Part 1. Software Security

Martin Georgiev
1901560

1. Introduction

1.1 Background

ScottishGlen, a small organisation within the energy sector, requested a technical evaluation of its system and infrastructure. The testing was carried out due to fearmongering messages received by employees of the company, claiming that a hacktivist group would target them after a blog post by the CEO.

Knowing that the company uses ASP.NET (Microsoft, 2002) as their web frontend system used by HR to manage users and details, the analyst analysed various vulnerabilities which may be critical for the organisation's security. ASP.NET is a web application framework for dynamic web pages. It is open-source, allowing developers to freely use, change, and share it (in both modified and unmodified forms) based on their needs. While this may be good from a development standpoint, the free access to the source code of the framework allows adversaries to analyse it and identify various vulnerabilities that can be used to exploit potential targets.

Depending on how the framework was used to develop the frontend of the web application, attackers may exploit it to execute malicious code through XSS (Cross-Site Scripting), inject payloads to obtain sensitive data, or even send altered malicious links as phishing to steal user information (credentials, cookies, etc.).

1.2 Aims

The investigation aims to identify vulnerabilities that may threaten the organisation's network and business security. This will be achieved by reviewing vulnerabilities in the software and providing appropriate secure development practices to ensure that the issue will be mitigated before a potential attack. The report consists of the following sections:

- CVE Identification – Identifying how the software works and CVEs from a specific class of security faults. The section will cover the vulnerability in detail, including how it can occur, what issues are causing it, and how it can impact the organisation.
- Mitigation – Analysis and recommendations for development practices that will secure the vulnerable parts of the authentication system.
- Implementation – Justification of the recommended practices and how they would prevent the vulnerability.

2. CVE Identification

CVE (abbreviation for **Common Vulnerabilities and Exposures**) is a program run by MITRE. Its main goal is to "identify, define, and catalogue public cybersecurity vulnerabilities" (Mitre, 2022). The program allows security professionals to easily identify various vulnerabilities due to them being recorded with unique IDs. They can also be used to find specific security patches by various vendors that mitigate the weaknesses in the software.

As ASP.NET is used for developing web applications, it may be susceptible to attacks such as cross-site scripting, SQL injections, and DOS (Denial of Service). While all of them can lead to serious issues for an organisation, cross-site scripting can cause damage to both the company and their users. A carefully created and executed XSS can turn a benign and trusted website into a page with a malicious payload. It could be used to steal data and harm the victim and the reputation of the affected organisation. The effect may also vary depending on the type of XSS (reflected XSS by

altering the link or stored XSS by storing the malicious payload on the website (OWASP, 2022)). Reflected XSS is a more common attack, so the analyst chose to analyse **CVE-2005-0452**.

2.1 CVE-2005-0452

This CVE covers multiple cross-site scripting vulnerabilities in ASP.NET versions 1.0 and 1.1, as well as the .NET framework created by Mono Project (version 1.0.5). The vulnerability was first discovered in 2004 by a post-graduate student called Andrey Rusyaev (Rusyaev, 2004). Due to the vulnerability being so old, the National Vulnerability Database has only CVSS Version 2.0 severity score calculated for it (**4.3 Medium**). The score is set to Medium because of the medium access complexity and its impact on the system.

If specific conditions are met, attackers can carry out attacks on ASP.NET web applications. The circumstance requires improper filtering of HTML characters, indicating that the **security fault** is **improper URL character filtration**. The attack converts Unicode strings to National ASCII, bypassing regular filters and allowing an adversary to use special HTML characters ("<" and ">"). A way to execute the attack is with the fullwidth ASCII version of the shown characters – **%uff1c** (<) and **$uff1e** (<), and all characters between **%uff00** to **%uff60**. A successfully carried out attack can inject malicious HTML code or scripts (using **<script>**). Based on the payload, the attacker may cause various security failures such as data theft, user/website impersonation/defacement, carrying out actions that the user can perform, reading any data the user has access to, injecting Trojan functionalities, etc.

2.2 Identifying Security Faults

To efficiently identify security faults in the web application and the code during the development phase, developers can use the **OWASP Top 10** framework (OWASP, 2021). It is a standard specifically created for the developer and web application security and represents a consensus regarding websites' most common and serious security vulnerabilities. In terms of **CVE-2005-0452**, the vulnerability can fall into three different categories of security faults:

- **A03:2021-Injection** – any security failure resulting from injections such as SQLi and XSS.
- **A04:2021-Insecure Design** – any security failures resulting from design flaws such as improper validation/filtration, etc.
- **A06:2021-Vulnerable and Outdated Components** – any security failure resulting from the use of old and vulnerable code, libraries and practices.

While the source code of the ASP.NET framework is vulnerable, the security faults can be mitigated by the developers by following different secure software engineering practices.

3. Recommendation

The developers should consider the following mitigation techniques to protect the organisation's network integrity and prevent the discussed attack in the previous section. As the vulnerability poses a risk to the company and its users, it is recommended that secure software development practices should be immediately and continually implemented.

