RogueWave SOFTWARE

Accelerating Great Code

DEFEND AGAINST INJECTION-BASED ATTACKS

Understanding and mitigating common security vulnerabilities

As we rely on software more and more each day to support our daily activities, it becomes imperative that we implement that software in the most secure manner possible.

Not only does software control the computers we use to do our work, it also controls our phones, cars, and even life-saving devices like insulin pumps and pacemakers. It's used to control machines that perform complex or dangerous tasks, as well as those that humans no longer manage, such as flight control systems in modern aircraft. It manages all the networks that provide us with electricity, water, natural gas, transportation, communications, and more. Increasingly, all these devices and systems are connected to each other behind the scenes, resulting in a growing network effect.

Couple this with end-users that expect devices to work properly and don't want to think about the technical ramifications of the software being used and we can realize how important it is to write code that is resistant to malicious attack.

We'll explore some of the most common security vulnerabilities currently plaguing the software development industry, and present different ways in which Static Code Analysis, or SCA, can detect them.

In this paper, we'll:

- Provide a detailed description of the weakness
- Show how it presents itself to the end user and the developer
- Explain mitigation strategies to help resolve each issue

SECURITY VULNERABILITIES

Security weaknesses today occur most often in software that is accessible from a user's desktop, tablet, or mobile device.

Web-based applications, network-enabled or controlled devices, and widely-used mobile software are the applications most targeted. This is followed by infrastructure applications such as operating systems, web servers, and browser-based software including plug-ins and extensions.

The cause of these weaknesses typically stems from the developer not anticipating how the software could be misused and made to perform actions it wasn't designed to do. The root problem is often a lack of secure input handling to block any application input or content that has not first been scanned for and had any harmful aspects filtered out.

This technique is often called input sanitization and it takes on different forms depending on the application and the tasks being implemented. For example, it can be designed to allow only alphanumeric characters in the range of 0 through 9 and A through Z.

However, this is a very simple example. Applications today need to accept input from a wide variety of sources, and that may make it harder to detect hostile intent. For example, the modern web application not only accepts input from the end user, it also reads data from a database, accepts uploaded files like pictures and documents, links to third-party web services, uses operating system APIs and frameworks to resolve computer and network names, integrates with LDAP services, and much more. This makes it increasingly difficult to simply trust all of the sources of input into an application.

Classes of vulnerabilities

There are different ways to classify software vulnerabilities, including using well known taxonomies such as OWASP Top Ten and Seven Pernicious Kingdoms. This white paper has taken those classes and grouped them according to the way their shared key features and criteria are addressed in automated analysis.

This paper focuses on the software vulnerability class described as injection vulnerabilities. This refers to the ability of an attacker to insert specific commands into the application or code that will execute undesired behavior on their behalf.

These attacks usually exploit an application at the point where it requests user input (data) for later processing. The most common types of injection vulnerabilities include SQL injection, command injection, XPath, and LDAP injection. HTML/script injection is also common, and is better known as Cross Site Scripting (XSS). All of these weaknesses are exploited and mitigated in the same way.

MITIGATING VULNERABILITIES

Finding and fixing security vulnerabilities is a process that can be built into your existing software development lifecycle. For every type of vulnerability, the process involves three steps:

1. Detecting the problem
2. Understanding the problem
3. Fixing the problem

Detecting the problem

The most difficult and time-consuming step in this process is the detection of vulnerabilities. Traditional detection methods often involved manual code inspections, creation of test suites that usually covered only a subset of scenarios or analysis of code after check-in. These approaches are inefficient from both a time and resource perspective and are prone to missing vulnerabilities. Modern approaches to detection are far more cost-effective and reliable.

To make vulnerability detection automatic and comprehensive, static code analysis (SCA) has become the standard. SCA is the automated analysis of computer software in lieu of program execution. It's accomplished through a variety of techniques that include native build comprehension, source code compilation, examination of Abstract Syntax Trees (AST), and model-based simulation of all possible control flow paths and data object lifecycles.

