Preparing for a security assessment

Your team has been working on a web application for several sprints. They are now preparing for the first official release to the production environment, where the end-users will start registering accounts and interacting with their data. The user stories are implemented, tested, and ready to be deployed. There's just one last hurdle to overcome: the security assessment.

Author Wesley Cabus

A team of security experts will review your web application from top to bottom, trying to abuse the application to gain access to data they're not supposed to see or - even worse - to the filesystem where the web application is being hosted. But don't panic! Let's walk through some steps your team can – and should – take to prepare for the security assessment.

Gather information about your system

The first step is to know everything about your web application, the actors, and the infrastructure it's using or interfacing with. This means you'll probably need to involve additional people from various divisions to answer all the questions: the development team, infrastructure, DevOps, architects, and someone from the security reviewers.

Here are some questions to get you started:

- Which frameworks are we using in our code (.NET, Java, React, Xamarin, ...) and which dependencies do we have (NuGet, Maven, NPM, ...)? Are we using the latest versions where possible? Do we know why some versions have to stay behind? Are there alternatives?
- How are our applications being configured at runtime? In the case of static files, are these accessible from the outside world? Are secrets being protected during the deployment process or are they visible at any time?
- Which services are we interacting with, both internally and externally? How are their interactions protected? Think about credentials, certificates, firewall rules, explicitly allowed IP address ranges, ...
- Where are our web application components being hosted and how is the environment configured? Are we using a virtual network, are web servers configured to use the latest TLS version or not? Do we have additional protection in place around our components, like a firewall and/or API management layer?
- If your application has multiple roles/ rights per user or action, do you have a clear overview per resource, action, and user type or role in which actions should be allowed or rejected?

Create a threat model

After gathering a lot of information, it's time to visualize the system and highlight interactions between all components. There are several threat modeling tools available to assist you in this process. One of the oldest tools is the Microsoft Threat Modeling Tool[1], which focuses on the Microsoft technology stack and is freely available as a Windows client at.

Whichever threat modeling tool you end up using, they all have in common making it easier to raise potential vulnerabilities or risks for every interaction in the model. For example, if you add an interaction between a web server and a SQL database, one of the identified risks is: "An adversary can gain unauthorized access to the SQL database due to weak network security configuration." In Azure, this could mean that you've configured the SQL database to allow access from all networks or even from all Azure

[1] https://www.microsoft.com/en-us/securityengineering/sdl/threatmodeling

services. That's a risk as this option also includes Azure services managed by other Azure customers.

In the following screenshot, you can see an example threat model for a web application which communicates directly with a SQL server instance. Azure Traffic Manager is placed in front of the Azure App Service to redirect traffic coming from a user's browser to multiple instances of the App Service. For the sake of brevity, these multiple instances have been left out of the diagram. Between every component or actor in the diagram, request and response flows have been added to show how the components interact with each other:

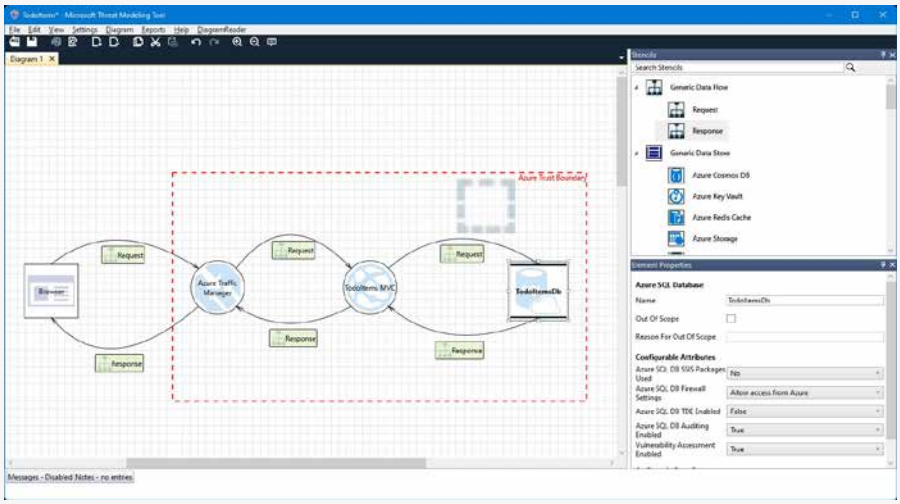
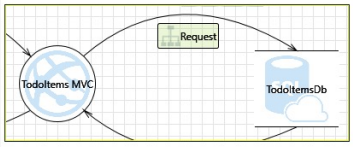


Figure 1. Designing a threat model

After designing the threat model, you can generate a report using Microsoft's Threat Modeling Tool. This report will analyze all drawn interactions and, depending on the configured attributes, will create a list of potential risks per interaction or component, as shown in the screenshot below.

