Data Validation, Data Neutralization, Data Footprint: A Framework Against Injection Attacks

Guy-Vincent Jourdan*

School of Information Technology and Engineering, University of Ottawa, 800 King Edward Avenue, Ottawa, Ontario, Canada, K1N 6N5

Abstract: Untrusted data validation is an important part of software security, yet most current validation techniques fall short in two ways: they lack practicality when it comes to validating data in large scale, real life applications, and they do not clearly identify the different goals of handling untrusted data securely. In this paper, we clarify the different, independent problems that "data validation" should solve, and we provide a clear and detailed three step process to data validation: a "data validation" step to protect the application itself against malicious users, a "data neutralization" step to protect other applications from malicious users of the application, and a "data footprint" step to protect against attacks on future, unforeseen components that will be connected to the application.

1. INTRODUCTION AND MOTIVATIONS

Securely handling untrusted data is an important part of software security, yet it is a question that is often overlooked by security researchers. Indeed, most of the academic focus in the security domain is on cryptographic systems, secure protocols, intrusion detections etc. In other words, a lot of attention is (rightfully) given to security software and the security features of non security software, but the over-all security of the resulting software does not seem to receive the required attention from the academic world. This is particularly unfortunate since a large number of security vulnerabilities published over the last few years come from exploitation of non security software, in parts that are not dealing with the existing security features of the software. In fact, the software industry seems to have taken the lead on that front. For example, after having lunched its "Trustworthy Computing" campaign in January 2002 [1, 2], Microsoft has started publishing documentation on what it calls the "Security Development Lifecycle" [3], an attempt to integrate software security concerns into the software development life cycle, while current software engineering books aimed at university level courses are essentially silent on the question of producing secure software [4, 5].

Although there have been some (mostly industry led) efforts to improve the overall security of software, much is left to be done. A 2004 study of over 250 Web applications showed that over 90% of them were vulnerable to "common hacking techniques" [6]. A rapid overview of the software vulnerabilities published within the last couple of years shows 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 [7], were direct command injection vulnerabilities.

*Address correspondence to this author at the School of Information Technology and Engineering University of Ottawa 800 King Edward Avenue, Ottawa, Ontario, Canada, K1N 6N5; E-mail: gvj@site.uottawa.ca

These types of attack should not be confused with control flow attacks, which have been know since at least 1972 [8] and have been widely researched ever since. Control flow attacks, sometimes also referred as command injection attacks, are in fact based on the attacker ability to direct the program counter to a location where some malicious code has been put, usually using a bug in the host application such as a buffer overflow [9]. Protecting against this type of attacks calls for a better control over the program counter [10-12] or a hardware or software based separation between code memory and data memory [13]. Despite some similarities, command injections as defined in this paper are much broader and do not necessitate neither a bug in the host application nor a control over the program counter. Therefore, techniques to prevent control flow attacks are unfortunately not able to contain command injection attacks as defined here.

In this paper, we look at the question of data validation, that is, how to safely process data coming from an untrusted source, throughout the application, in order to avoid vulnerabilities such as command injections. We argue that most of the existing published solutions fall short of their expectations in several aspects: they are usually unclear about the goal of data validation, they are at odds with good software engineering practices, and they are not really usable when dealing with real, medium to large size software systems (see Section 3 for a detailed explanation of this claim). We propose a new solution based on a three steps framework to address these issues. Our solution breaks down the "data validation" efforts into two, clearly identified, formally defined and clearly localized steps. As we will show in Sections 4 and 5, this solution integrates well with good software engineering practices. In particular, it avoids the duplication of efforts often encountered in other approaches, it does not make unrealistic assumptions about the application or the environment, and has been successfully used with real life large web applications [14].

Our contributions are the following:

1874-107X/08

- We provide a formal definition of command injections, and formally identify the two separated goals that "data validation" should reach: protecting the application itself and protecting the other applications that are used by the application being secured.
- We show that most current data validation techniques fail to address the question in an effective way for large applications and that even with simple, small applications, these techniques are mostly incompatible with good software engineering practices.
- We provide a practical and intuitive three steps framework that can be used to achieve effective data validation even with large software. Our framework is compatible with modern, good software engineering practice and provides clear guidelines about what to do and when to do it. The framework also accounts for future problems linked to the processing of untrusted data that cannot be known at the time of coding.

The rest of this paper is organized as follows: in Section 2, we introduce the problem of command injection and give a formal model for it. In Section 3, we give an overview of some of the current data validation techniques, and show their limitations. We then introduce our three step framework in the next three sections: Section 4 covers the data validation step, where the application itself is protected from untrusted data. Section 5 covers the data neutralization step, where other applications used by the application are protected. And Section 6 describes the last step, the data footprint, used to handle future evolutions of the application's environment. We conclude in Section 7.

2. PRELIMINARIES

2.1. Command Injections

In this section, we give a quick overview of the most common command injection vulnerabilities and then we provide a general, formal definition which can be used to explain current and future instances of the problem.

