do similar things with other ORMs, but this example uses brand new feature in SQL Server 2016 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!