Attack Lab 1: Input Injection Attacks

Date Due: Wednesday, September 20, 2023, by 11:59 PM.

Introduction

This lab will demonstrate some of the common input related weaknesses in web applications and some of the attacks that make those weaknesses dangerous. This assumes you have completed Lab 0 which involves performing the Lab environment setup. For this lab, you will need to run the WebGoatCore application which was performed as part of Lab 0: Phase 2.

WARNING: Do not attempt to use these techniques elsewhere. Only run them on the WebGoatCore application either inside the virtual machine or the application running on your own machine. There are legal consequences to you if you use these elsewhere!

Submission Instructions

For each of the Phases provided in this lab, you will be exercising attacks on the WebGoatCore application. Please submit a single writeup containing all the Phases and the answers to each of the questions provided in each phase. For each question, the answers should be provided in the following format in a Word or PDF document:

Question Number.

a) Describe in words and provide the input given to the application. Provide the input as is with no decorations. If the input is a URL, provide the URL encoded string. If the input is provided as text to multiple fields, list the fields and the input provided for each. If the input is non-printable characters, please provide the bytes provided to the input in Hexadecimal format. For example: 9ABCD01234. If BurpSuite was used, **copy the cURL command from BurpSuite** and provide it as part of this answer.
b) Describe the result (the output as well as any effects that were not seen).
c) Provide screenshot(s) of any output or effects observed.
d) Provide any applicable CWE-ID, Filename, Line Number of all the weaknesses that relate to the vulnerability. Please refer to the **ENPM809W Project handout Phase 3: Code Review Findings** section for a template of how to report this. You may have multiple findings related to the vulnerability you attacked.
e) Describe the vulnerability and what role the weakness plays in allowing the input (attack vector) to compromise the application.

Phase 1: SQL Injection (8 points)

Structured Query Language or SQL is a domain-specific language used for programming and managing data stored in database systems (typically relational database management systems). It acts as both a command-and-control channel for manipulating databases. In this part of the lab, we look at why they should be analyzed for security weaknesses [1]. The WebGoatCore application uses Entity Framework (see tutorial at https://docs.microsoft.com/en-us/aspnet/core/data/ef-mvc/intro?view=aspnetcore-3.1).

CWE-89: Improper Neutralization of Special Elements used in SQL Command ('SQL Injection'), is typically used to describe a weakness where a software system or component constructs all or part of a SQL command or query using externally influenced input but does not neutralize or incorrectly neutralizes special elements that could modify the intended SQL command.

For example:

```
string sql = "select name, email from Login where name like '" + name + "%'";
```

Here the last name field can be supplied as "' OR 1=1 OR '" which causes the original query condition to be ignored and the provided condition in the input field for 'name' to be used instead of returning all the records (since 1=1 is always TRUE).

Now it is your turn to attempt the same kinds of attacks. You can use the DBeaver application to explore the NORTHWIND.sqlite database file found in the WebGoatCore directory. Please note that when you Build the solution using Visual Studio or VS Code, it copies this file into the bin directory under WebGoatCore for use in the running application.

Navigate to the 'Blog' section of the WebGoatCore application and answer the following questions.

1. Now attempt to provide a reply to any message performing an SQL injection to give the 'MyUser' user an Admin role. Password for MyUser is "MyPassword" without the quotes. Please answer the questions a-e given above that are relevant to this attack. (8 points)

Phase 2: Path Traversal and Upload of Dangerous Files (20 points)

A lot of web-based applications or components use Uniform Resource Locators (URL) to identify the resource a client application would like access to. They additionally map those resources to locations within the server's file system. The system sometimes also allows for file uploads to facilitate the exchange of information or as part of its service or mission. Additionally, the system may read the content of the files and use it as part of its functionality or for validation. A lot of security related weaknesses can be introduced when implementing such features.

CWE-73: External Control of Filename or Path is typically used to describe a weakness in a software system or component that allows the path or filename to be controlled external to the software system such as through user input, environment of the system or other means not part of the software system. This weakness describes the condition where the software system allows external control of the path, for example, via HTTP parameters etc.

