Securing Large Applications Against 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

Abstract

The ability to produce more secure software or to improve the security of existing software is a growing concern and a real challenge for the field of software engineering. Among the various existing types of software vulnerabilities, command injections are particularly common. It is a difficult problem to address, having seemingly endless variations. We present here a unified, formal definition of command injections that, is not based on a particular technology and captures not only the existing variations but also the future instances of the problem. We then propose a simple, yet effective strategy to deal with the problem in existing large applications, focusing on the cost effectiveness of the method. We also report on successful experiments applying our solution to large commercial applications.

1 Introduction

Software security has been an increasing concern over the past few years, expending far beyond the restricted circle of security specialists and being now regularly featured on generalist news outlets such as CNN or the BBC. Microsoft, the largest software company in the world, launched its "Trustworthy Computing" campaign in January 2002 [9, 19]. Yet, despite the increased attention, the computing world seem to be faced with an ever increasing number of reported security problems.

In a study released in March 2006, Symantec reports that it was able to document 40% more vulnerabilities in 2005 than in 2004 [26]. Even more troubling for software engineers, 69% of these vulnerabilities were coming from Web applications, while Web applications represented "only" 60% of the vulnerabilities in the first half of 2005 and 49% in the second half of 2004 [26]; this shows a sharp increase in security vulnerabilities that are directly related to software quality, as opposed to other security issues such as unsafe user's behavior. Still according to that report, it took software producers an average of 49 days to release a patch correcting a vulnerability after it was released (which is still a big improvement over the 64 days in average of the previous report). Meanwhile, it took only an average of 6.8 days for an exploit to be released after a vulnerability was disclosed, leaving software users with an average of 42 days with an unpatched application having at least one disclosed and exploited security vulnerability.

Clearly, the software engineering community is struggling to produce secure applications. In fact, a 2004 study of over 250 Web applications showed that over 90% of them where vulnerable to "common hacking techniques" [30]. In [1], Adams and Jourdan point out that there is a large gap between the current software engineering practices as they are classically taught in universities across the world and the reality of secure software production: testing processes are not equipped to catch security related bugs, compartmentalization into components deliveries does not account for the global security of the system etc. In fact, it is worth noting that an overview of current software engineering books aimed at university level courses shows that these books are essentially silent on the question of producing secure software, presumably leaving the problem to specialized teams and specialized books [12, 6, 18]. However, countless security specialists have stressed the need for better security awareness and more secure coding practices among the core of software engineers. For example, the Microsoft security team has started publishing documentation on what they call the "Security Development Lifecycle" [16], an attempt to intergrate software security concerns to the software development life cycle.

In this paper, we focus on a particular category of software security vulnerabilities especially common in Web application, known as *command injections*. Command injections are often categorized into several different types of injections, each of these types being studied separately. It is our aim to study them as a whole, since we will show that they are indeed variations of the same problem.

A rapid overview of the software vulnerabilities published within the last couple of years show that a number of them belong to the category of command injections. For example, three of the "Ten Most Critical Web Application Security Vulnerabilities" in 2004, as reported in [27], were direct command injections vulnerabilities. There are various published approaches to deal with these vulnerabilities, ad hoc solutions as well as more generic ones. However, none of these solutions seem to provide a clear, technology independent description of the root cause of the problem, or an applicable road map to address the problem in large commercial software. In our research, our aim is to provide practical solutions to the problem, applicable in the context of industrial software development.

Our contributions are the following:

- We provide a formal, generic definition of command injections which captures the source of the problem and can be used to understand every current declinations of the vulnerability, as well as to anticipate future instances of it, using yet unknown technologies.

- We provide a practical, intuitive road map to address the issue in existing, large software applications. Our solution is quite open and does not seek perfection but realism. It is intended for teams of professional software engineers that are not security specialists but must adequately address security issues in their application, at reasonable cost and with tangible results.