2.1.1 Examples of Command Injections

What we call is this paper "command injection" is a type of vulnerability that is very common today. Many different technologies can be exploited, and as new technologies are introduced, new command injection vulnerabilities opportunities arise. Unfortunately, all of these vulnerabilities are not always understood as a variation on the same problem, leading to an array of specialized defense mechanisms and an inefficient approach to solving the problem overall.

In today's typical application environment, there is a large spectrum of tools that are susceptible to command injection attacks, including but not limited to SQL-based databases, XML parsers, HTML browsers, scripting languages embedded into HTML browsers, XSL transforms [15], LDAP servers, word-processing and spreadsheet embedded application macros, interpreted programming languages, and shell commands.

Shell command injection vulnerabilities are historically important, although they seem to have become less common lately. This type of command injection occurs when the application invokes the operating system shell (C-shell, Bash etc. on Unix, command shell on Windows etc.) to initiate another program, such as the "grep" or the "mail" program under Unix. If, as part of this program invocation, the application is using some untrusted data without proper filtering, a malicious user can craft an input that will terminate the intended command and start another one of the attacker's choice.

SQL injections are perhaps the most well known among command injection vulnerabilities and have been extensively studied (see e.g. [16-19]). An SQL command injection vulnerability exists whenever an application uses an SQL based database and constructs unfiltered (or improperly filtered) SQL commands based on untrusted input. An attacker can then use this opportunity to inject an SQL command, which will be sent by the application to the SQL-database engine and executed. As an example, consider the following SQL query, assumed to be built on-the-fly on the server. The intent is to query a 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".

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

Since the query is built directly from user input, malicious users can actually modify the end query in various ways. One of many ways to exploit such code is to bypass the password verification by providing a password such as `'OR 1=1;--`. If the user name provided is, for example, `administrator`, the query "LoginQuery" that is sent to the database ends up being[1]

```
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.

HTML-browsers can also be the target of command injection attacks. As a simple example, assume that the application produces an HTML page containing user-provided comments to be displayed in other users' HTML browsers. If the application has stored the user nickname and comment into the `NickName` and `Comment` variables, and if the HTML page showing the user-provided comments is built from the following server side ASP code:

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

[1]In SQL, "--" is the beginning of a comment.

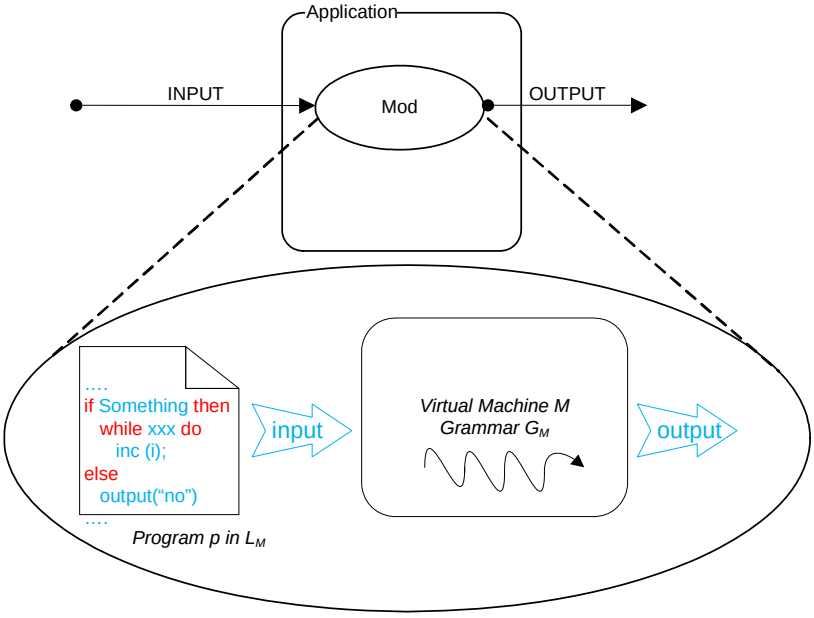

Fig. (1). An application seen as a virtual machine *M*: the input of the module Mod is a program *p* written in the language L_M, recognized by the grammar G_M of the virtual machine. Processing this input is the execution of *p* by the virtual machine. *p* contains keywords (in red) and literals (in blue).

then if a malicious user gives a nickname or a comment that includes HTML tags, 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. For example, that attacker can easily insert an entire fake login form similar to that used by the application. It will lie within a legitimate page of the application but can send the credential to some other location, controlled by the attacker. If instead the attacker injects a command such as