Interaction: Request



1. An adversary can gain unauthorized access to Azure SQL database due to weak account policy [State: Not Started] [Priority: High]

Category:	Elevation of Privileges
Description:	Due to poorly configured account policies, adversary can launch brute force attacks on TodoItemsDb
Justification:	<no mitigation provided>
Possible Mitigation(s):	When possible use Azure Active Directory Authentication for connecting to SQL Database. Refer: th10c">https://aka.ms/tmt-th10c
SDL Phase:	Implementation

Figure 2. Example of an identified risk

The threat model report, as shown above, highlights the request between the TodoItems MVC app service and the TodoItemsDb SQL database and has identified a potential risk:
An adversary can gain unauthorized access to Azure SQL database due to weak account policy.

Every identified risk should be sized and discussed by the assessment team to decide on the next steps. When sizing a risk, think about the impact an exploited risk has, for example, using the DREAD model:

- Damage: how bad would the exploit be?
- Reproducibility: how easy is it to reproduce the exploit?
- Exploitability: how much work is it to execute the exploit, to make it work?
- Affected users: how many people would be impacted?
- Discoverability: how easy is it to discover the threat?

The last letter of DREAD, discoverability, needs to be treated with caution however: it's easy to dismiss a risk as being difficult to discover, for example, when configuration files containing secrets can't be discovered because the web server doesn't list directory contents. But when an attacker knows your application uses a specific framework which expects a file called "appsecrets.xml" to be present at the path "/config" and they can still access that file, that could potentially turn a low risk into a very big issue.

You have gathered information, built a model, and sized the risks. But before letting the security reviewers examine your new web application, why not go the extra mile and check if there aren't some points you can improve already?

Hardening the application

In our example diagram, we have a SQL Server and Azure App Service running within an Azure trust boundary network: all services within this boundary are freely able to communicate with each other. This does not however limit access to our specific resource group, but also allows any other Azure-hosted service to attempt and access the resources from our application.

To prevent unwanted access, we can configure a virtual network in Azure. In this virtual network, we add the App Service and SQL database, and only allow Azure Traffic Manager to access the App Service from outside the virtual network. This ensures that only the Azure App Service instances can access the SQL database.

HTTPS all the things!

Let's start with the most basic rule that everyone should be applying to their web applications from day one: use HTTPS everywhere, even during development. If it means creating a self-signed local certificate for your local development environment, so be it, but use HTTPS from the get-go.

This includes containerized applications as well: please refrain from only protecting the external endpoints and then switching to HTTP traffic inside

the container network! If you focus only on the outside layer being protected, then the chances are high that this methodology will make its way into the production environment. And if someone with malicious intent manages to get their own container deployed into your network, then it's game over. But to be honest, if that would happen, there are probably other systems that need your immediate attention as well.

Add or remove HTTP headers

Every web application should respond with the correct set of HTTP headers.

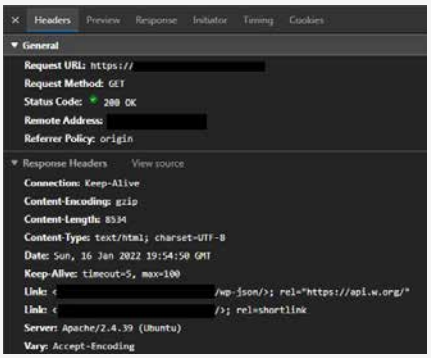


Figure 3. Example HTTP request, showing the response headers

However, most web servers and application frameworks add some information in these HTTP headers that immediately divulge what server or framework is running on that system. Combining this data could be enough for someone to launch a specific attack just by looking up some CVE (Common Vulnerability and Exposure).

