EPiC Series in Computing

Volume 58, 2019, Pages 317–324

Proceedings of 34th International Conference on Computers and Their Applications

Study of Secured Full-Stack Web Development

Ziping Liu[1] and Bidyut Gupta[2]

[1] Southeast Missouri State University, Cape Girardeau, MO, U. S. A.

[2] Southern Illinois University at Carbondale, Carbondale, IL, U. S. A.

zliu@semo.edu, bidyut@cs.siu.edu

Abstract

In this paper, we reviewed the tiered architecture and MVC pattern for web development. We also discussed common vulnerabilities and threats in web applications. In order to better understand how to develop a secured web application, we furthermore examined best practices from Angular and ASP.NET core frameworks as well as sample codes for secured web apps.

1 Introduction

According to (IBM, 2018), in 2017, more than 2.9 billion records were leaked from publicly disclosed incidents. Even though the number of breached records dropped by nearly 25 percent, ransomware attacks saw growing and they cost companies more than eight billion dollars globally. In (Symantec, 2018), it also stated that in 2017, ransomware attacks such as WannaCry and Petya/NotPetya attacks made headlines and remained a major threat. Furthermore, the attackers never ceased to target Internet applications using longstanding techniques such as SQL Injection, Cross-Site Scripting, Cross-Site Request Forgery.

Normally, in network security there are five aspects to be considered: availability, confidentiality, integrity, authentication and non-repudiation. As pointed out in (Liu, Z., & Gupta, B., 2016), for the de facto TCP/IP internet layered architecture, there should be appropriate security measures applied at each layer. In the proposed holistic framework for Internet security (Liu, Z., & Gupta, B., 2016), a standalone application security design is crucial in the framework. Among OWASP listed attack categories, a number of them are the results of loopholes in application coding (OWSAP, 2015), hence it is necessary to study secured design in web development so that vulnerabilities from cyber-attacks can be reduced to minimum.

The most commonly adopted architecture for web development is the presentation-logic-data three tier architecture. Coupled with the three-tier architecture is the heavy use of Model-View-Controller (MVC) pattern for the implementation. In the following sections, we will first discuss web development architecture and design pattern in section 2. Then in section 3 we will discuss common vulnerabilities and threats exposed on web. Next, we will study how to fulfill a secured full-stack web development in section 4. And finally, in section 5 we will conclude the paper.

2 Full Stack Web Development

Web development normally involves both client-side and server-side, which is often referred to as full-stack web development. Among them, client-side is also called front-end and server-side is called back-end. As shown in figure 1, front-end most of times the program is written in HTML/CSS/Javascript, and in the last few years Angular, Vue and React have emerged as front runner frameworks. For back-end development, it can be programmed with Java, C#, node.js and PHP, and data server can either be SQL based such as MySQL or NoSQL based such as MongDB.

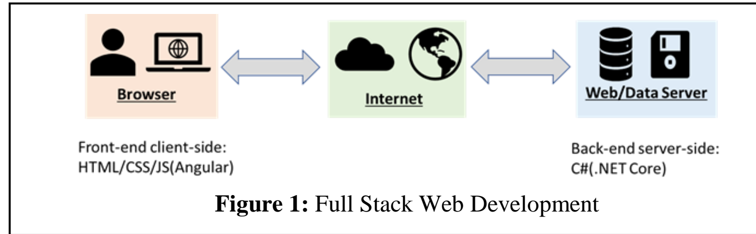

Figure 1: Full Stack Web Development

2.1 Three-tier Web Architecture

In either the mobile-first web design or responsive web design, the most commonly adopted architecture for web development is the presentation-logic-data three-tier architecture, as shown in figure 2. Among the three tiers(Wikipedia), the top tier is the presentation tier, and its functionality is to display web content and provide user interactions. Logic tier is a middle tier between presentation tier and data tier. It conducts business logic operations via processing and moving data between two surrounding layers. Data tier is at the bottom layer where data stored in a database or file system can be pulled/retrieved, and then the information is passed back to logic tier for processing. Information flow occurs two-way, users can push data from presentation tier to logic tier, and then to data tier for storage.