```
<script>
location.href='http://attack.com/?'
            +document.cookie
</script>
```

then the JavaScript interpreter that is embedded in the browser will be invoked and the user's cookies will be sent to the attacker's server, which can lead to the victim's session being hijacked (this particular type of injections is known as *Cross Site Scripting*, or *XSS*).

2.1.2 Formal Definition of Command Injections

The various examples in Section 2.1.1 show variations of the same type of attack. Here, we give a formal, technology independent definition of the problem.

As seen in the previous examples, the root cause of the problem is the interaction between the application and another component (database server, HTML browser etc.), where something that is simple data within the application becomes command(s) in the other component. Thus, the vulnerabilities lies in the ability of the attacker to inject data into the application that is going to be interpreted as a command by one of the components used by the application.

Let us consider that the application makes use of *virtual machines*, without any assumptions about what these virtual machines actually are. A typical virtual machine *M* accepts as input "programs" written in a particular language L_M, as specified by a grammar G_M. Provided with such a valid program, the virtual machine executes it. So, for example, the "HTML browsers" virtual machine accepts programs written in HTML, while the virtual machine "SQL database" accepts programs written in SQL. Fig. (**1**) provides an illustration of this concept.

As usual (see e.g. [20] for an overview of these classical concepts), these programs are *words* of L_M, made of terminal symbols of G_M that can be derived from the rules of G_M. The terminal symbols in G_M can be split in two categories: the keywords of the language L_M and the variables (or *literals*). The keywords are the predefined symbols of the language that are interpreted by M upon execution of a program, while the literals specify values, variable names etc.

An application making use of such a machine *M* has to produce a program *p* recognized by G_M and has to send *p* to *M* for execution. We say that there is a *command injection vulnerability* (more precisely in that case, an L_M *-injection* vulnerability) when *p* is at least partially generated at runtime, based in part on some untrusted data i_p, and when there exists some i_p for which the resulting *p* contains elements from i_p that are going to be recognized as keywords of the language L_M by the grammar G_M.

Fig. (**2**) provides an overview: an untrusted user sends some input to a first application, Application 1. This input is processed inside Application 1 in the module Mod 3, where it is combined with other data coming from the database Data App 1. The resulting data is eventually output to Application 2, where it is processed inside module Mod 5, combined with data coming from Data App 2. In turn, this eventually produces an output that is given as input to Application 3. This input is itself processed by module Mod 6. When we abstract Mod 6 as a virtual machine *M* accepting programs written in the grammar G_M, the input received from Application 3 is a program *p* written in the language L_M of G_M. The program *p* is produced from different sources,

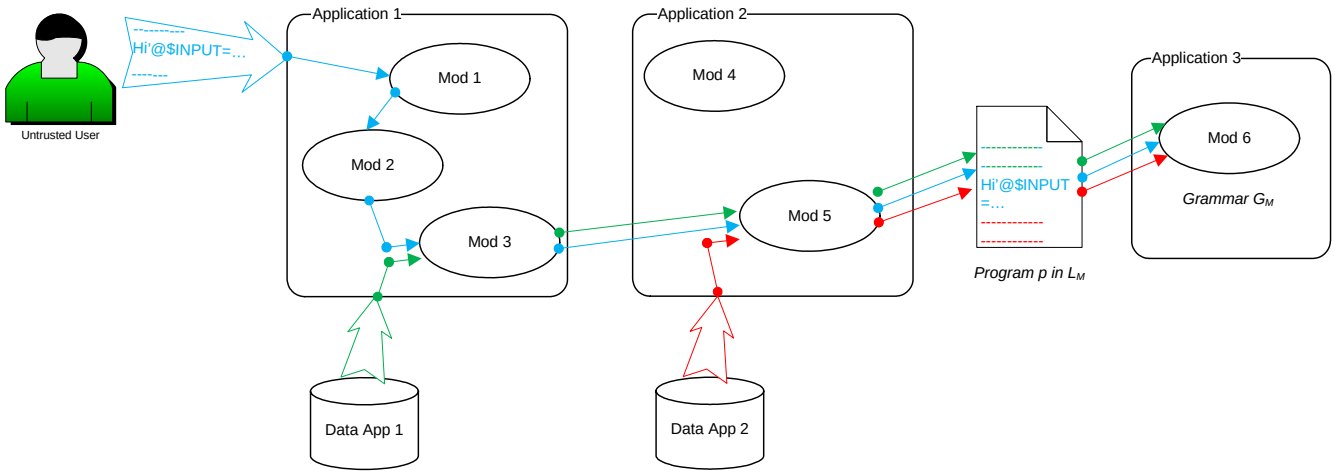

Fig. (2). An overview of the interaction between a user and a sequence of applications. Data entered by the user will be handled by the applications 1, 2 and 3, in different contexts each time.

including input from the user, data from Data App 1 and Data App 2, as well as processing from at least Mod 3 and Mod 5. If there is a way for part of the user input to end up in p as keywords of the language L_M when parsed by the grammar G_M, then there is an L_M-injection vulnerability, and the untrusted user can control to some extend the execution of Mod 5 inside Application 3.

Definition 1 (L_M-injection vulnerability) *An application has an L_M-injection vulnerability if the application uses a virtual machine M and it is possible for a user to provide a set of inputs i_p to the application that will cause the application to generate a program p sent to M such that there are in p some elements coming from i_p that will be parsed by the grammar G_M as keywords of the language L_M.*

In other words, a command injection vulnerability is a flaw that allows a user 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.

2.2. Data Normalization

Before the actual validation effort, it is useful to first normalize the data. This step is often known as *data canonicalization*[2] (see for example [21, 22]). The goal is to massage the data to put it in its canonical form, defined as "its simplest and standard form". We prefer to use "data normalization", because the notion of data's "simplest form" is neither well defined nor very relevant to this problem.

Data normalization is the process of transforming the data into some predefined format. A very common example of normalization is encoding format: there are many different ways of encoding the same input, for example using different character sets. Therefore, if no preprocessing is done, the data validation step will have to guarantee that whatever is done during validation is done effectively regardless of the format. Otherwise validation would simply be bypassed, e.g. by using a different character set. This significantly complicates the validation process. It is thus more efficient to first *normalize* the encoding format, e.g. by encoding the input into some predefined character set prior to the validation process.

Character sets are just one of the issues that normalization concerns itself with. It is relevant when there is more than one way to provide an equivalent input to the application, and these different ways have an impact on validation. For example, in most current operating systems there are many different ways of naming a file, using absolute or relative paths, mixing up actual path and parent path (e.g. dir_1/dir_2 and $dir_1/../dir_1/dir_2$ are equivalent on Unix), using symbolic links, mixing cases on case-insensitive operating systems or using Universal Naming Convention shares, to name but a few. Another example would be protocols allowing that some of the messages, or some of the information inside the message, be given in different orders, or even repeated several times (see for example [23]).

It should be noted that the goal of the data normalization step is to put the data in a predefined format; the actual format chosen, "simplest" or not, is thus not particularly relevant.