The current generation of SCA tools has matured greatly over the past decade and they now discover many more weaknesses — including the injection vulnerabilities discussed here — while consuming fewer computing resources in the process. This means that SCA tools can find real issues and not just typing errors. These issues can be found earlier in the software development lifecycle which reduces costs and the analysis can be integrated into existing development environments so as not to add additional steps or tools to the developers existing workflow.

Leading SCA engines use control and data flow analysis, in addition to model-based simulation of runtime expectation, to find code injection (buffer overflows), denial of services (memory leaks and integer taint), uninitialized data (data injection), string taint (XSS, SQL/command injection), and concurrency related issues. These engines follow the code on an inter-procedural basis to determine where data enters the system and where it finally ends up, such as on a file system or in a database. These sources and sinks are key pieces of the puzzle when it comes to analyzing software.

The latest generation of SCA tools not only uncovers errors in existing code, but identifies security weaknesses as developers type. Analyzing each line of code as it is being written — like a spell checker for source code — helps ensure secure code is developed prior to check-in.

Static code analysis tools are an important part of a secure software development lifecycle. In fact, Microsoft recommends that developers perform static analysis as part of the "implementation phase." Specifically, SDL #10 states: "Analyzing the source code prior to compile provides a scalable method of security code review and helps ensure that secure coding policies are being followed."

Each of the vulnerabilities described in this paper can be detected with source code analysis.

UNDERSTANDING AND FIXING SECURITY ISSUES

In order to address detected issues, a developer must have a complete understanding of that issue, how it works, what it looks like, and what options are available to fix it. This section provides this type of overview for the various types of injection flaws that one can encounter.

SQL injection vulnerabilities

The most common code exploit is SQL injection, or the process of inserting into an application a string of SQL that will be executed by the database engine and/or retrieve data by executing commands (Insert, Update, and Delete) on behalf of the application.

These attacks are particularly common because there are more web-based applications out there than any other type. Also, developer experience and maturity is sometimes lower for this class of software, causing more vulnerabilities to emerge.

SQL injection vulnerabilities usually manifest themselves in source code as a concatenation of unvalidated user input converted to a string that is passed directly to the database engine's execution API. As a result, there are two parts of the defect that can be addressed separately: the user input path and the SQL execution processing, and SCA can help identify both parts.

```
public ResultSet getUserData(ServletRequest req, Connection conn)
    throws SQLException
{
    // Source of data comes via HTTP parameter
    String account = req.getParameter("account");

    // Use that string in SQL statement
    String query = "SELECT * FROM user_data WHERE userid='" + account + "'";

    // Execute it
    Statement stmt = conn.createStatement(...);
    ResultSet rows = stmt.executeQuery();

    // All good
    return rows;
}
```

Figure 1 | Example of an SQL Injection flaw

For example, in Figure 1, you can see that the "account" variable has been assigned directly from the request parameter, "req." It isn't checked or modified from its native state. This is then passed to the "query" variable along with the bulk of the SQL select command to create a new SQL command. This is then passed directly to the "executeQuery" API that runs the database engine.

Finally, the result is passed back to the call regardless of the result. At no point are checks made as to the reasonableness of the data.

Use a parameterized query

To mitigate this vulnerability, first and foremost you need to use a parameterized query to create the SQL that eventually gets passed to the database engine API.

All of the major languages provide some kind of support for this, especially the most popular ones. The Query Parameterization Cheat Sheet identifies which libraries you need.

In the example above, the Java "prepareStatement()" method would handle any potential abuse of the SQL call, because the query is precompiled by the database engine with placeholder values, separating the SQL code from the data. These placeholder values are then updated as the application is running and data is passed into the placeholder positions. These are processed as values to the preexisting compiled query and not as a modified version of the query, which is the key to how it prevents SQL injection.

