Command Injections

Guy-Vincent Jourdan
School of Inf. Tech. and Engineering
University of Ottawa
800 King Edward Avenue
Ottawa, Ontario, Canada, K1N 6N5
gvj@site.uottawa.ca

Command injections

In this section, we first define formally what a command injection vulnerability is, and we then provide several examples of different types of command injections chosen among the most common ones at the time of writing.

Formal definition of command injections

Command injections vulnerabilities are common and occur with different technologies, current and future. In order to grasp the essence of the problem, we propose here a definition of command injections which is technology independent.

Assume that we have a virtual machine M that accepts programs as input and "executes" these programs in some way. As it is customary with virtual machines, in order to be executed, the programs must be "valid", in that they have to be an element of the input language L_M recognized by M. The language L_M is usually specified by a grammar G_M. Thus, a valid program for a machine M is a program recognized by the grammar G_M.

The grammar G_M has two types of symbols: the *terminal* and the *non-terminal* symbols. Non-terminal symbols can be on both sides of the grammar's derivation rules, while terminal symbols can only appear on the right hand side of these rules. A word of L_M is made of terminal symbols and can be derived from the rules of G_M (see e.g. [9] for an overview of these classical concepts).

When looking at virtual machines grammars, we can identify two types of terminal symbols in G_M: the predefined constants of the language L_M and the variables. The predefined constants are the keywords of the language, the predefined symbols that are interpreted by M upon execution of a program, while the variables specify values, variable names etc.

Assume that an application makes use of such a machine M. That is, the application produces a program p recognized by G_M and sends p to M for execution. Assume moreover that p is at least partially produced at runtime, based in part on some user provided input i_p. We say that there is a *command injection vulnerability* (more precisely in that case, an L_M*-injection* vulnerability)

if for some inputs i_p, the application produces a p containing an element from i_p that is going to be recognized as a predefined constant of the language L_M by the grammar G_M.

Definition 1 (L_M-injection vulnerability) *An application has an L_M-injection vulnerability if*

1. *The application uses a virtual machine M.*
2. *It is possible for a user to provide a set of inputs i_p to the application that will cause the application to pass a program p to M.*
3. *There are in p some elements coming from i_p that will be parsed by the grammar G_M as a predefined constant of the language L_M.*

As Definition 1 specifies, a command injection vulnerability is a flaw that allows a user, potentially malicious, to modify the parsed input of a virtual machine in such a way that the modified portion of the input is going to be interpreted by the machine as a command. It is called a command injection because it reflects the ability of the user to inject a "command" that will be executed directly by the targeted virtual machine.

Our definition is in practice a bit too broad. Indeed, there are applications that intentionally let users "drive" some of the virtual machines used by the program, ideally in some limited and controlled way. For example, there are numerous Web-based applications that let by design users enter some HTML formatting instructions (such as bold, itemization etc.). Any such possibility would be flagged as command-injection flaw according to our definition. Clearly, if the application is meant to allow such input, then this is not an injection "flaw", although it can still be a security vulnerability[1]. Our definition can easily be modified to account for "application expected modifications", but we do not consider it in this paper for the sake of simplicity.

Most software applications make use of such virtual machines at runtime. In fact, a typical application will use several of these machines, used sometimes independently and sometimes in sequence. Common examples include SQL engines, XML parsers, HTML parsers, scripting engines etc. Whenever an command injection vulnerability against a machine M exists in an application, the attacker gets some level of control over the execution within M. In the following subsections, we review some of these injections attacks on practical examples.

SQL injections

Among command injections, SQL injections are perhaps the best known and the most studied (see e.g. [1, 2, 5, 8]). A SQL command injection vulnerability can exist whenever an application uses a SQL based database and constructs unfiltered (or improperly filtered) SQL commands based on user input. An attacker can then take this opportunity to inject its own SQL command, which will be passed down by the application to the SQL-database engine and executed.

The archetype example of a code which presents SQL-injections vulnerabilities is the following code snippet, where a SQL query is being build on-the-fly based on user provided user name and password:

[1]It should be noted that applications allowing directly this types of user-level manipulations can often be misused to allow unexpected command injections. See e.g. [3] for an example of possible consequences of allowing an "<a href=..." tag.

```
LoginQuery = ''SELECT * FROM UsersTable
    WHERE UserId=''' +
    request.getParameter(''userName'') +
    ''' AND Password = ''' +
    request.getParameter(''password'') +
    ''';'';
```

In this code, "LoginQuery" is built based directly on user-provided inputs (parameters "userName" and "password"). The intent is to query the database to see if the table "UsersTable" contains a record where the field "UserId" matches the user-provided parameter "userName" and the field "Password" matches the user-provided parameter "password".

Because the query is built directly by concatenation of predefined commands and user input, malicious users can actually modify the end query in various ways. For example, the password check can be bypassed by providing a password such as `' OR 1=1;` . If the user name provided is `administrator`, this will turn the query "LoginQuery" into

```
SELECT * FROM UsersTable WHERE
      UserId= 'administrator'
      AND Password = '' OR 1=1;
```

This command will always return the record with a field `UserId='administrator'`, regardless of the associated password value.

Another way to manipulate the code above is to insert `';` in the user name or password, followed by any SQL command. Again, this would lead to the execution of that user-provided command, which can be any operation (table manipulation, database update etc.) permitted by the access rights of the application on the database.

SQL-injections are clearly a particular case of our definition, where the virtual machine is the SQL-based database engine, and the virtual machine language is SQL. As we can see from the above examples, an attacker will successfully perform a SQL injection by providing input that will be interpreted by the database as actual SQL (e.g. the `OR` in the example above).