3. EXISTING APPROACHES AND LIMITATIONS

Data validation has long been identified as an important part of software security. However, we will show that the current approaches to data validation do lack both practicality and clarity.

One of the main theories about data validation is that one should use "white lists" (permitting what is known as correct input) rather than "black lists" (blocking inputs that are known to be harmful). This is a fair idea as such, but it does not help in understanding what it is that validation is trying to protect against, nor does it help understand where to validate. As we will see, this approach is still very relevant, but cannot be considered a validation technique on its own.

The most common validation technique uses the idea of "trust boundaries" [24-26]. In this approach, we are

[2]Or even *C14N*, standing for "C, then 14 letters, then N".

supposed to identify various zones inside the application within which the level of trust is similar. The main idea is that whenever data crosses a trust boundary, that is, crosses the frontier between two of these zones, the data should be validated. The theory sometimes includes the definition of *chokepoints* [24] that should be used when crossing a trust boundary. Chokepoints are the possible points of entries into the guarded boundaries, where data must be validated, although the scope of this "validation" is usually not very clear.

Trust boundaries are a natural idea, that fit rather well with the natural security intuition of *gatekeepers* that are used to ensure that whatever enters a zone is "safe". Unfortunately, in real life, there are several problems with this view:

- The notion of trust boundaries has no particular correspondence to any architectural boundaries. In other words, a "trusted zone" may slice the software architecture in any way, meaning that the chokepoints will have a difficult time integrating with the actual software architecture. Having a security architecture at odds with the rest of the software architecture is surely not a good situation.
- As a consequence of the previous point, it is very likely that a given module can be reached through several chokepoints and trust boundaries. In fact, the target module for a particular input can be quite remote from the chokepoints, and several chokepoints might lead to the same module via different routes, crossing the trust boundaries through different paths. In other words, the validation of the data leading to a particular module will be done away from that module, and will have to be done at different locations, once for every path leading to that module and crossing the trusted boundary. This is clearly in opposition to good software engineering practices, where the same work should be done only once. To make things worse, because a particular module's data is validated away from that module, at the current trust boundary, any extension to the software that will reuse the module will likely re-create a new trust boundary crossing, and thus a new duplication of the verification. And of course, the same problems occur if we want to reuse the module: the validation will have to be redone in the new context for the reused module to be secure.
- Last, but not least, there is just no reason to assume that when crossing the trust boundary, the data is in its final form. If it is not, then it means that we are attempting to validate partial information, a futile exercise without much doubt. And here again, even in the favorable case where the data is reasonably complete when crossing the boundary, validating it here requires an understanding of the expected valid format. When looking at a simple application, that seems to be fairly doable. But when the data is part of a large, complex application, it is often the case that the data can not be simply validated by looking at a simple value range. Instead, a complex evaluation is required, which basically requires the application logic to be restated at the validation point. This is again at odd with any good software engineering practices, which states that the logic of the application should be coded once and maintained at that one place. Following the trust boundaries paradigm means that the logic of the application must be duplicated and maintained at every chokepoint, a situation that is clearly to be avoided.

Another common approach to data validation, particularly popular with Web applications, is to perform field by field the validation of the data entered in a form. This idea is presumably common because the current programming environments offer excellent support for it, with advanced field-level and form-level validation functions that can be used to easily enforce some input formats (see e.g. [27, 28] for SUN's J2EE [29] and [30, 31] for Microsoft® .NET [32]). This approach, however convenient, suffers from the same flaws as the trust boundaries approach: the validation occurs outside the module that handles the data, leading to a duplication of logic, a duplication of code, poor modularity and maintenance problems. In addition to these very serious problems, we are faced with an overall strong limitation of what can actually be validated since a very limited amount of data is available at the validation point. Basically, one has to validate based on the data present in the current form (or current screen). This approach can only work with simple applications (such as most current Web applications) but will not scale to more complex scenarios[3]. Finally, this approach is typically only available in the limited context of inputs via the user interface. If the application is extended to provide other types of interactions, these other types of input may not have this solution available.

Pushing the logic even further, in some contexts it is sometimes suggested to validate data through an *application firewall*, such as [33]. This obviously suffers from the very same problems as the techniques described above, since an application firewall is situated outside the application, even further away from the end module that will process the data. Note that in some cases, application firewalls are the only option available (e.g. when the application source code is not available or cannot be changed) but that does not change the fact that application firewalls are a very poor solution to data validation[4].

In [34], Stephen de Vries proposes a validation approach which attempts to solve some of the issues outlined above. This is the only such attempt that we are aware of at the time of writing. Recognizing the fact that, with real, large applications, validation cannot be done just about anywhere, de Vries proposes a framework with a "validation" step in the "business object", that is, where the context of the data