Provide additional validation of parameters

Additional validation of the parameters passed into this function would also help prevent SQL injection attacks.

In the previous example, it might be prudent to check the type and length of the content being passed into the function. Limiting the character range to only numbers would also limit a possible attack.

Checking to determine if the result set makes sense would also limit an attacker's ability to exploit this weakness. For example, if the query returns hundreds of rows, this may indicate that a malicious or erroneous query was made and that an error condition should be returned.

Command injection vulnerabilities

The second example, shown in Figure 2, reveals a command injection flaw.

```
#include "stdafx.h"
#include "stdlib.h"

#define MAXPATH 255

int main(char* argc, char** argv)
{
    char cmd[MAXPATH] = "/usr/bin/cat ";
    strcat(cmd, argv[1]);
    system(cmd);

    return 0;
}
```

Figure 2 | Example of Command Injection flaw

In this simple example from the OWASP page, you can see that the user supplied input is passed to the "system()" command after it has been concatenated to the "cat" command.

By supplying the program with an additional command separated by a semicolon, an arbitrary application could be executed. For example, an input of "; rm –rf /" would delete the entire root directory. If this application was run with elevated privileges, the results could be catastrophic to the system.

Avoid calling the system API

The best way to mitigate this weakness is to never call the system API at all. From a programming perspective, there are less risky ways to interact with the system, although they can be more cumbersome to implement.

Ensure malicious input can't be executed

In cases where you absolutely must implement a call directly to the operating system to execute a command on your application's behalf, you need to ensure that malicious input can't be executed. You can accomplish this by validating the input using the most restrictive sanitizing method, preferably an exact match or white list approach. Note, a black list would not anticipate any new and novel techniques applied to work around the existing restrictions.

Cross site scripting vulnerabilities

Following SQL and command injection vulnerabilities, Cross Site Scripting (XSS) is one of the most common forms of attack on applications, especially for web-based applications.

The reason XSS is included in the injection grouping is because it relies on the same type of techniques to inject HTML and JavaScript into the web application through user-supplied input paths. The primary difference between XSS and SQL/command injection is that XSS is processed or rendered by a browser rather than being handled by separate resources such as a database engine or the operating system.

There are two main types of XSS attacks: stored (persistent) and reflected (non-persistent). The persistent type of XSS injection refers to an attack stored on the server for later dissemination without having to explicitly target particular users, which is also the primary limitation of the reflected type of attack. The reflected type of XSS injection is more common and refers to an attack initiated through data provided by a web client, such as in an HTTP query or through an HTML form submission.

One common type of reflected XSS attack is DOM-based. These attacks happen when the browser's DOM on the client-side is modified by the attacker's script, and this is the primary difference between DOM-based attacks and the other attacks which occur on the server-side.

```
<FORM Action="getmypassword.asp" Method="post">
       <INPUT Type="hidden" Name="email" Value="<% = Request.QueryString("email") %>">
       <INPUT Type="hidden" Name="id"    Value="<% = Request.QueryString("id") %>”>
       <INPUT Type="hidden" Name="h_name"Value="h_value">

       <b>Choose a Password:</b><br>
       <INPUT Type="Password" Name="thePass" Size="8" Maxlength="16"><br><br>

       <b>ReEnter Password:</b><br>
       <INPUT Type="Password" Name="thePass2" Size="8" Maxlength="16"><br><br>

       <INPUT TYPE="image" Src="images/btn_submit.gif" Width="100" Height="30" Border="0">
</FORM>
```