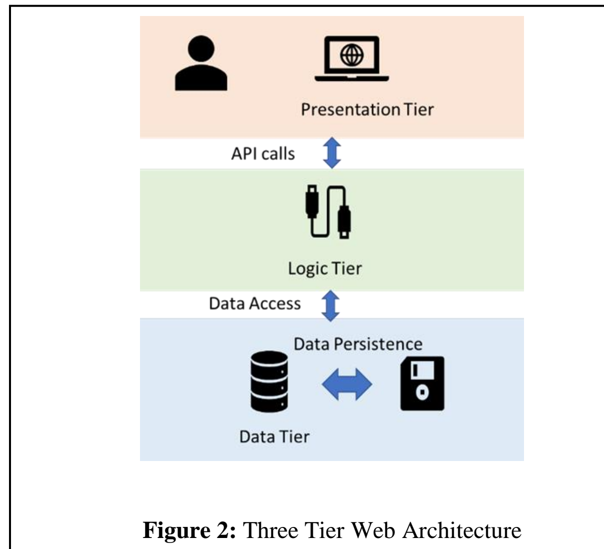

Figure 2: Three Tier Web Architecture

2.2 Model-View-Controller Pattern

To implement the three-tier architecture, Model-View-Controller (MVC) pattern is heavily adopted. As shown in figure 3, there are three components in MVC (chrome), and they are model, view and controller. Model defines data structures used for web application such as users and products. View defines user interface such as web layout and it doesn't know model neither directly interacts with model. While controller serves a bridge between model and view, it processes application requests from view and then forwards them to model, it also carries updates from model to view. In MVC each component can maintain its independence, which offers flexibilities for developer community to select the most suitable frameworks/programming languages for the front-end and the back-end. For example, one of the popular picks in industry is Angular front-end plus ASP.NET core framework using C# programming language for back-end.

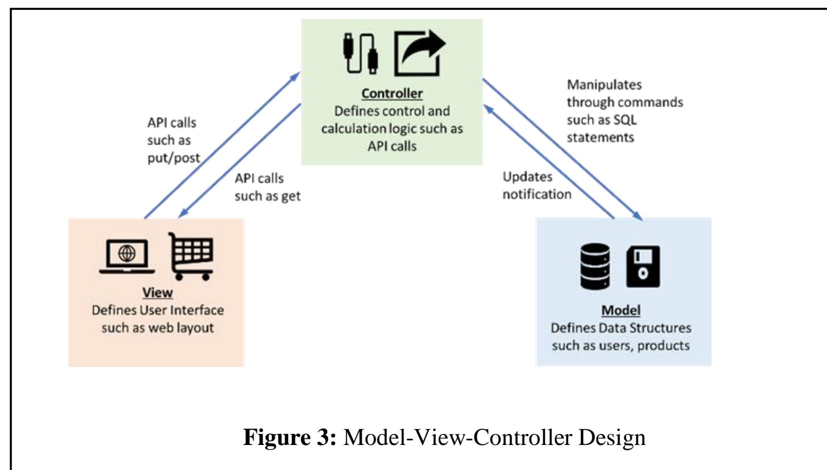

Figure 3: Model-View-Controller Design

3 Web Application Threats

In (IBM, 2018), one of the top enterprise network attacks is injection attack. Among the injection attacks, most are botnet-based command injection (CMDi). Furthermore, (IBM, 2018) stated that publicly reported financial breaches in 2017 affected a major US credit reporting firm and may have impacted more than 145 million people. And the cause of the data breach was an unpatched web application vulnerability which led to the unauthorized access of highly sensitive information.

In (OWSAP, 2018), Injection, Broken Authentication and Session Management, Sensitive Data Exposure, XML External Entities (XXE), Broken Access Control, Security Misconfiguration, Cross-Site Scripting (XSS), Insecure Deserialization, Using Components with Known Vulnerabilities, and Insufficient Logging&Monitoring are listed as 2017 OWASP Top Ten web application threats. The command injection (CMDi) attack reported in (IBM, 2018) is an attack (OWSAP, 2018 May) which may execute arbitrary commands on the host operating system. When a web application passes unsafe user supplied data (forms, cookies, HTTP headers etc.) to a system shell, the attacker-supplied operating system commands can be executed if the application is vulnerable. As shown in figure 4, if front-end lacks input validation and sanitation, malicious data will then be passed to back-end. If back-end lacks further security cross examination, the injection attack will then be triggered on the server side.