[3]We are not suggesting not to use these types of tools; good defense-in-depth security suggests that an additional layer of security is always a good thing. However, we argue that one should not rely purely on these tools to validate untrusted inputs in an application.

[4]Here again, we are not suggesting not using application firewall. In fact, we believe application firewalls support should be part of any secure distribution, since they are a very efficient way to quickly react to a problem and secure an installation until a proper, fully tested patch can be distributed.

usage is known, and an "encoding" step at the data access layer. His approach is compatible with the one proposed in this article. However, de Vries does not clearly define, formally or informally, the precise goals of the "validation" step and the "encoding" step. Moreover, the suggestion that the encoding step should be "performed close to where the data is processed" [34] does not seem precise enough. The closer we move to where the data is processed, the more likely user-provided data and application provided data will be mixed together, and thus the less likely it will be that injection attacks will be caught. Finally, the suggestion to use a specific data access object to "encode" the data for each target "processing context" ignores the problem of multiple processing contexts, where the data is aimed at a series of contexts (e.g. the data is sent to a database (SQL context) to be eventually rendered in an HTML browser (HTML context) with scripting support such as a JavaScript interpreter (JavaScript context)).

A few other papers have been published on the topic, but they typically focus on a very specific sub-problem and one possible solution. As such, these techniques are typically compatible with our proposal, which focuses on what should be done and where it should be done, rather than how it should be done (for which we refer to existing techniques). Among others, we can refer to [35, 36], where a systematic way of dealing with user inputs with filtering techniques can be found. In [18, 37, 38], we can find solutions based on the static analysis of the code to automatically link the command injection points to user inputs. These solution are not perfect and can miss some of the links, and even 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. The solution provided by Su and Wassermann in [19] is 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 keywords 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. Unfortunately, this method seems to be 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.

In the following sections we provide our own solution to the validation problem.

4. DATA VALIDATION

4.1. Goals of the validation steps

In our framework, we split the overall untrusted data validation process into two steps, each having a well identified purpose. The first step, called the *validation step*, is there to protect the application itself, and only the application, against potentially harmful untrusted data. The second step, called the *neutralization step*, is there to protect other applications being utilized by ours. It is described in Section 5.

Definition 2 (Data Validation) The data validation *step is the process by which we ensure that the untrusted data is of the form that is expected by the application for proper processing, and will not cause the application to behave in an insecure way.*

This definition is a strong departure from the usual approach, since at the data validation step, we do not attempt to protect against injections of any kind. Instead, we have a more focused and better defined scope, which is solely about the application itself.

It should be noted that with this definition, the data validation step is closely related to classical good software engineering practice. The goal is to ensure that the input data is within the expected types and range, which one would assume is already done by the application anyways. As such, the effort required for that step is minimal, at least if the application was engineered correctly to begin with. It still departs from the expected existing checks in place in at least two ways: first, our data validation should be concerned with *denial of service* attacks (an attempt by an attacker to prevent legitimate users from using the application, see e.g. [7]), and thus may act on inputs that are not invalid but would cause the application to consume too much of some restricted resource (such as CPU, memory or storage). Second, the checks must be "paranoid" and not assume anything about that data coming in, for example not assume that the data has been entered normally via the user interface.

4.2. Where to Validate

As we have seen in Section 3, the temptation of disconnecting the validation from the application logic is strong, with a "natural" tendency to block potentially harmful data as early as possible, typically just as it enters the application. We argue that one should not yield to this temptation, because this does more harm than good: first, the data is either validated without context, which can work only for the simple case, or the context has to be re-coded at the validation point. Second, the data might be incomplete or partial, in which case the validation cannot be very effective. And third, the same data should be validated along every possible path, leading to a duplication of the same validation process along each path and the production of modules that are not safe to re-use as such, since they do not contain the validation step.

The conclusion is clear, and the validation step should only be performed within full context, inside the module that is going to use the data. In this way, the application logic and validation steps are not duplicated, safe reusable modules are produced and the data validation step can be integrated with the existing input validation steps of the modules. In Fig. (**2**), it means that within Application 1, the data shown should be validated as it enters module Mod 3, where it is processed; at that point (and only at that point) data coming from Data App 1 is known and a context-aware validation can be performed.

The apparent drawback is having "unsafe" data traveling around the application for a long time (within modules Mod 1 and Mod 2 in the example of Fig. (**2**)). Security-minded programmers might feel uncomfortable with this, but there is no real problem from the security viewpoint of having unvalidated data simply passed along. The data will still be validated from the viewpoint of Mod 1 and Mod 2, which in

