Injection Prevention Cheat Sheet

From OWASP

Contents

Introduction

This article is focused on providing clear, simple, actionable guidance for preventing the entire category of Injection flaws in your applications. Injection attacks, especially SQL Injection, are unfortunately very common.

Application accessibility is a very important factor in protection and prevention of injection flaws. Only the minority of all applications within a company/enterprise are developed in house, where as most applications are from external sources. Open source applications give at least the opportunity to fix problems, but closed source applications need a different approach to injection flaws.

Injection flaws occur when an application sends untrusted data to an interpreter. Injection flaws are very prevalent, particularly in legacy code, often

found in SQL queries, LDAP queries, XPath queries, OS commands, program arguments, etc. Injection flaws are easy to discover when examining code, but more difficult via testing. Scanners and fuzzers can help attackers find them.

Depending on the accessibility different actions must be taken in order to fix them. It is always the best way to fix the problem in source code itself, or even redesign some parts of the applications. But if the source code is not available or it is simply uneconomical to fix legacy software only virtual patching makes sense.

Application Types

Three classes of applications can usually be seen within a company. Those 3 types are needed to identify the actions which need to take place in order to prevent/fix injection flaws.

A1: New Application

A new web application in the design phase, or in early stage development.

A2: Productive Open Source Application

An already productive application, which can be easily adapted. A Model-View-Controller (MVC) type application is just one example of having a easily accessible application architecture.

A3: Productive Closed Source Application

A productive application which cannot or only with difficulty be modified.

Forms of Injection

There are several forms of injection targeting different technologies:

Query languages

The most famous form of injection is SQL Injection where an attacker can modify existing database queries. For more information see the SQL Injection Prevention Cheat Sheet.

But also LDAP, SOAP, XPath and REST based queries can be susceptible to

injection attacks allowing for data retrieval or control bypass.

Scripting languages

All scripting languages used in web applications have a form of an eval call which receives code at runtime and executes it. If code is crafted using unvalidated user input code injection can occur which allows an attacker to subvert application logic and eventually to gain local access.

Every time a scripting language is used, the actual implementation of the 'higher' scripting language is done using a 'lower' language like C. If the scripting language has a flaw in the data handling code 'Null Byte Injection' attack vectors can be deployed to gain access to other areas in memory, which results in a successful attack.

Operating System (OS) Commands

Application developers sometimes implement operating system interactions using calls to system utilities to create and remove directories for example. Here unvalidated input can lead to arbitrary OS commands being executed.

Network Protocols

Web applications often communicate with network daemons (like SMTP, IMAP, FTP) where user input becomes part of the communication stream. Here it is possible to inject command sequences to abuse an established session.

Injection Prevention Rules

Rule #1 (Perform proper input validation):

Perform proper input validation. Positive or “whitelist” input validation with appropriate canonicalization is also recommended, but is not a complete defense as many applications require special characters in their input.

Rule #2 (Use a safe API):

The preferred option is to use a safe API which avoids the use of the interpreter entirely or provides a parameterized interface. Be careful of APIs, such as stored procedures, that are parameterized, but can still introduce injection under the hood.

Rule #3 (Contextually escape user data):

If a parameterized API is not available, you should carefully escape special characters using the specific escape syntax for that interpreter.

Injection Prevention Matrix

The matrix below shows how much work is required depending on the proposed solution and the application type.

In order to fix or prevent the problem there are currently three ways of doing it:

- (Re-)Design:

With a clear application design decision the injection problem can be avoided. Using an OR-Mapper (e.g. Hibernate) or consistent parametrization of all input data (e.g. stored procedures or ideally: prepared statements). Other injection flaws (e.g. with XML) can only be avoided with dedicated output coding, where necessary.

- Internal Patching / Code Rewrite:

Rewrite parts of the application, to fix the problem or embed a library like the OWASP ESAPI to introduce black and white list based input validation

- External Patching / Web Application Firewall:

Deploy a web application firewall to introduce virtual patching capability (black, white and pro active security measures)

Solution	App Type A1	App Type A2	App Type A3
(Re)Design	easy	hard	n/a
Rewrite	medium	hard	n/a
WAF	medium	medium	medium

Depending on the application and the proposed solution a specific amount work is required.

- easy: little work required
- medium: moderate amount of work required
- hard: considerable amount of work required
- n/a: not possible

Retrieved from "http://www.owasp.org/index.php/Injection_Prevention_Cheat_Sheet"
Categories: OWASP Application Security Verification Standard Project | OWASP Enterprise Security API | How To | Cheatsheets

- Powered by MediaWiki