- We describe two real life experiments where our ideas have been applied in the intended context. For confidentiality reasons, we cannot disclose products or company names, but we can report on the successes and limitations of the attempts and share some of the conclusions that we have reached during these experiments.

The rest of the paper is organized as follows: in Section 2, we give a formal model for command injections, and we illustrate it with some of the currently most common types of command injections. In Section 3, we outline a strategy to deal with command injections in existing large, commercial-grade software applications. In Section 4, we report on experimental results we have obtained applying our strategy in the context of large commercial applications. In Section 5, we discuss related work and we conclude in Section 6.

2 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.

2.1. 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. [25] 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.

2.2. SQL injections

Among command injections, SQL injections are perhaps the best known and the most studied (see e.g. [2, 3, 11, 24]). 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:

```
LoginQuery = ''SELECT * FROM UsersTable
    WHERE UserId=''' +
    request.getParameter(''userName'') +
    ''' AND Password = ''' +
    request.getParameter(''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. [4] for an example of possible consequences of allowing an "<a href=…" tag.

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).

2.3. 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" [23, 7, 13], 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.

2.4. 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.

2.5. 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.

2.6. 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.

3 A strategy to secure large applications against command injections attacks

The formal definition of command injections provided in Section 2 gives us the foundation needed to define a strategy to protect applications against command injections by neutralizing potentially harmful inputs. We assume here 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.

Our aim is not to achieve perfect protection, which would be in our case total riddance of command injections possibilities. Regardless of whether or not such a goal would be reachable, it would anyway be too costly a goal in our setting, involving too much efforts and too much time for the intended benefits. Instead, we want to remove as many command injections vulnerabilities as possible, making it significantly harder for an attacker to find and successfully exploit an injection point. Consequently, the strategy described here is not perfect or exhaustive, but is rather an attempt to remove efficiently as many command injection vulnerabilities as possible during the time allocated for the securing effort.

The strategy that we propose is in fact quite natural once command injections are understood. Having a straightforward approach to dealing with the problem is particularly helpful in our context, since the work will be carried out by teams of programmers not specialized in security. This securing effort can be concentrated within a short, focused timeframe dedicated to securing the product, and will likely be carried through by the teams who are in charge of developing or maintaining the application.

3.1 Virtual machines identification

As said in Section 2, there is a possibility for command injections when an application makes use of a virtual machine M that will interpret the data passed by the application as a program to execute.

The first natural step when securing any application against data commands injections is thus to identify every such 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.

A *partial* listing of the type of virtual machines that are commonly used today includes:

- SQL-based databases,
- XML parsers,
- HTML browsers,
- scripting languages embedded into HTML browsers,
- XSL transforms [29],
- LDAP servers,
- word-processing and spreadsheet embedded applications macros,
- programming languages,
- shell commands.

One should thus identify if any of these interpreters are used by the application being secured. However, this list is not exhaustive and is bound to change significantly over time, as technology evolves. In addition, there is also the possibility that the application being secured uses some non standard virtual machines that represent similar risks. It is thus important to spend the time necessary to identify any possible virtual machine used by the application at hand.

The application will have to be protected against harmful inputs for each of the identified virtual machines. In some cases, this may require more work that can be afforded, and it may be necessary to reduce the list of virtual machines for which protective measures will be taken. That is certainly a dangerous approach, but if the necessary budget is not available, the various virtual machines identified can be classified according to their perceived dangerousness. That is, for a given machine M, what would be the worst consequence of a successful L_M-injection? For example, if a successful HTML-injection could only lead to a web page being poorly displayed, this might be deemed as being an acceptable risk and the associated neutralization process being postponed to be able to focus on higher risk virtual machines, for example neutralizing the input against HTML-browser scripting languages.