this case may simply mean to check that enough storage is allocated to accommodate the data as it passes through the modules.

4.3. How to Validate

Now that we know what to do when validating untrusted data and where to do it, we still need to know how to do it. For this, we do not suggest anything outside classical techniques, since they seem very appropriate: precise type checking, bounds verifications and any relevant application-level logic should be used to ensure that the data can be safely processed by the module being protected. As previously stated, no assumptions should be made regarding the untrusted data, in particular, it should not be assumed that the data has not been tampered with, unless strong cryptographic techniques have been used to guarantee or verify it. If one is concerned with efficiency and wants to avoid duplicating validation (for data that is used in different objects), again no assumption should be made and application-level checks should be implemented to ensure that validation has indeed occurred. In addition, clear guidelines should be established regarding what steps to take when validation fails (rejection of the data, reformatting to an acceptable form etc.).

5. DATA NEUTRALIZATION

5.1. Goals of the Neutralization Steps

Since the data validation step was solely about protecting the application itself, we still need to protect against all possible command injection attacks. It means that, when using a virtual machine M, we want to remove the possibility for untrusted data to contain an element that is going to be recognized as a keyword of the language L_M by the grammar G_M. This is the purpose of the *neutralization step*, which protects the other applications (the "virtual machines") against attacks carried through our application.

Definition 3 (Data Neutralization) *The data neutralization step is the process by which we ensure that the untrusted data can be safely passed along to the various virtual machines M used by our application and is free of any L_M-injection attacks.*

This clear separation between validation and neutralization allows to properly cover the various issues that must be addressed with untrusted data. It also helps clarifying an apparently widespread misconception suggesting that this particular area of software security boils down to good software engineering practices. In other words, if sound engineering practices are followed, then everything that needs to be done around data validation will be done and security problems will come only from glorified bugs. We have already identified some aspects of the validation step that are not typically covered by the typical software engineering steps; the data neutralization actions are even less likely to be addressed by proper software engineering techniques, since this side of the problem is not about bugs in the application but about protection of other applications [39].

5.2. Where to Neutralize

Much like the data validation step, the data neutralization step should be done only once the context is fully clarified. This is typically only once the data is ready to be sent over to the target virtual machine, which we call the *injection point*. Injection points do not necessarily imply a direct communication with the target virtual machine. It is the point at which the data that will eventually reach this virtual machine leaves our application.

Thus, as opposed to current approaches which tend to handle the threat as early as possible, in our framework the neutralization is a "late" step, we neutralize the data as close to the injection point as possible. In the example of Fig. (**2**), in App 1 the neutralization step, which neutralizes the user input from command injections in module Mod 6, is done at the output of module Mod 3, where maximum contextual knowledge as been accumulated (for App 1) and the most informed decision can be taken. Note that if we have control over App 2 as well, then a similar (and potentially better) neutralization would be performed at the output of module Mod 5.

There is an exception to this situation, which is when some previous computation makes it impossible to distinguish the untrusted data from the other data at the injection point. This happens typically because the untrusted data is mixed with trusted data at some point in the computation. When this is the case, we simply would not be able to neutralize at the injection point and we have to apply that step earlier (before the information is lost). It is however best, if at all possible, to modify the code to allow neutralization to happen close the injection point instead.

5.3. How to Neutralize

Again, how to best neutralize, and what to do with faulty data, is context dependent and should be adapted to the application being secured. In practical terms, it seems that the usage of simple neutralization rules making use of usual white lists mitigated with black lists is appropriate: allow only known good, and if necessary, filter out, escape or encode known bad (see e.g. [35, 40]). Clearly, the neutralization step is greatly simplified by the usage of our framework, which helps understand precisely what this step is all about.

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 [19] a method consisting of tagging what we call the variable terminal symbols of G_M and then simulate a parser for G_M to see if it derives the existence of keyword of L_M within tags. Other approaches are based on static analysis of the code and automatic inference of monitoring tools [18, 37, 38].

Ideally, the need for neutralization can be mostly avoided by interacting directly with the virtual machine to safely construct the statements by passing the variables as such and avoid any possible confusion between code and data (such as using prepared statements when building an SQL query). Unfortunately, this interactive solution is not always available with the virtual machine at hand, and in some cases will perhaps not be available any time soon (e.g. to construct HTML pages or XSL Transformations [15]). And even when such an interactive construction is possible, it is possible that

using it is not an option, for example to avoid dependency on a particular implementation of the virtual machine.

When neutralizing data, one important thing is to identify all the target virtual machines for the data and to neutralize for each of them. In the following example, a JavaScript code is built partially from the untrusted data `Evil`:

```
<HTML>
...
<BODY>
...
<SCRIPT language="JavaScript">
var unsafe = '<%Evil%>';
...
```

This code is very insecure, since the value of `Evil` is used as is inside the JavaScript code. In order to secure that, we should clearly neutralize `Evil` against JavaScript injections, since JavaScript is the target virtual machine. However, this would not be enough. Indeed, the JavaScript code is inside an HTML page, and thus we should also neutralize against HTML injections, since the resulting page will be send to an HTML virtual machine (the user's browser). If we fail to do that, then a value such as `</SCRIPT>...` for `Evil` will give the attacker control of the HTML engine. We should thus neutralize against both JavaScript injections and HTML injections (and in that order) to properly neutralize that code. And if the data was to be stored in an SQL database before, then `Evil` should also be neutralized against SQL injections.

6. DATA FOOTPRINT

If the framework described above has been correctly implemented, then the result should be an application free of untrusted data attack problems, protected against attacks to the application logic, as well as against attacks sent to other applications via the protected application.

However, that does not mean that we are fully safe, even after having done the best possible job at data validation and data neutralization. There are at least three reasons for that:

- The virtual machines will evolve. Our application has been designed with a set of existing possible implementation for the virtual machines (e.g. a set of existing SQL-based databases or a set of existing web browsers). If our application is successful, then it will outlive these existing applications, and will end up being used in conjunction with virtual machines that did not exist at the time of development (e.g. a new SQL-base database will be popularized, a new HMTL browser will be introduced, new version of existing ones will be used etc.). Each one of these new virtual machine can implement a slightly modified grammar, and thus be vulnerable to new command injections that were not present at the time of development. Clearly, there is not much we can do against such future unknown threats, and clearly once these future threats are actually available, our application is not secure anymore.
- The application might be used in an unforeseen environment. The data that is stored by our application might be redirected to an unexpected virtual machine, creating unforeseen injections opportunities. For example, the data stored in a database might be later used as input to another application. Or, the log file created on-the-fly by our application can be redirected by some of our users to a log formatting tools that creates HTML reports out of it, creating unexpected HTML injection opportunities through logged information. Again, there is not much that can be done against that ahead of time, since we cannot possibly neutralize against everything "just in case", but once the problem occurs our application is a threat despite our efforts.
- We could also simply have made some mistakes. If the application is large enough and complex enough, and is interacting with enough virtual machines, then we are looking at a complex task. Even if our framework simplifies the problem, it remains that the task is difficult, and thus error prone. It is not reasonable to expect creating a vulnerability free application just using a good framework, just as it is not reasonable to expect creating a bug free application just using good software engineering methods.

Against these problems, at first we seem to be powerless, since we cannot anticipate them precisely enough to act. Our suggestion is then to be upfront about how the application interacts with its surroundings, identify and document what it is writing and where, so that after having made the best possible efforts to produce a safe application, we can also give to the application's user the necessary information to decide whether a particular evolution of the system (new version, new virtual machine, new usage) is safe or not. This is what we call the application *data footprint* map. A data footprint map will thus identify where the application stores data (e.g. this database, this table, this field) and provide in each case the possible form of the data that can be stored, as a regular expression. This is the *stored data* footprint. We need to also list the *volatile data* footprint, capturing direct interactions between the application and the virtual machine, typically command invocations. In the example of Fig. (**2**), it is shown that App 1 has a direct interaction (in this picture with App 2, although it might be that in a different context, App 1 interacts with some other application App' 2). Thus, the data footprint map of App 1 will record this volatile interaction, describing with a regular expression all the possible values that can sent there by App 1. App 1 is also reading from a database Data App 1. If what is read was written by App 1, then this stored data footprint will also be included in the map, along with the possible values.

Definition 4 (Data Footprint) *The application data footprint is an exhaustive "map" of the application's interaction with the outside, identifying precisely where and under what possible form the application stores data, as well as how it interacts with virtual machines. The possible form of the stored data and virtual machines interaction is typically specified using regular expressions.*

Armed with such as data footprint map, we are now able to mitigate the three problems identified above: looking at the bare information might help identify overlooked injection possibilities with existing virtual machine. Moreover, when the environment in which the application evolves, either because new virtual machines or new usage is introduced, we are now able to look at the data footprint of the application and decide in an informed way whether or not the new situation constitute a security risk. Without the map, we would have had no way of deciding, and we would have had to embark into a costly evaluation for every possible evolution. With the footprint map, we can assert that the new situation is not introducing new security risks, or on the contrary we can identify new vulnerabilities and act accordingly.

