Securing Large Applications Against Command Injections

Guy-Vincent Jourdan[1]

[1]University of Ottawa, SITE
Ottawa, Canada
gvj@site.uottawa.ca

2007 International Carnahan Conference on Security Technology

Outline

Software security

An increasing concern...

Software security has been an increasing concern over the past few years

- Regularly in the popular news (CNN, CBC ...)
- Front and center in computing press
- Microsoft “trustworthy computing”, Oracle “Unbreakable”

Software security

... but still ways to go

Despite the increased attention, the computing world seem to be faced with an ever increasing number of reported security problems. Symantec's study, March 2006:

- 40% more vulnerabilities in 2005 than in 2004
- 69% of these vulnerabilities were coming from Web applications (60% in the first half of 2005 and 49% in the second half of 2004)
- Average of 49 days to release a patch correcting a vulnerability after it was released (down from 64 days in the previous report)
- Average of 6.8 days for an exploit to be released after a vulnerability was disclosed

Software security

Command injections

We focus on a particular category of software security vulnerabilities especially common in Web application: command injections.

- Often categorized into several different types of injections, each of these types being studied separately.
- Informal overview of the software vulnerabilities published within the last couple of years suggests that a majority of them belong to the category of command injections.
- Three of the OWASP's "Ten Most Critical Web Application Security Vulnerabilities" are command injections

Software security

Our contributions

- Provide a formal, generic definition of command injections
- Provide a practical, intuitive road map to address the issue in existing, large software applications
- Report on two real life experiments

About command injections

Command injections vulnerabilities are common and occur with different technologies, current and future. In order to efficiently protect application against this type of attacks, we need a definition of command injections which is technology independent and grasps the essence of the problem.

SQL injections

What are SQL injections?

Among command injections, SQL injections are perhaps the best known and the most studied. 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.

SQL injections

Example

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

SQL injections

Example

Because the query is built directly by concatenation of predefined commands and user input, malicious users can actually modify the end query.
With password = `' OR 1=1;` and username=`administrator`, "LoginQuery" becomes

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

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

HTML-browsers command injections?

HTML-browsers can be the target of several distinct command injections attacks. Dismissed at first as harmless, browsers injections are now understood as real threats, giving access to user's browsers, session, web applications etc.

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

Example: HTML injection

HTML page built on the fly from user provided `Username` and `Comment` with 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.

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

Example: Cross Site Scripting

HTML page built on the fly from user provided `Username` and `Comment` with the following server side ASP code:

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

If a malicious user enters a comment such as:

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

The following page is sent:

```
<B>name</B> says
 <script>alert(document.cookie)</script>
```

More examples

Other examples of command injections?

- shell command injections
- LDAP-injections
- XPath injections
- XML injections
- macro injections
- ...

More importantly, future injections based on new technologies.

Main idea

What is a command injection?

A command injection occurs when some input that is seen as data in a particular context becomes instructions in a new context .

Our approach

See applications as interacting “virtual machines” feeding “programs” to each other. Prevent users from interacting with the instructions of the programs.

Formal definition

Virtual machines

A virtual machine M accepts programs as input and "executes" these programs in some way. 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.

Formal definition

Virtual machines grammars

The grammar G_M has two types of symbols: the *terminal* and the *non-terminal* symbols.
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.

Formal definition

Virtual machines usage

Assume that an application makes use of a virtual machine M: it produces a program p recognized by G_M and sends p to M for execution.

If p is partially based some user input i_p, we say that there is a *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.

Formal definition

Definition

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.

The goal and the context

The formal definition of command injections gives us the foundation needed to define a strategy to protect applications against command injections by neutralizing potentially harmful inputs.

Our assumptions

We assume that we have to secure a large application that has been developed over many years by several teams of professional software engineers that are not security specialists. In other words, we can assume that there are several exploitable injection points to be found, that mistakes where made over the years and more mistakes will be made during the securing process itself.

The goal and the context

Our goal

Our aim is not to achieve perfect protection(too costly). We want to remove as many command injections vulnerabilities as possible during the time allocated for the securing effort, making it significantly harder for an attacker to find and successfully exploit an injection point.