Once we are convinced that all virtual machines have been identified, we need to apply the steps listed below for each such virtual machine.

3.2 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.

In order to successfully create this inventory, one must carefully examine the API of M to make sure that no possible potential injection point is missed. Albeit tedious, this step should be usually easy to accomplish, unless M happens to be improperly documented. Once the possible injection points to M are known, it should be relatively easy to 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. We thus assume that this step will lead to an accurate and exhaustive listing of the application's injections points for the machine M.

Note that depending both on the application and M, this list can be very concise or discouragingly long. For example, for SQL-injections, when the application is well structured it is not unusual to find relatively few injection points, since the application has typically relatively few actual interactions with the database[2], and moreover these interactions are typically grouped together in some data access component. On the other hand, a typical XSL transformation will be literally covered with potential injection points (HTML, XML, scripting . . .), since by design almost every line of an XSL document transforms its input into potentially harmful output. Obviously, cases such as the latter are going to be much more challenging to deal with, which might explains why most currently published material focus on the former.

3.3 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. Unlike the previous step, this step is not going to be easy to automate, and will require manual processing and user's judgment.

Untrusted data is data that can be influenced, that is set or modified in any ways by a malicious user. Moreover, this modification does not necessarily need to be direct; in other words, it is not sufficient for the "last" user or the last point of storage to be trustworthy for the data to be trusted. Indeed, data coming from an apparently trusted source such as the application own database might have been at one point modified by a malicious user and thus might contain malicious content. Similarly, all users must be considered potentially malicious since a trustworthy, authenticated user might be led to submit malicious data by various means, such as social engineering [10, 20] or session hijacking[15].

Clearly, with such a broad definition, 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.

[2]Even when the application is database intensive, one rarely sees a code making hundreds, let alone thousands of different database calls along the way while processing a request, if only because it would be very inefficient.

3.3.1 Gaining trust

Untrusted input data does gain trust through data validation [28]. In other words, 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.

As pointed out in [8], validation steps that are performed at various location within the application may not be very effective at preventing command injection attacks, for a variety of reasons including the fact that when the validation is performed, it might still be unknown that the input would end up being passed to the virtual machine M, and that in addition, the data might not be in its final form yet. Consequently, 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.

3.3.2 Being trusted

We said that trust is gained through data validation. 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.

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. This approach is diametrically opposed to the concept of "trust boundaries" (see e.g. [12]). With the trust boundaries approach, data is validated as it enters a component and is then trusted within this component. We depart from this model because, as suggested above, it is usually difficult and sometimes impossible to effectively validate data against all types of commands injections before the data is actually used in the context of being harmful. That is not to say that we disregard the value of systematic data validation at "chock points", but rather that we do not necessarily trust this validation step to always protect against all possible L_M-injection for any machine M.

3.3.3 Reducing the size of the inventory

The goal of this step is to reduce as much as possible and as quickly as possible the amount of untrusted data at injection points, since this is the data for which there is a risk and for which we will need to act.

In practice, we find that we will quickly find some "clusters" of injection points that we will be able to group together because they present the same type of attack vector. For one such cluster, we will typically have identified some common steps of validation which, if done, are deemed sufficient to trust the data. This sort of natural grouping allows to quickly eliminate large set of untrusted data at injections points.

Whether or not the particular application offers natural clustering of injection points, we nevertheless need to go through the complete set of untrusted data manually, which is typically a time consuming, labor intensive process whose outcome is subject to error. Great care should be taken while performing this step, since data that is wrongly classified as trusted will obviously lead to a security flaw remaining in the application.

