ENPM809W – Introduction to Secure Software Engineering

Attack Lab 1: Input Injection Attacks

Date Due: Tuesday, September 21, 2021, 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 WebGoat.NET 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 WebGoat.NET 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!

For each of the following on the WebGoat.NET 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, please look at the Lesson Instructions button on the given page unless specified otherwise. The answers should be provided in the following comma separated format:

Question Number.

a) Provide the URL of the WebGoat.NET application page where you are exercising the question. Do not include any query parameters or any other special characters in the answer.
b) Describe 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.
c) Describe the output result.
d) Provide a screenshot of the output.
e) Provide the CWE-ID, Filename, Line Number of the weakness that is at the heart of the vulnerability.
f) Describe the vulnerability and what role the weakness plays in allowing the input (attack vector) to compromise the application.

Phase 1: SQL Injection (10 points)

Structured Query Language or SQL is a domain-specific language used for programming and managing data held 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].

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 returning all the records (since 1=1 is always TRUE).

Now it is your turn to attempt the same kinds of attacks. Navigate to the Input Attacks section of the WebGoat.NET application and answer the following questions.

1. Navigate to 'Input Attacks' on the left-side navigation bar. Now attempt the 'SQL Error Messages' exercise and answer using the format given at the top of this lab handout.
2. Now attempt the 'SQL Injection' exercise and answer using the format given at the top of this lab handout.

Phase 2: Path Traversal and Upload Dangerous Files (10 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-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.

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.

3. Now attempt the 'File Download Path Manipulation' exercise and answer using the format given at the top of this lab handout.
4. Now attempt the 'File Upload Path Manipulation' exercise and answer using the format given at the top of this lab handout.
5. Now attempt the 'Readline DoS' exercise and answer using the format given at the top of this lab handout. Try using https://www.fec.gov/files/bulk-downloads/2018/indiv18.zip.

Phase 3: Regular Expression DoS (10 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 the 'Regex DoS' exercise and answer using the format given at the top of this lab handout.

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 2021 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) to run 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 a weakness.

7. Now attempt the ‘Stored XSS’ exercise and answer using the format given at the top of this lab handout.
8. Now attempt the ‘Reflected XSS’ exercise and answer using the format given at the top of this lab handout.

Phase 5: WinSharpFuzz fuzzing (10 points)

9. Now we will try to fuzz and find weaknesses in a JSON parser library using WinSharpFuzz fuzzing tool. The Jil library for .NET is a popular and fast JSON serialization and deserialization library for objects. The WinSharpFuzz project (https://github.com/nathaniel-bennett/winsharpfuzz) by Nathaniel Bennett uses a .NET interface to interact with libfuzzer (https://www.llvm.org/docs/LibFuzzer.html) in order to mutate inputs and try to crash the application. WinSharpFuzz is based on SharpFuzz by Nemanja Mijailovic, which uses AFL (American Fuzzy Lop) to fuzz inputs. AFL cannot be installed on Windows without using WSL 2 (Windows Subsystem for Linux version 2) and hence we use libFuzzer.
    a. The WinSharpFuzz project has already been cloned and built on the VM provided for the class under C:\Users\student\Workspace\winsharpfuzz.
    b. The WinSharpFuzz.CommandLine.exe command and WinSharpFuzz.Instrument.exe binaries have also been added to the PATH variable for the System.
    c. Download the Jil NUGET package from https://www.nuget.org/api/v2/package/Jil/2.16.0 and rename the extension from ‘.nuget’ to ‘.zip’ and extract to jil.2.16.0 folder.
    d. We will first instrument the Jil.dll library using WinSharpFuzz. Open a Developer Command Prompt for VS 2019 and go to the extracted folder for jil.2.16.0. Now run the following command to instrument the library:

```
C:\...\jil.2.16.0> cd lib\netstandard2.0
C:\...\jil.2.16.0\lib\netstandard2.0> WinSharpFuzz.Instrument.exe
Jil.dll
```

    e. Create a new .NET Console application called JilWinSharpFuzzer and follow directions on the WinSharpFuzz GitHub page to create a Testing Harness. The code in the test harness will be:

```
using Jil;
using System.IO;
using WinSharpFuzz;

namespace JilWinSharpFuzz
{
    public class Program
    {
        public static void Main(string[] args)
        {
            Fuzzer.LibFuzzer.Run(bytes =>
            {
                // pass input bytes to library functions here
                try
                {
                    JSON.DeserializeDynamic(bytes.ToString());
                }
                catch (DeserializationException) { }
            });
        }
    }
}
```

f. Note above that we catch 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.

g. Now copy the instrumented Jil.dll file to the root of your new .NET Console Application. You will need to modify the CSharp Project file and add the following to add a reference to the instrumented Jil library:

```
<ItemGroup>
        <Reference Include="Jil">
                <HintPath>Jil.dll</HintPath>
        </Reference>
</ItemGroup>
```

h. Fix any other dependency issues such as adding the WinSharpFuzz and Sigil (a dependency for Jil) packages to the project.

i. Finally to start the fuzzer using the test harness, go to the directory where the test harness executable JilWinSharpFuzzer.exe is generated (typically in <Project directory>\bin\Debug\netcoreapp3.1), create a "Tests" directory and run:

```
..\bin\Debug\necoreapp3.1> WinSharpFuzz.CommandLine.exe --
target_path="C:\Users\student\Workspace\JilWinSharpFuzz\bin\Debug\netcor
eapp3.1\JilWinSharpFuzz.exe" "Tests" -max_len=285 -jobs=4
```

j. This will attempt to run the fuzzing across 4 threads using inputs with a maximum length of 285 bytes and will put output into a log file with the name fuzz-#.log. You can open the file to check its progress. WARNING: This will take a very very long time.

k. To speed it up a little, we can provide an example json input file in the "Tests" directory. Create a test.json with the following contents and then try running the above command:

```
{"menu":{"id":1,"val":"X","pop":{"a":[{"click":"Open()"},{"click":"Close
()"}]}}}
```

l. Please ONLY answer items b-f for this question and put N/A for item a. 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.

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 phases.

Please only submit a **Word document** or a **PDF**. There is no code submission for this lab. Each of the phases is worth 10 points.

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. (2021, July 26). 2021 CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html. Retrieved August 8, 2021.