1 - Virtual machines identification

The first natural step when securing any application against data commands injections is to identify every virtual machine used by the application. Failure to recognize one of these machines will lead to the application being potentially open to commands injections against that machine, regardless of the amount of effort spent securing the application against other commands injections attacks.

1 - Virtual machines identification

Common virtual machines

A *partial* listing of virtual machines commonly used today:

- SQL-based databases,
- XML parsers,
- HTML browsers,
- scripting languages embedded into HTML browsers,
- XSL transforms,
- LDAP servers,
- embedded applications macros,
- programming languages,
- shell commands.

2 - Injection points inventory

For a given virtual machine *M*, the first step is to create an injection points inventory. At that stage, an injection point is basically any interaction between the application and *M*, where data that will be recognized by *M* as parseable input is sent by the application.

2 - Injection points inventory

Approach

To create this inventory, one must carefully examine the API of M. Once the possible injection points to M are known, create an exhaustive inventory of the ones that are used in the application. This search can often be automated and can thus provide a complete and fully accurate result.
Note that this list can be very concise (e.g. SQL) or discouragingly long (e.g. XSL).

3 - Untrusted user data inventory

Once the injection points have been identified, the next step is to create an inventory of untrusted data used at these injection points.
Untrusted data is data that can be influenced, that is set or modified in any ways by any user.
Initially almost all input data is untrusted. Untrusted data must be carefully examined in order to evaluate whether it can in fact be trusted at the injection point, or if it should be neutralized against command injections.

3 - Untrusted user data inventory

Gaining trust

Untrusted input data does gain trust through data validation. The paths through which the data might have gone, and the various validation steps that have been performed along the way leading to the injection point must be researched, in order to decide whether or not this data can be trusted at this particular injection point.
Merely having gone through a step of validation is in no way enough for data to become trusted. The details of the actual validation steps must be evaluated to see their specific effectiveness against L_M-injections.

3 - Untrusted user data inventory

Being trusted

The reality is that our trust in the data simply increases when the data is validated. We are not trying to achieve complete, blind trust in the data, but rather to reach a level of trust that we deem sufficient in the context of the injection point.
The required level of trust has to be adapted on a case by case basis, taking into consideration the cost associated to raising it, the likelihood of an attack being successfully carried out with the current level of protection and the actual consequences of an L_M-injection.

3 - Untrusted user data inventory

Being trusted

The trust that we give to the data has to be understood as a contextual trust: it is related to the injection point and it is related to the virtual machine M. In other words, the data is not gaining any kind of general “trusted” status for a different purpose.

4 - Data neutralization

The last step in our process is the neutralization of the data that was identified as untrusted at injection points during the previous step (or rather that was not identified as sufficiently trusted). If we have proceeded correctly, then we should be in possession of the complete listing of potentially harmful data, and of the location where it can harm.

4 - Data neutralization

Data neutralization approach

Neutralizing data means to remove the possibility for the data to contain an element that is going to be recognized as a predefined constant of the language L_M by the grammar G_M.
If the machine M against which the data is neutralized works with a relatively simple grammar G_M, then it is possible to look at systematic approaches that would provably prevent L_M-injections.
In general, use simple neutralization rules making use of usual white lists mitigated with black lists: allow only known good, and if necessary, filter out, escape or encode known bad.

4 - Data neutralization

Data neutralization approach

The problem is greatly simplified by the realization that we are neutralizing data against L_M-injections, and thus the neutralization effort is clearly directed toward avoiding user-provided inputs that would be parsed as predefined constants of L_M by a G_M parser.

Experimental Results

The approach described here has been followed in some forms with various large-size Web applications, each involving several teams of professional programmers. In this industrial settings, one key requirement was the cost effectiveness of the effort, with significant results expected at reasonable cost.
Our technique has proved to be fairly effective, in that we were able to get large teams of programmers that had no prior knowledge of the problem to rapidly understand the issue and fully participate to the securing effort.
The results was verified by a formal security audit done subsequently.
See proceedings for details

Conclusion

What we have

- a formal definition of command injections which is completely independent from any particular technology
- a simple strategy that can be used to track and remove existing command injections of any types in a given
- a real life test for our strategy

What we need

- more automation in the strategy
- better virtual machine API that do not permit injections
- more awarness