When using some particular environments, there are tools available to help (see e.g. [17] for Java). These tools can be of great help to trace the data flow from the injection point back to the entry point(s).

3.4 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.

Neutralizing this untrusted data means that we want 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. In order to achieve this, several techniques are available. 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. For example, Su and Wassermann describe in [24] a method consisting of tagging what we call here the variable terminal symbols of G_M and then simulate a parser for G_M to see if it derives the existence of predefined constants of L_M within tags. Other approaches are based on static analysis of the code and automatic inference of monitoring tools [14, 17, 11].

Although these techniques might works under some circumstances, we seem to still be some ways away from being able to protect against general commands injections using them. The use of these techniques did not seem appropriate in our settings, where the work must be performed by teams of non specialized programmers under strict time constraints, applied to existing large applications. Instead, we advocate the usage of 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 (see e.g. [21, 28]). 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.

4 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.

In this section, we report on two experiments, where teams of application programmers were asked to participate in "application scans" aimed at auditing large software application to look for command injections vulnerabilities.

4.1. First application scan

The first scan was carried out on a large commercial application that was using a proprietary technology similar to XSLT [29]. The focus was solely on that technology, that is, securing against commands injections vulnerabilities against a specific virtual machine. The application had hundreds of "transformation" documents. The scan involved a team of about ten software engineers, involved for several days, scattered throughout several weeks.

During this first scan, we did not create a precise inventory of the command injection points. Instead, we simply identified the types of injection points that were to be found in the application. For the inventory of untrusted data, we used the fact that in this particular application, a significant part of the data could be trusted because it was coming from a database that was not writable by users. This allowed us to create a set of trusted data. Moreover, we were able to automatically infer this set from the way this data was read from the database. We decided to flag the rest of the application input data as untrusted.

The neutralization effort was therefore to manually neutralize the data that was flagged as untrusted, which was still about 60% of the input data. This meant several thousands of manual neutralizations. In order to deal with such a large effort, we attempted to ease the process by enhancing the virtual machine itself, adding "self neutralization" routines that were able to neutralize data according to its type, which greatly simplified the syntactic implication of updating the documents.

One of the outcomes of this first experiment was that some programmers ended up mis-typing some of the more complex data, due perhaps to the repetitive effort of adding the types to thousands of virtual machine calls. For example, in some instances, an integer variables assignments in JavaScript (of the form `intVar = <input>;`) ended up being erroneously neutralized as a JavaScript string.

The conclusion was that the repetitive and tedious pattern of manual neutralization over many days prevented the programmers to detect some of the non-standard instances of required neutralization. More importantly, the broad scope of the effort was due to the lack of trust in the data. We had an algorithm to automatically trust some of the data and we simply decided to not trust anything else. Had we spent more time building the inventory of trusted data, paying more attention to the trust we could have in the existing application level validation, we would have been able to significantly reduce the amount of required neutralization. Moreover, the narrow definition of "trusted data" implied that the data that was flagged as untrusted was actually unlikely dangerous. With a more stringent definition of "untrusted", the programmers would have kept their guard higher.

Interestingly, another downside was excessive neutralization, were valid input was "neutralized", affecting the application functionalities at places and increasing the performance price that had to be paid for neutralization. Again, we believe that the root cause of the problem was the size of the untrusted data list.

It should be noted that in the end, despite the issues outlined above, the scan produced a good coverage of the application and was deemed a success, as verified by a formal security audit done subsequently. But the cost paid to reach that level was high.

4.2. Second application scan

For the second scan, we were dealing with a larger project, involving about twenty programmers from about seven teams. The technology used was similar to the one of the first scan.