When visiting a website, this was the response my browser received, which tells me three important things:

- The web server is running Apache version 2.4.39, as the Server HTTP header tells us. At this moment, the latest release is 2.4.52, and the most recent update includes two fixes for CVE's.
- The web server runs Ubuntu Linux, also divulged by the same Server HTTP header. While we don't get a version number, the combination of Apache and Ubuntu could lead us to specific vulnerabilities.
- The website uses WordPress. This is a bit trickier to find out, but if you look at the first Link header closely, you'll see wp-json and a reference[2]. While this or any other header doesn't give us the version number, that's easily located in the page's source. In this case, the website is using WordPress version 5.2.3, while 5.8.3 is the current latest release.

With only two HTTP headers, which seem completely harmless, and a bit of digging into the returned web page, we've received enough information to start searching for potential ways to gain control over this website or even the server. If this web server had been configured not to send the Server HTTP header, my attack vector would have been reduced significantly.

[2] https://api.w.org

Some application frameworks tend to add their own HTTP header, X-Powered-By, to announce that a framework is being used, often including a version number as well, for example:

```
X-Powered-By: PHP/5.4.0
```

This information is a goldmine for hackers! Do yourself a favor and remove this header from every HTTP response as well.

Other HTTP headers, however, are worth adding to your application to make your web applications more secure:

- **Strict-Transport-Security:** this header tells the browser that your website will always support HTTPS. You can also preload this information by announcing your website[3]. This allows browsers to immediately redirect visitors to HTTPS, even if they've never visited your website before.
- **Content-Security-Policy:** in short, this header is used to specify which sources are allowed or denied to be used by the web application. Think about CSS styles, fonts, JavaScript, but also AJAX requests, hosting framed content or being hosted as a frame, etc. However, setting up a good CSP[4] (Content-Security-Policy) header[5] is a daunting task and will most likely also require rewriting parts of the application if it's heavily using JavaScript. Whatever you do, do not take the easy route and specify "unsafe-inline" for the script-src directive because that would open up your web application to any JavaScript to run.

Is your cookie jar secure?

Server-side web applications use cookies to store if a visitor is authenticated, to keep track of their shopping cart, language preference and many other things. But are your cookies properly secured?

Cookies should be marked **HttpOnly** and Secure, and should have the correct SameSite policy applied to them:

- **HttpOnly** prevents cookies from being used in JavaScript. A language preference cookie doesn't need HttpOnly to be set, so that a script could properly format a date value using the correct locale. But JavaScript should never have access to authentication cookie values, so these cookies should definitely be marked as HttpOnly.
- **Secure** indicates that the cookie will only be sent over HTTPS connections.
- **SameSite**[6] defines or limits how your cookies can flow. The most strict setting limits cookies to only be sent when the user is interacting within the website, while the least secure setting will allow cookies to also be sent when clicking on a link from within an email, for example.

And, just as with the **Server** and **X-Powered-By** HTTP headers, some cookies names include a clear indication of the framework being used by the web application. For example, when using cookie-based authentication in ASP.NET Core, you'll see a cookie appearing with the name ".AspNetCore.Cookies". It's very easy to reconfigure these cookies and rename them, which is a quick win to make it less obvious to potential attackers what framework your web application is built with.

And many more...

There are still a lot of application hardening steps you can take depending on your architecture and the functionality of the web application:

- Parsing XML can potentially load external data during the parsing process;
- Uploaded files could contain viruses, or uploads could attempt to overwrite data from other users or the system;
- Each data store type has a form of injection attacks, so make sure you're correctly parameterizing your queries;
- Is your containerized application using a well-known base image or one you found somewhere randomly? Are you applying patches yourself? Then keep them up-to-date as well.

Conclusion

By gathering knowledge about your system and creating a threat model, it can become clear where to focus your efforts as a team to make your web application more secure, even before the actual security assessment starts. And if you go the extra mile and already implement some additional fixes, then the security team will most likely be happily surprised and challenged to test even more thoroughly.

In any case, you're going to be better prepared to face the security assessment, and together, you'll be able to deliver a more secure solution. </>

Wesley Cabus
Coding Architect

xpirit.com/wesley

[3] https://hstspreload.org
[4] https://content-security-policy.com/
[5] https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy
[6] https://web.dev/samesite-cookies-explained/