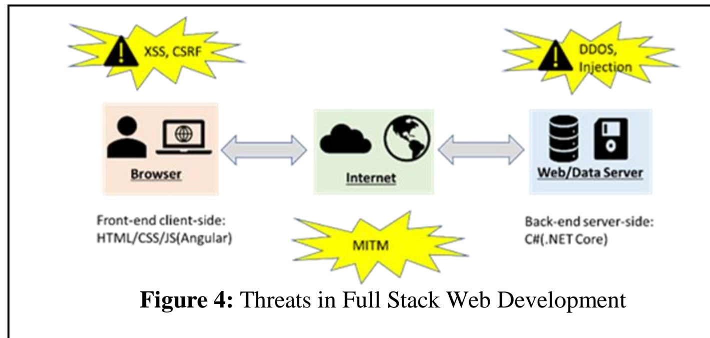

Figure 4: Threats in Full Stack Web Development

SQL Injection (SQLi) (OWSAP, 2016) is conducted by inserting a SQL query via the browser input to the application. As shown in figure 4, the malicious query will further be passed to data server to conduct illegal SQL commands. Cross-Site Scripting (XSS) (OWSAP, 2018) can occur with any user browser supporting scripting and doesn't validate it. Web server directly publishes output from its user input without validating or encoding it. Malicious scripts are injected into otherwise benign and trusted web sites. Cross-Site Request Forgery (CSRF) (OWSAP, 2018) can occur if server lacks authentication of state-changing requests. The identity and privileges of the victim are inherited by the attacker to perform an undesired function on the victim's behalf. And users of a web application may be tricked into executing actions of the attacker's choosing such as transferring funds, changing their email address, and so forth.

4 Secured Full-stack Web Development

In (Liu, Z., & Gupta, B., 2016), we proposed a holistic framework for Internet security based on the analysis of cyber threats and furthermore argued that the standalone application security design is the basic building block. To prevent threats posed by vulnerability issues of front-end software and back-end software from entering the web application ecosystem, full-stack web development should follow secured software guidelines. As shown in figure 5, various security measures can be applied to achieve the goal. And the approaches to exercise the measures can be found in table 1.

If a web server deploys HTTPS for its website, it can eliminate Man-in-the-Middle threat in most cases. HTTPS also encrypts all the data in transfer such that the possibility of eavesdropping and the potential of data stolen can be removed. However, HTTPS is still venerable to CSRF attack, and it cannot replace the security measures for front-end and back-end.

ATTACK	SECURITY MEASURES
CMDI	avoid calling OS commands directly; escape values added to OS commands specific to each OS; parametrization in conjunction with Input Validation (OWSAP, 2017)
SQLI	Use of Prepared Statements (Parameterized Queries); Use of Stored Procedures, Option; and Escaping all User Supplied Input (OWSAP, 2016)
XSS	perform the appropriate validation and escaping for the output on the server-side(OWSAP, 2018)
CSRF	use challenge token for each session (OWSAP, 2018)

Table 1: Security Measures for Common Attacks

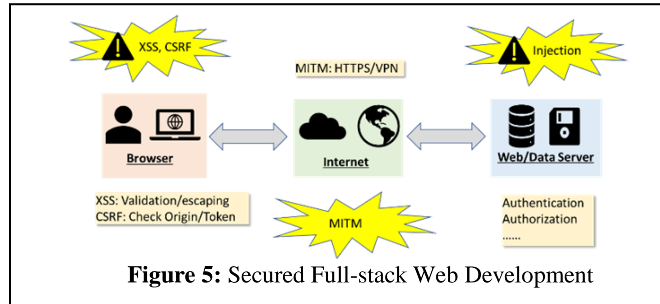

Figure 5: Secured Full-stack Web Development

In the following sections, we will study security best practices on web development both from Angular framework for front-end and from ASP.NET Core framework for back-end.

4.1 Protections from Front-end

As a front-end framework, Angular has built-in protections against common web-application vulnerabilities and attacks (Google), such as XSS, CSRF and XSSI. As shown in table 2, recommended Angular best practices can provide built-in protections. For example, it is recommended to use offline template compiler to prevent attacker's input entering source code template, which is deemed to be trusted. Certain application-level security, such as authentication and authorization, Angular leaves it to the back-end. However, Angular's HttpClient library also has support for the client-facing end for CSRF, which can be found in figure 6.