Figure 3 | Example of an XSS flaw

Sanitize and verify input, as well as encode output

Mitigating XSS vulnerabilities is more complicated, because there are many ways that unvalidated input can be inserted into a web page. Figure 3 demonstrates a vulnerable input form as detected by SCA, while Figure 4 demonstrates one way to mitigate the flaw.

```
<FORM Action="getmypassword.asp" Method="post">
       <INPUT Type="hidden" Name="email" Value="<% = Server.HTMLEncode(Validator(email, Request.QueryString("email"))) %>">
       <INPUT Type="hidden" Name="id"    Value="<% = Server.HTMLEncode(Validator(numeric, Request.QueryString("id"))) %>”>
       <INPUT Type="hidden" Name="h_name"Value="h_value">

       <b>Choose a Password:</b><br>
       <INPUT Type="Password" Name="thePass" Size="8" Maxlength="16"><br><br>

       <b>ReEnter Password:</b><br>
       <INPUT Type="Password" Name="thePass2" Size="8" Maxlength="16"><br><br>

       <INPUT TYPE="image" Src="images/btn_submit.gif" Width="100" Height="30" Border="0">
</FORM>
```

Figure 4 | Example of a fixed XSS flaw

In addition to sanitizing the input and verifying that it isn't inserted into an incorrect location in the web page, it's also a good idea to encode the output into a safe format. In fact, this approach is much easier, since the permutations of location to output type are significantly smaller.

All of these recommended techniques are described on The Query Parameterization Cheat Sheet. OWASP is the gold standard when it comes to "anti-XSS" education and has a wealth of information about how to properly remove these weaknesses from your applications.

XPath injection vulnerabilities

XPath and LDAP injection attacks are very similar in nature to the SQL and command injection attacks mentioned earlier. An XPath injection attack can occur when a web application uses XML data for storing important information. The attacker sends intentionally malformed input to the website in order to determine the XML data structure behind the page or site, and then crafts an XPath query that will return data not intended to be exposed to the attacker or any other audience.

In this way, the attacker can enumerate an entire XML file by logically returning arbitrary parent and child nodes and recording the data. And just like SQL injection attacks, XPath injection techniques convince the language to emit overly broad query results.

Sanitize data and use pre-compiled XPaths

To mitigate XPath injection vulnerabilities, you need to use input sanitation just as we explained for SQL injection. You also need to use precompiled XPaths, which are similar to prepared statements, in that they are compiled ahead of time and only accept parameters as input.

LDAP injection vulnerabilities

LDAP injection attacks apply only to applications that build LDAP commands from user-supplied input.

An application may want to integrate with an LDAP server in order to make logins compatible with an organization's corporate network. If the application's input is not properly sanitized, an attacker can fool the application into returning sensitive information from the server, including user credentials and organizational structure. Additionally, a vulnerable login implementation might allow anyone who can craft a proper LDAP command to login.

Use white listing

The best way to mitigate this weakness is to white list only the characters that are needed, which does include some non-alphanumeric characters, depending on whether the input is being used as part of the DN (distinguished name) or as part of a search query.

CONCLUSION

There are many types or classes of security vulnerabilities that exist in computer software today. With limited time and resources, it's difficult to understand, anticipate, or avoid all the different ways in which user input can be used against the intent of the application. Finding and removing these weaknesses as quickly as possible ensures that software running crucial functionality remains free from compromise.

Static code analysis tools can help you identify, understand, and remove these types of issues from an existing code base, as well as from new code, while you're writing it. In addition, having an SCA tool on your desktop helps you learn from existing mistakes, making you and your whole team more productive, and resulting in a more secure software solution.

For more information about injection vulnerabilities and SCA, check out the following resources:

- Take the free CWE-77 injection vulnerabilities online course to see more examples and types of injection vulnerabilities.
- Visit the Klocwork developer network to see a list of SCA checkers available and examples that show you the best way to remedy security vulnerabilities, even if you've never heard of them before.
- See how easily you can write secure code by signing up for a free trial of Klocwork.

Rogue Wave provides software development tools for mission-critical applications. Our trusted solutions address the growing complexity of building great software and accelerates the value gained from code across the enterprise. The Rogue Wave portfolio of complementary, cross-platform tools helps developers quickly build applications for strategic software initiatives. With Rogue Wave, customers improve software quality and ensure code integrity, while shortening development cycle times.