CWE-22: Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal'), is typically used to describe a weakness in a software system or component which allows external input to be used in the

construction of a pathname that is intended to identify file or directory that is located underneath a restricted parent directory. This is more specific than CWE-73 in that the user input is used to craft the path name along with the software system's restriction of the path to a specific directory, however the weakness allows for escaping that restriction.

CWE-434: Unrestricted Upload of a File with Dangerous Type, is typically used to describe a weakness in a software system or component where it allows an attacker to upload or transfer files of dangerous types that can be automatically processed within the system's environment.

Now it is your turn to attempt attacks on these types of weaknesses. Attempt the following and for each question, answer using the format given at the top of this Lab handout:

2. The WebGoatCore store stores secret credit card information in a file. Attempt to access the stored credit card information by manipulating only the URL in the browser. Please answer the questions a-e given above that are relevant to this attack. (4 points)
3. Using BurpSuite to capture the request, try uploading a file in the About page which causes manipulation of "**appsettings.json**". Please answer the questions a-e given above that are relevant to this attack. (6 points)
4. Now try uploading a file in the About page with an extension that will get executed when accessed. Please answer the questions a-e given above that are relevant to this attack. (6 points)
5. Readline DoS: Now attempt the file upload on the home page and cause the page to be unresponsive or the IIS server to crash. Try using https://www.fec.gov/files/bulk-downloads/2018/indiv18.zip. Please answer the questions a-e given above that are relevant to this attack. (4 points)

Phase 3: Regular Expression DoS (4 points)

There are many approaches to input validation including metacharacter recognition, filtering, allow list approaches and content validation. Another useful tool is the use of regular expression languages to filter, search or match the required input in a software system or component. However, there exist security weaknesses that may exist depending on how the regular expressions are implemented.

CWE-1333: Inefficient Regular Expression Complexity, is typically used to describe a weakness in a software system or component that uses a regular expression with an inefficient, possibly worst-case computational complexity that consumes excessive CPU cycles.

Now it is your turn to attack such a weakness.

6. Now attempt to produce a DoS on the product search page by crafting an input that leads to a Regex DoS and answer the questions a-e above in the format given at the top of this Lab handout. (4 points)

Phase 4: Cross-Site Scripting (XSS) (10 points)

Web applications use scripting languages like ECMAScript/Javascript/TypeScript etc. to facilitate user interface changes and make dynamic web applications. Today those same languages now run both client side and server side and come with the risk of security weaknesses being introduced during implementation.

Cross-Site Scripting has been in the Top 25 Most Dangerous Software Weaknesses list for a while now and still made it in second in the 2022 list [3].

CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-Site Scripting') describes a weakness where the software does not neutralize or incorrectly neutralizes user-controlled input before it is place in output that is then used as part of a web page that is then served to other users.

This one is specific to web applications, however the scripting language used is agnostic to the weakness. JavaScript is one of the common web based languages used to exercise attacks on such weaknesses since it allows for some dangerous functions like eval (), alert () or confirm () to run with the privilege of the web application user.

There are several types of XSS attacks. Two of them include Stored XSS and Reflected XSS. In normal XSS attacks, the malicious script is typically injected directly into the output of another web page running a web application. Today's browsers typically prevent these kinds of attacks and implement the Same Origin Policy that prevents scripts from other sites (read domains) from running in the context of the current site loaded in the same browser. If you are browsing abc.com and securedbank.com at the same time, this prevents scripts sourced from abc.com running in the context of securedbank.com.

However, a Stored XSS attack occurs when the application accepts a script in its input field which is then stored (either in a database or file etc.) and then displayed/run as part of the application's output. However, the script may run under any user context since it is part of the application's output for all users. This can be used, for example, to steal session IDs, user's cookie information/data etc.

A reflected XSS attack is a further refinement on the XSS attack concept, where the script is not directly stored, rather the attacker provides a malicious script via an input (like query parameters or cookie values that are not sanitized) and that script is immediately displayed/run in the resulting output or response. This type of attack is used in clickjacking attacks for example, where a victim clicks a specially crafted link by an attacker which then immediately executes the script based on the URL and fools the victim into seeing something that is not real.