Angular recommendation	*Built-in protection provided*
DomSanitizer.sanitize method	Built-in Angular sanitization
Use the offline template compiler	Content Security Policy
Angular's HttpClient library	Prevent two common HTTP vulnerabilities, cross-site request forgery (CSRF or XSRF) and cross-site script inclusion (XSSI)

Table 2: Angular Built-in Protections

```
app.module.ts
// ...
@NgModule({
  declarations: [   // ... ],
  imports: [   // ...
    HttpClientModule,
    HttpClientXsrfModule.withOptions({
      cookieName: 'XSRF-TOKEN',
      headerName: 'X-CSRF-TOKEN'   }),
    // other imports ],
  //...})
```

```
form = new FormGroup({ email: new FormControl("", [Validators.required, Validators.pattern('[a-zA-z0-9_\.]+@[a-zA-Z]+\.[a-zA-Z]+')]), password: new FormControl("", [Validators.required, Validators.pattern('^(?=.*[0-9])(?=.*[a-z])(?=.*[A-Z]).{8,}$')]) });

<div *ngIf="email.errors"> <div class="alert alert-danger" *ngIf="email.errors.pattern"> The email is not valid </div> </div>
<div *ngIf="password.errors"> <div class="alert alert-danger" *ngIf="password.errors.pattern"> The password is not valid </div> </div>
```

Figure 6: Example XSRF protection and form validation

Even though Angular has built-in sanitization, form validation can offer adds-on front-door security checkpoint. For example (Kolev, K., 2018), when we create the form control objects, we can assign all the validators as shown in figure 6.

4.2 Back-end ASP.NET Core Security

As one of web application frameworks, ASP.NET Core can scaffold back-end services for web API, identity and database. It also supports security features such as authentication, authorization, data protection, SSL enforcement, app secrets, anti-request forgery protection, and CORS management (Addie, S., Lowell, C. & Appel, R., 2018). To configure antiforgery features with IAntiforgery, the following can be done as shown in figure 7: request antiforgery in the Configure method of the Startup class; require antiforgery validation with ValidateAntiForgeryToken set to individual action, controller, or globally; use AutoValidateAntiforgeryToken broadly.

```
var tokens = antiforgery.GetAndStoreTokens(context);
context.Response.Cookies.Append("XSRF-TOKEN", tokens.RequestToken, new CookieOptions() { HttpOnly = false });

[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> RemoveLogin(RemoveLoginViewModel account){...}

services.AddMvc(options =>
options.Filters.Add(new AutoValidateAntiforgeryTokenAttribute()));

[Authorize]
[AutoValidateAntiforgeryToken]
public class ManageController : Controller{
```

Figure 7: Anti fogery in Back-end

To prevent open redirect attacks, as shown in figure 8 the LocalRedirect helper method can be used and use the IsLocalUrl method to test URLs before redirecting:

```
public IActionResult SomeAction(string redirectUrl)
{   return LocalRedirect(redirectUrl);   }

private IActionResult RedirectToLocal(string returnUrl)
{   if (Url.IsLocalUrl(returnUrl))   {      return Redirect(returnUrl);   }
    else   {      return RedirectToAction(nameof(HomeController.Index), "Home");   }}
```

Figure 8: Prevent Open Redirect Attacks

To override browser's default same-origin policy, ASP.NET core allows to enable CORS(Cross Origin Resource Sharing) for specified cross-origin requests through applying CORS policies globally to the app as well as applying CORS policies per action or per controller.