7. CONCLUSION

In this paper, we have presented a framework that can be used to handle untrusted input in a secure manner. Unlike most of the current techniques, our approach can be used with large, complex applications and is compatible with current software engineering best practices.

We have identified a three step process, where each step has a precisely defined goal, and indications are provided for where to implement these steps in the application. The first step, data validation, protects the application itself against malicious input. The second step, data neutralization, protects the other applications used by the application being secured. The third step, the data footprint, addresses the question of the evolution of the environment in which the application will evolve, and the fact the vulnerabilities will be created in the future due to these evolutions. This is, as far as we know, the first precise, realistic and complete solution to the problem. Experiments done with earlier version of this framework on real life, large web applications were reported in [14].

The next step will be to provide tools to help creating the data footprint, or even to automate the production of such a map. This tool will help trace backward from the injection point back to the untrusted data insertion point, and to help automatically inferring the possible syntactic forms of these interactions.

ACKNOWLEDGMENTS

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

REFERENCES

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

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

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

[4] H. van Vliet, *Software Engineering: Principles and Practice*, 2nd ed. Wiley, 2000, ISBN 0-4719-7508-7.

[5] I. Sommerville, *Software Engineering*, 7th ed. Addison Wesley, 2004, ISBN 0-3212-1026-3.

[6] 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.

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

[8] J. P. Anderson, "Computer security technology planning study", *accessible from http://csrc.nist.gov/publications/history/ande72.pdf*, p. 61, October 1972.

[9] E. Levy (a.k.a. Aleph One), "Smashing the stack for fun and profit", *Phrack magazine*, vol. 7, no. 49, November 1996.

[10] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks", in *Proc. 7th USENIX Security Conference*, San Antonio, Texas, jan 1998, pp. 63-78.

[11] E. G. Barrantes, D. H. Ackley, T. S. Palmer, D. Stefanovic, and D. D. Zovi, "Randomized instruction set emulation to disrupt binary code injection attacks", in *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*. New York, NY, USA: ACM, 2003, pp. 281-289.

[12] J. Xu, Z. Kalbarczyk, and R. Iyer, "Transparent runtime randomization for security", *Reliable Distributed Systems, 2003. Proceedings. 22nd International Symposium on*, pp. 260-269, Oct. 2003.

[13] R. Riley, X. Jiang, and D. Xu, "An architectural approach to preventing code injection attacks", in *DSN '07: Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 30-40.

[14] G. -V. Jourdan, "Securing large applications against command injections", *41st Annual IEEE International Carnahan Conference on Security Technology*, pp. 69-78, Oct. 2007.

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

[16] C. Anley, "Advanced sql injection in sql server applications", http://www.ngssoftware.com/papers/advanced sql injection.pdf, January 2002.

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

[18] 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*. New York, NY, USA: ACM Press, 2005, pp. 174-183.

[19] 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*. New York, NY, USA: ACM Press, 2006, pp. 372-382.

[20] T. A. Sudkamp, *Languages and machines: an introduction to the theory of computer science*, 3rd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006, iSBN 0-32132221-5.

[21] J. Boyer, "Canonical xml", *http://www.w3.org/TR/xml-c14n*, 2001.

[22] The Open Web Application Security Project (OWASP), "Canonicalization, locale and unicode", *http://www.owasp.org/index.php/Canonicalization%2C loc ale and Unicode*, Novembre 2006.

[23] C. Linhart, A. Klein, R. Heled, and S. Orrin, "Http request smuggling", *http://www.watchfire.com/resources/HTTP-Request-Smuggling.pdf*, 2005.

[24] M. Howard, and D. LeBlanc, *Writing Secure Code*, 2nd ed. Microsoft Press, 2002, ISBN 0-7356-1722-8.

[25] The Open Web Application Security Project (OWASP), "Trust boundary violation", *http://www.owasp.org/index.php/Trust Boundary Violation*, July 2006.

[26] M. Curphey, and D. Raphael, "Software security code review: Getting it right before you release", *Software Magazine*, http://www.softwaremag.com/L.cfm?Doc=2005-04/2005-04coderev.

[27] Apache Software Foundation, "Apache struts 2 validation", *http://struts.apache.org/2.x/docs/validation.html*, September 2006.

[28] Apache Software Foundation, "The apache myfaces project", *http://myfaces.apache.org/*, 2006.

[29] Sun Microsystems, Inc., "Java platform, enterprise edition", *http://java.sun.com/javaee*, 2006.

[30] A. Moore, "User input validation in asp.net", *http://msdn2.microsoft.com/en-us/library/ms972961.aspx*, March 2002.

[31] Microsoft Corporation, "Introduction to Validating User Input in Web Forms", http://msdn.microsoft.com/library/en-us/vbcon/html/vbconintroductiontovalidatinguserinput inwebforms.asp, 2006.

[32] Microsoft Corporation, "NET framework", *www.microsoft.com/net/*, March 2002.

[33] I. Ristik, "Modsecurity", *http://www.modsecurity.org*, 2006.

[34] 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.

[35] 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.

[36] D. Scott, and R. Sharp, "Specifying and enforcing application-level web security policies", *IEEE Transactions on Knowledge and Data Engineering*, vol. 15, no. 4, pp. 771-783, 2003.

[37] 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*. New York, NY, USA: ACM Press, 2004, pp. 40-52.

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

[39] 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.

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

Received: October 30, 2008 Revised: December 12, 2008 Accepted: December 18, 2008