This time, we created an exhaustive inventory of command injections points ourselves, using automated scripts and without involving the team of application programmers. After having obtained this listing, we educated the software engineers about command injections using the same listing and focusing solely on the types of command injections that would be encountered during the scan (in the previous scan, the training was a general, text book injection training). The teams were then instructed to find untrusted data at the injection points themselves. This time, we did not attempt to narrow down the set of untrusted data automatically, and let the programmers do the work from the ground up. They in fact did find again rather quickly types of data that could be automatically trusted, but then went beyond this initial set and looked at the validation processes that could be found in the application. This further reduced the inventory of untrusted data and increased the team's understanding of the dangerousness of the data that was not flagged as trusted. Programmers were also requested to add comments as to why they flagged data as trusted.

This proved to be a better approach, since a lot fewer mistakes were subsequently caught, showing that the teams had been more effective in their neutralization process. It should be noted that a total coverage was not achieved the first time through. We were able to find data that was neither flagged as trusted nor neutralized, and had simply been missed by the teams.

Overall, high effectiveness in the neutralization process of the data with the highest risk for attacks was achieved, as verified by a subsequent formal security audit.

One additional positive outcome of this scan effort was that over the following months, programmers involved in the effort started to flag to the security team instances of possible command injections, including command injections against virtual machines that were not considered during the scan, showing the benefits of the method in term of education. The simplicity of the method made it easy for the programmers to add command injections prevention as part of their development routine.

5 Related work

Various existing methods exist to try to address the problem of command injections in different ways. Perhaps the simplest way (from the view point of methodology) is *penetration testing* [5], were specialized teams of testers attempt to find security vulnerabilities in a system by providing the type of inputs a malicious user would provide. This method can be quite effective in some cases but is clearly limited and cannot provide any guaranty of quality. It is also difficult to implement on large applications that require specialized knowledge to be used.

A more systematic way of dealing with the issue can be found in input filtering techniques [21, 22], where acceptable user inputs can be specified and enforced. This type of technique is fairly natural and tools are available to help implementing it, both within the current programming environments and as independent tools. The main limitations that we see with this technique lies around the difficulty of efficiently neutralizing data against command injections vulnerabilities at the user input level, where the context is lacking and some crucial information regarding the destination of the data might be missing.

Automated solutions based on static analysis of the code have also been suggested [14, 17, 11]. These techniques are useful to automatically link the command injection points to user inputs, although they are not perfect and can miss some of the links. Once the path from the user input to the injection point is detected, much is left to be done in order to decide whether the data can be trusted. Some of the tools permit to specify the acceptable form of the user input (or even automatically infer this form [11]) and subsequently enforce compliance by rejecting non conforming inputs at runtime. These tools seem to be good helpers for our method, although perhaps slightly limited in their ability to handle large scale application and massive injection points inventories as in the examples of Section 4.

From a software engineering viewpoint, the classic notion is the one of "trust boundary" [12], where the application is split into components and data is validated as it crosses the boundary between two components. On the other hand, once inside, the data is trusted within the component. We believe that the practical limitation of this approach are similar to the input filtering techniques: in practice, it might be difficult to neutralize data against command injections at some boundary that is not the command injection point, if only because the ultimate destination of the data might be unknown or unsure and the data might simply be a small piece of a larger eventual input. We do believe that data validation and data neutralization do serve a different purpose and should usually be performed independently [8].

A special note should be made of the solution provided by Su and Wassermann in [24]. They propose to tag what we have called the variable terminal symbols of G_M and to then simulate a parser for G_M. If that parser infers the existence of predefined constants of L_M within tags, then a command injection has been detected. The solution is the only one to our knowledge that would provably prevent any possibility of command injections in the tagged data. This technique still requires finding the untrusted data at injection point, but could be used in our method to perform the neutralization step. Unfortunately, this method seems to be again somewhat limited in practice, since it requires the ability to properly tag the data and moreover assume the availability of a G_M parser simulator, which is perhaps possible with simple grammar but much more difficult with complex and in practice poorly standardized languages such as HTML.