To implement authentication, ASP.NET Core Identity membership system can be used to add login functionality and then use IdentityServer4 to secure the app. IdentityServer4 is a framework used in ASP.NET Core for authentication and it uses OpenID Connect and OAuth 2.0. OAuth uses access tokens to offer identity proof, but it does not specify what format tokens should take, and JWT (JSON Web Token) can be a token choice. Figure 19 shows how to issue a JSON web token in ASP.NET Core back-end and how to retrieve the token in Angular front-end after user's credentials are validated (Spasojevic, M., 2018).

```
[Route("api/auth")]
public class AuthController : Controller
{// GET api/values
[HttpPost, Route("login")]
public IActionResult Login([FromBody]LoginModel user){//validate user…
var tokeOptions = new JwtSecurityToken(
issuer: " https://localhost:44313",
audience: " https://localhost:44313",
claims: new List<Claim>(),
expires: DateTime.Now.AddMinutes(5),
signingCredentials: signinCredentials);
var tokenString = new JwtSecurityTokenHandler().WriteToken(tokeOptions);
return Ok(new { Token = tokenString });}

export class LoginComponent {
  invalidLogin: boolean;
  constructor(private router: Router, private http: HttpClient) { }
  login(form: NgForm) {
   let credentials = JSON.stringify(form.value);     this.http.post("https://localhost:44313/api/Auth/login",
credentials, {
     headers: new HttpHeaders({
       "Content-Type": "application/json"})
    }).subscribe(response => {
     let token = (<any>response).token;
     localStorage.setItem("jwt", token);
     this.invalidLogin = false;
     this.router.navigate(["/"]);   }, err => {
     this.invalidLogin = true;   }); }}
```

Figure 9: Authentication using JWT in ASP.NET Core and Angular

5 Summary

With more and more apps are web-based, it is necessary to study secured design in web development so that vulnerabilities from cyber-attacks can be minimized. In this paper, we reviewed the tiered architecture and MVC pattern for web development. We also discussed common vulnerabilities and threats in web applications. In order to better understand how to develop secured web applications, we examined the best practices recommended from Angular and ASP.NET core frameworks. Furthermore, we also studied sample programming code snippets for secured web design. It is demonstrated that implementing security measures at both front-end and back-end can be an effective approach to achieve the goal of a secured design.

References

IBM Security. (2018, March). *IBM X-Force Threat Intelligence Index 2018*. Retrieved from https://www-01.ibm.com/common/ssi/cgi-bin/ssialias?htmlfid=77014377USEN

Symantec. (2018, March) *Internet Security Threat Report*. Retrieved from https://www.symantec.com/content/dam/symantec/docs/reports/istr-23-2018-en.pdf

Liu, Z., & Gupta, B.(2016) A Multifaceted Assay on Cybersecurity: The Concerted Effort to Thwart Threats, *Proceedings of 31st International Conference on Computers and Their Applications(CATA)* (pp 123 - 129). ISCA.

Open Web Application Security Project Foundation. (2015, July). *OWSAP Category: Attack.* Retrieved from https://www.owasp.org/index.php/Category:Attack

Wikipedia. (n.d.). *Multitier Architecture.* Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Multitier_architecture

Developer.chrome.com. (n.d.). *MVC architecture* Retrieved from https://developer.chrome.com/apps/app_frameworks

Open Web Application Security Project Foundation. (2018, March). *OWSAP Top 10-2017 Top 10*. Retrieved from https://www.owasp.org/index.php/Top_10-2017_Top_10

Open Web Application Security Project Foundation. (2018, May). *Command Injection*. Retrieved from https://www.owasp.org/index.php/Command_Injection

Open Web Application Security Project Foundation. (2017, November). OS Command Injection Defense Cheat Sheet. Retrieved from https://www.owasp.org/index.php/OS_Command_ Injection_Defense_Cheat_Sheet

Open Web Application Security Project Foundation. (2016, June). *SQL Injection*. Retrieved from https://www.owasp.org/index.php/SQL_Injection

Open Web Application Security Project Foundation. (2018, June). *Cross-site Scripting (XSS).* Retrieved from https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)

Open Web Application Security Project Foundation. (2018, March). *Cross-Site Request Forgery (CSRF).* Retrieved from https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)

Google. (n.d.). *Security*. Retrieved from https://angular.io/guide/security

Kolev, K. (2018, March). *By Quickly Create Simple Yet Powerful Angular Forms*. Retrieved from https://www.sitepoint.com/angular-forms/

Addie, S., Lowell, C. & Appel, R. (2018, October). *Overview of ASP.NET Core Security*. Retrieved from https://docs.microsoft.com/en-us/aspnet/core/security/?view=aspnetcore-2.1

Spasojevic, M. (2018, July). *ASP.NET Core Authentication with JWT and Angular – Part 1.* Retrieved from https://code-maze.com/authentication-aspnetcore-jwt-1/

Spasojevic, M. (2018, July). *ASP.NET Core Authentication with JWT and Angular – Part 2.* Retrieved from https://code-maze.com/authentication-aspnetcore-jwt-2/#