In terms of **CVE-2005-0452**, the most efficient development practice (both on a time and price scale) would be **input sanitisation** (Mauri, 2023). Any data from untrusted sources should be filtered (including full-width ASCII characters, as they are used in the exploitation of the vulnerability). Such sources may be user input fields, HTTP headers and web page URLs. The filtration must be done

manually and without the use of validation mechanisms embedded within ASP.NET versions 1.0 and 1.1. An example of such a mechanism is the **validateRequest="true"** – it is designed to protect against SQLi and XSS but allows full-width ASCII characters. (**Figure 1**)

```
http://server.com/attack2.aspx?test=%uff1cscript%uff1ealert('vulnerability')%uff1c/script%uff1e


Web page 'attack2.aspx' prints HTTP request parameter 'test'.
Web page like following:

<!-- Web page attack2.aspx -->
<% @Page Language="cs" validateRequest="true" %>
<%
   Response.Write(Request.QueryString["test"]); // Attack through URL parameter
%>

Web.config for server.com like following:

<configuration>
  <system.web>
    <globalization responseEncoding="windows-1251" />
  </system.web>
</configuration>
```

***Figure 1** – Attack bypassing **validateRequest="true"** (Rusyaev, 2004).*

Another example is the **Server.HTMLEncode** method. This function applies HTML encoding to a string specified by the developer. It is used to convert possibly malicious characters to their HTML-encoded equivalent to mitigate attacks. The method, however, does not have any filtration mechanisms for full-width ASCII characters. It allows attackers to freely use such characters and execute XSS attacks on the web application (**Figure 2**)

```
http://server.com/attack3.aspx?test=%uff1cscript%uff1ealert('vulnerability')%uff1c/script%uff1e


     Web page 'attack3.aspx' prints:
        1. HTTP request parameter 'test',
        2. Some string with injected Unicode characters.

     Web page like following:

    <!-- Web page attack3.aspx -->
    <% @Page Language="cs" %>
    <%
       Response.Write(Server.HtmlEncode(Request.QueryString["test"])); // 1) Attack \
               through URL parameter
       string code = \
Server.HtmlEncode("\xff1cscript\xff1ealert('vulnerability')\xff1c/script\xff1e"); // \
2) Attack through injected Unicode characters   Response.Write(code);
    %>

    Web.config for server.com like following:

    <configuration>
      <system.web>
        <globalization responseEncoding="windows-1251" />
      </system.web>
    </configuration>
```

***Figure 2 –** Attack bypassing the **HTMLEncoding** method (Rusyaev, 2004).*

Compared to other secure development practices, this would be the most efficient one for multiple reasons. In terms of **code reviews** (Collier, 2016), developers would not be able to identify the issue as everything would appear normal and the code will execute. The issue is in ASP.NET's filtering functions and not directly because of code faults. This would make code reviews an expensive yet inefficient and unsuccessful method for identifying this issue.

Another method for secure software development is **frequent software updates** (Symanovich, 2021). As ASP.NET versions 1.0 and 1.1 are outdated and have not been in use since 2005, they will have multiple publicly announced vulnerabilities. Attackers can use them to exploit the system, or

they may even find new vulnerabilities. The vendor will also not provide any security patches due to the framework being out of its service period. Updating to a newer version of the framework (such as **ASP.NET Core v6.0.0**) will ensure that the system will be secure and will not use any deprecated/vulnerable functions. As this will be a complicated, time-consuming, and expensive task (the entire application will need to be re-developed), input sanitisation will be a more efficient option. Updating the framework is still highly recommended and should be considered by the executives and implemented as soon as the organisation has the required resources.

Encryption mechanisms and **encrypted traffic (HTTPS**) (Jones, 2022) are also important secure development practices. They will prevent an attacker from obtaining human-readable data from the servers or intercepting traffic. This, however, would prevent attackers from attacking the server/sniff the traffic. Encryption would not prevent them from altering the web application's URL with a malicious payload and sending it to a user. Depending on the payload, adversaries can then capture user data such as cookies and plain text credentials or even inject Trojan functionality into the website. For this reason, this method will not be successful in preventing the exploitation of **CVE-2005-0452**.

The last secure software engineering practice that will be compared to the recommendation is "**Principle of Least Privilege**" (Palo Alto, 2022). As an attacker can execute any actions and read any data accessible to the victim, all accounts should be given the least privileges they require for their normal functionality. High privileges should not be provided to users who are not expected to access sensitive data or specific functions within the application. While this may mitigate other attacks, appropriate user privileges will not prevent adversaries from executing XSS payloads on the web application. The practice can only limit the payload's capabilities based on what victim was targeted and exploited.

4. Implementation

As previously mentioned, the development process of the application will require changes to properly implement the secure software engineering practice. Further information about the CVE itself can be found in MITRE's CVE database (Mitre, 2005) and in the National Vulnerability Database (NIST, 2005). The databases also contain multiple links regarding vendor advisory and a complete description provided by the analyst who discovered the vulnerability. It is possible to prevent **CVE-2005-0452** from being exploited with two different protection methods.