HTML-browsers injections (HTML, XSS...)

HTML-browsers can be the target of several distinct command injections attacks. These injections where at first not really considered harmful since they do not harm the system running the application in an obvious way; indeed, the application is not what is attacked, it is merely a vector used to send attacks down to the HTML-browsers of the users of the application.

We again consider a trivial example: assume that the application produces an HTML page containing user-provided comments to be displayed in other user's HTML browsers. These comments are typically stored in a database by another part of the application. Assume that the application has fetched an existing comment from the database, and stored the user name and comment into the `Username` and `Comment` variables. Assume that the HTML page showing the user-provided comments is built from the following server side ASP code:

```
<B><%UserName%></B> says <%Comment%>
```

If a malicious user has given a user name or a comment that include HTML tags, then those tags will be inserted into the resulting page as is and will be directly interpreted by the HTML browser of the users viewing the page. This is an example of an HTML-injection, and again clearly a particular case of our general definition: the virtual machine is the HTML-browser and the language is HTML. A malicious user successfully attacks such a system by injecting data that will be interpreted as HTML by the HTML grammar. This can for example be the insertion of a fake login form looking like the one of the application, which lies within a legitimate page of the application but sends the credential to some other location, controlled by the attacker.

In addition to HTML-injections, HTML-browsers typically have embedded scripting interpreters (such as JavaScript) that can be triggered from within an HTML page. Consider the same example above, but now with a malicious user entering a comment such as:

```
<script>alert(document.cookie)</script>
```

This would lead to an HTML-page being created by the application with the JavaScript instruction embedded in it, and thus again been interpreted by the HTML-browsers of subsequent users. This type of attack, known as "cross-site scripting", or "XSS" [7, 4, 6], can be very serious, since a successful malicious attacker can get access to the scripting engines embedded into the application's users' HTML-browser, and for example get a hold on their "cookies", which typically lead to session hijacking. In our model, the virtual machine is the script interpreter embedded into the HTML-browser, and a successful attack consists of injecting data that will be interpreted as script instructions (JavaScript in our example) by the interpreter. In the example, the injection did in fact trigger the invocation of the virtual machine, but of course similar example can be constructed where command injections happen within the scripting engine itself.

Shell command injections

Shell command injections vulnerabilities have been historically very important, although they seem to be in decline lately. This type of command injections occurs when the application invokes the operating system shell (C-shell, Bash etc. on Unix, command shell on Windows etc.) to initiate another program. For example, the application could be sending an email, and for that could be using directly the "mail" program under Unix; or, it could be attempting to print with the "lpr" command. If, as part of this program invocation, the application is using some user data without proper filtering, then again a malicious user can craft an input that will terminate the intended command and start another one of the user's choice.

This type of vulnerabilities used to be very common, presumably due to the Unix philosophy of "piping" applications together and of invoking programs from the command line. Of course, a successful shell command injection is potentially catastrophic, since the attacker gets an access to the operating system with the credential of the application.

Shell command injections are also a particular case of our command injection definition, where the virtual machine is the shell itself.

Other injections

Other types of command injections have been reported: LDAP-injection, XPath injection, XML injection, macro injection etc. More importantly, there are going to be new types of command

injections in the future, based on currently unknown technology. All of these types of attack are variation on the same pattern, so we must develop a defense strategy that is not based on the specificity of any given technology.

Commands injections versus commands modifications

One limitation of our definition and of the strategy described below is that it addresses command injections vulnerabilities, but does not address *command modification* vulnerabilities. A command modification attack consists of a malicious user modifying a legitimate command to transform it into an illegitimate one, but without inserting any new instructions. For example, in the case of SQL engines, it would mean to use an existing `SELECT` command to read different records than the ones intended, or to use a legitimate `DELETE` command to delete records that should not have been deleted. This type of attack is no less harmful than a command injection. It basically follows the same pattern but is not captured by Definition 1.

In fact, we argue that command modification flaws are of a different type entirely. When a command is "modified" in an application, it is the semantics of the application itself that is being abused. A purely syntactic approach such as the one proposed against command injections flaws is not appropriate for such a problem. It does not mean that this type of flaw can be overlooked, but rather that it will only be caught by an analysis of the inner workings of the application, much like the other security-related bugs in the application.

References

[1] C. Anley. Advanced sql injection in sql server applications. *http://www.ngssoftware.com/papers/advanced_sql_ injection.pdf*, January 2002.

[2] C. Anley. (more) advanced sql injection. *http://www.ngssoftware.com/papers/more_advanced_sql_ injection.pdf*, June 2002.

[3] Apache Software Foundation. Cross site scripting info: Encoding examples. *http://httpd.apache.org/info/css-security/encoding_examples.html*, November 2004.

[4] CGI Security. The cross-site scripting FAQ. *http://www.cgisecurity.net/articles/xss-faq.shtml*, August 2003.

[5] W. G. J. Halfond and A. Orso. Amnesia: analysis and monitoring for neutralizing sql-injection attacks. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 174–183, New York, NY, USA, 2005. ACM Press.

[6] D. Hu. Preventing cross-site scripting vulnerability. *http://www.giac.org/practical/GSEC/Deyu_Hu_GSEC.pdf*, May 2004.

[7] K. Spett. Cross-site scripting: are your web applications vulnerable. *http://www.spidynamics.com/support/whitepapers/SPIcross-sitescripting.pdf*, 2002.

[8] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 372–382, New York, NY, USA, 2006. ACM Press.

[9] T. A. Sudkamp. *Languages and machines: an introduction to the theory of computer science*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, second edition, 1997. ISBN 0-201-82136-2.