6 Conclusion

In this paper, we provide a formal definition of command injections which is completely independent from any particular technology. Our formalization goes at the heart of the problem and captures seemingly widely different types of attacks, such as SQL injection, cross-site scripting, XML-path injections or shell command injections. More importantly, it also captures yet unknown command injections vulnerabilities based on future technologies, and allows to take preemptive measures against them and not be taken off guard when they appear.

We then propose a strategy that can be used to track and remove existing command injections of any types in a given application. The simple and effective strategy is meant to be cost effective and is openly targeted toward large commercial applications. Our practical solution can be implemented within large teams of professional engineers that are not security specialists and will help deliver more secure applications under realistic assumptions of time, manpower and budget.

We have tested the proposed solution in its intended context, resulting in large application being secured by team of software developers and being subsequently certified secure by formal security auditors. We describe these experiments and draw several conclusions for them.

Acknowledgments

This work is supported in part by the Natural Science and Engineering Research Council of Canada under grants RGPIN 312018.

References

[1] C. Adams and G.-V. Jourdan. Why good software engineering practices often do not produce secure software. In *IEEE Workshop on Cyber Infrastructure Emergency Preparedness Aspects*, Ottawa, ON, Canada, April 2005.

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

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

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

[5] B. Arkin, S. Stender, and G. McGraw. Software penetration testing. In *IEEE Security & Privacy Magazine*, pages 84–87, 2005.

[6] M. Bishop. *Introduction to Computer Security*. Addison-Wesley, November 2004. ISBN 0-321-24744-2.

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

[8] S. de Vries. A modular approach to data validation in web applications. *http://www.corsaire.com/white-papers/060116-a-modular-approach-to-data-validation.pdf*, March 2006.

[9] B. Gates. Trustworthy computing. *http://www.wired.com/news/business/0,1367,49826,00.html*, January 2002.

[10] S. Granger. Social engineering fundamentals, part I: Hacker tactics. *http://www.securityfocus.com/infocus/1527*, December 2001.

[11] 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.

[12] M. Howard and D. LeBlanc. *Writing Secure Code*. Microsoft Press, second edition, 2002. ISBN 0-7356-1722-8.

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

[14] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 40–52, New York, NY, USA, 2004. ACM Press.

[15] K. Lam, D. LeBlanc, and B. Smith. *Assessing Network Security*, chapter 21, pages 333–354. Microsoft Press, June 2004. ISBN 0-7356-2033-4. http://www.microsoft.com/technet/technetmag/issues/2005/01/SessionHijacking/default.aspx.

[16] S. Lipner and M. Howard. The trustworthy computing security development lifecycle. *http://msdn.microsoft.com/security/sdl*, March 2005.

[17] V. B. Livshits and M. S. Lam. Finding security errors in Java programs with static analysis. In *Proceedings of the 14th Usenix Security Symposium*, pages 271–286, Aug. 2005.

[18] G. McGraw. *Software Security: Building Security In*. Addison-Wesley, February 2006. ISBN 0-321-35670-5.

[19] Microsoft. Trustworthy computing. *http://www.microsoft.com/mscorp/execmail/2002/07-18twc.asp*, July 2002.

[20] K. Mitnick and W. L. Simon. *The Art of Deception*. Wiley, 2002. ISBN 0-471-23712-4.

[21] D. Scott and R. Sharp. Abstracting application-level web security. In *WWW '02: Proceedings of the 11th international conference on World Wide Web*. ACM Press, 2002.

[22] D. Scott and R. Sharp. Specifying and enforcing application-level web security policies. *IEEE Transactions on Knowledge and Data Engineering*, 15(4):771–783, 2003.

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

[24] 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.

[25] 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.

[26] Symantec. Symantec internet security threat report - trends for july 05 - december 05. volume IX. *http://enterprisesecurity.symantec.com/pdf/ISTR_IX_FullReport.pdf*, March 2006.

[27] The Open Web Application Security Project (OWASP). The ten most critical web application security vulnerabilities. *http://www.owasp.org/documentation/topten.html*, January 2004.

[28] The Open Web Application Security Project (OWASP). Data validation. *http://www.owasp.org/index.php/Data_Validation*, June 2006.

[29] W3C. XSL Transformations (XSLT). *http://www.w3.org/TR/xslt*, November 1999.

[30] WebCohort Inc. Only 10% of web applications are secured against common hacking techniques. *http://www.imperva.com/company/news/2004-feb-02.html*, February 2004.