The first method is more straightforward – filtering and allowing only a Unicode for output on ASP.NET pages. It can be achieved with a simple web.config function. The function will force the web application to output only Unicode characters, which may be problematic based on the operating system of the device and whether it supports the format. This may still be an issue depending on how different software (such as databases) are setup. However, if the coding framework is outdated, using different encoding standards may cause unexpected bugs. Developers will be required to add the following configuration structure to the web application to filter the entire output data (Rusyaev, 2004):

```
<configuration>
    <system.web>
            <globalization responseEncoding="utf-8" />
    </system.web>
</configuration>
```

The second method, recommended by the researcher, is input sanitisation. It is also an OWASP-recommended strategy for preventing XSS vulnerabilities in websites (Suranga, 2022). It is a mechanism for removing unsafe data from unknown HTML strings before they are presented to the user. There are two types of sanitisations – client-side (remove dangerous input from the website's DOM) and server-side (prevention of stored XSS attacks). Various libraries can provide developers with client-side sanitisation – i.e., **DOMPurity**, **js-xss**, **sanitize-html**, etc. Another way to sanitise input/output is with the **HTML Sanitizer API**. It is an API with two developer functionalities – a class and a method (**Element.setHTML**).

The class allows developers to create an object with either default or customised configuration. The default configuration object contains various techniques to mitigate known XSS vulnerabilities. It can then be customised with various functions – allow/block/drop elements, allow/drop attributes, allow custom elements and allow comments (W3C, 2022). An example of a custom object can look as follows:

```
{
        'blockElements':[
                '%uff1c',
                '%uff1e'
        ],
        'allowElements': [
                'p'
        ],
        'allowAttributes': {
                'style': ['*']
        }
}
```

Using such an API will allow developers to block only elements used in cross-site scripting attacks. When such elements are detected, they will be removed from the input while their children will be retained. An example can be given with ~~%uff1c~~script~~%uff1e~~alert(1)~~%uff1c~~/script~~$uff1e~~. The input will then be changed to "scriptalert(1)/script", preventing it from executing the malicious payload.

Developers can experiment with this API in the playground created by the vendor (West, 2021). This will allow them to efficiently test various configurations before they deploy them on the web frontend. Using the recommended secure software engineering practice will protect the system from cross-site scripting attacks and the organisation's users.

References

MITRE (2005). *CVE-2005-0452*. [online] Available at: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-0452 [Accessed 8 Mar. 2023].

NIST (2005). *CVE-2005-0452 Detail*. [online] Available at: https://nvd.nist.gov/vuln/detail/CVE-2005-0452 [Accessed 8 Mar. 2023].

Microsoft (2002). *ASP.NET*. [online] Available at: https://dotnet.microsoft.com/en-us/apps/aspnet [Accessed 8 Mar. 2023].

OWASP (2022). *Cross-Site Scripting (XSS)*. [online] Available at: https://owasp.org/www-community/attacks/xss/ [Accessed 9 Mar. 2023].

Rusyaev, A. (2004). *XSS vulnerability in ASP.net*. [online] Available at: https://marc.info/?l=bugtraq&m=110867912714913&w=2#5 [Accessed 9 Mar. 2023].

OWASP (2021). *OWASP Top 10*. [online] Available at: https://owasp.org/www-project-top-ten/ [Accessed 9 Mar. 2023].

Mauri, J. (2023). *How to Use Input Sanitization to Prevent Web Attacks*. [online] Available at: https://www.esecurityplanet.com/endpoint/prevent-web-attacks-using-input-sanitization/ [Accessed 9 Mar. 2023].

Suranga, S. (2022). *What you need to know about inbuilt browser HTML sanitization*. [online] LogRocket. Available at: https://blog.logrocket.com/what-you-need-know-inbuilt-browser-html-sanitization/ [Accessed 10 Mar. 2023].

Collier, J. (2016). *The Evolution of Code Review: Pros and Cons of Code Review Methods [Infographic]*. [online] Available at: https://smartbear.com/blog/pros-and-cons-of-code-review-methods-infographic/ [Accessed 12 Mar. 2023].

Symanovich, S. (2021). *5 reasons why general software updates and patches are important*. [online] Available at: https://us.norton.com/blog/how-to/the-importance-of-general-software-updates-and-patches [Accessed 12 Mar. 2023].

Jones, J. (2022). *11 Best Practices for Developing Secure Web Applications*. [online] Available at: https://www.lrswebsolutions.com/Blog/Posts/32/Website-Security/11-Best-Practices-for-Developing-Secure-Web-Applications/blog-post/ [Accessed 12 Mar. 2023].

Palo Alto (2022). *What Is the Principle of Least Privilege?* [online] Available at: https://www.paloaltonetworks.com/cyberpedia/what-is-the-principle-of-least-privilege [Accessed 12 Mar. 2023].

W3C (2022). *Sanitizer API Documentation*. [online] Available at: https://wicg.github.io/sanitizer-api/ [Accessed 12 Mar. 2023].

West, M. (2021). *Sanitizer API Playground*. [online] Available at: https://sanitizer-api.dev/ [Accessed 12 Mar. 2023].