Now it is your turn to attack such weaknesses.

7. The 'Blog' section has a 'respond' feature where any user can go and comment on blog posts. Craft a blog response which causes your full name to be displayed in an alert box when a user visits the Blog. Answer the questions a-e above in the format given at the top of this lab handout. (5 points)
8. Now navigate to one of the product detail pages and observe the URL. Now craft a Blog response containing a script that opens the product details page of any product in the Store which then causes your full name to be displayed in an alert box.. Answer the questions a-e above in the format given at the top of this lab handout. (5 points)

Phase 5: SharpFuzz fuzzing (8 points)

9. Now we will try to fuzz and find weaknesses in a JSON parser library using SharpFuzz fuzzing tool. The Jil library for .NET is a popular and fast JSON serialization and deserialization library for objects. The SharpFuzz project (https://github.com/Metalnem/sharpfuzz) by Nemanja Mijailovic that uses a .NET interface to interact with AFL (American Fuzzy Lop) to fuzz inputs. AFL cannot

be installed on Windows without using WSL 2 (Windows Subsystem for Linux version 2). You will need to install and setup AFL on your ***local*** machine following the instructions:

i. This assignment needs to be performed on your host system, ***not on VM.***

ii. For Windows host **only**: Please install **WSL 2** (https://docs.microsoft.com/en-us/windows/wsl/install) with **Ubuntu 20.04** (https://ubuntu.com/tutorials/install-ubuntu-on-wsl2-on-windows-10#1-overview) on it. Make sure that the WSL version for Ubuntu is set to **2** or you may face issues with next steps.
For Mac and Linux Host systems, you don't have to install any additional features at this point.

iii. Install **DotNet SDK 3.1** on your system.
For Linux (and WSL 2 Linux) please refer to this article https://docs.microsoft.com/en-us/troubleshoot/developer/webapps/aspnetcore/practice-troubleshoot-linux/1-3-install-dotnet-core-linux
For Mac, please refer to this article https://docs.microsoft.com/en-us/dotnet/core/install/macos

iv. Once you have setup dotnet sdk 3.1 on your host/wsl 2, copy this shell script (https://github.com/Metalnem/sharpfuzz#installation) on your system or use curl or wget to fetch it from https://raw.githubusercontent.com/Metalnem/sharpfuzz/master/build/Install.sh.

v. Follow the implementation steps as mentioned here: https://github.com/Metalnem/sharpfuzz#usage. For creating a console application mentioned in the Step 4 of this GitHub repository, you can use your VM's Visual Studio 2019 (or use your host machine's VS, if available) to create a new console application. Make sure that while creating a new console application, you select .NET Core 3.1 Long Term Support option to work on. This is mandatory. If you plan to use the VM, after creating the application, please copy the whole console project to your WSL 2 instance or in case of mac - on your host system.

vi. The Fuzzer (afl-fuzz) should run a minimum of 5 minutes

vii. As you can note, we are catching DeserializationException since we are not interested in known exceptions already caught by the Jil library, rather we are interested in any other crashes that remain uncaught.

viii. Please ONLY answer items b-d for this question and put N/A for items a, e-f. Here the weakness will be with respect to the Jil library with no Filename, or Line numbers and the output will be based on how it crashes the library. Please provide at least one crashing input generated by the fuzzer in the **crashes** directory.

Submission Criteria

Please submit a writeup with:

1. Your name, UMD email ID and Lab Number.
2. The answers provided in the format given at the top of this lab handout as specified in each of the questions in all the phases.

Please only submit a **Word document** or a **PDF**. There is no code submission for this lab.

References

1. Wikimedia Foundation. (2021, August 2). SQL. Wikipedia. https://en.wikipedia.org/wiki/SQL. Retrieved August 8, 2021.
2. CWE. (2021, July 20). CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection'). https://cwe.mitre.org/data/definitions/89.html. Retrieved August 8, 2021.
3. CWE. (2022, June 28). 2022 CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html. Retrieved September 5, 2022.