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. The simplest would be to extract this logic into a helper method and call it from all 4 controller actions or something. That certainly helps with consistency, but nothing prevents a developer from just forgetting to call that helper in their feature.

As much as possible, I want my features to be *secure by default*, which means that I want them to automatically inherit the appropriate security checks so that a developer can almost completely forget about security and still ship a secure feature. That isn’t always possible, but you can do a lot more than you probably expect.

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

To show you some real code I’m going to share snippets from a sample app that I put together for this talk. The source code is all up on GitHub and linked at the end of this slide deck.

In this demo app I created a very simple set of features, very similar to that example I just walked you through. There’s a page that shows a list of orders, a page for drilling into the order details, and a page for making changes to the order. There are also a couple of simple security requirements.

I set up three side-by-side implementations of that feature set. The first one doesn’t do any security checking at all, it’s just wide open. The second implementation handles each security requirement with some feature-level code, and the third implementation pushes everything into various parts of the application framework. I set it up to be really easy to navigate between the three different implementations in case you want to play around with it on your own.

I’m going to use that demo app to demonstrate a number of cross cutting concerns such as:

1. Authentication, which prevents anonymous users from accessing a feature
2. Feature-level authorization, which ensures the user is allowed to access a feature
3. Access control, which ensures that Bob can’t view Alice’s data
4. Permission-driven access to sensitive data, such as restricting access to SSNs
5. CSRF defense, which prevents Malicious Website A from exploiting a user’s active session on Protected Website B to do something nefarious

Everything I’m going to show you has been pared down so that it’s easily digestible during this session, but I’m currently using many of these techniques in production today so it’s absolutely possible to scale them up to real-world scenarios.

**ANTI CSRF**

Normally requires two things:

* Html.AntiForgeryToken() inside the form (creates hidden field and sets cookie)
* [ValidateAntiForgeryToken] on the action

Instead, tweak framework to make that automatic