Software Security Vulnerabilities Seen As Feature Interactions

Guy-Vincent JOURDAN
School of Information Technology and Engineering
University of Ottawa,
Ottawa, Ontario, Canada
E-mail: gvj@site.uottawa.ca

Abstract. The security of software applications is an important domain, and one that mixes formalisms (e.g. when dealing with cryptography and security protocols) with very ad hoc, low level practical solutions. In this paper, we look at a subset of the "security" field: the production of secure, general purpose software from a software engineering viewpoint. We call this simply "software security". We show that, when we analyze this particular subset of the field, many if not most problems turn out to be instances of feature interactions problems. We illustrate our claim by looking at three of the top ten most common vulnerabilities in Web application as published by OWASP (the three that are in fact software security issues) and show that in each instance, we can express the problem as a feature interactions problem. We also reach the same conclusion with one of the latest generalized software security vulnerability, "ClickJacking".

Keywords. Software security, interactions, injections, software vulnerabilities

Introduction

With raising awareness among the general public about the consequences of software security vulnerabilities, industry leaders are heavily investing into securing their existing products, and producing more secure software going forward. 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. In parallel, many groups of professionals have been setup to discuss and attempt to address problems linked to software security. Meanwhile, the academic world has been a little slow to catch up and has mainly focused on security software[1]. This lack of academic leadership is quite clear when browsing classical software engineering books and noting that they are essentially silent on the question of producing secure software (see for example [5,6] among many others).

The mostly non academic interest groups working on software security have spent time trying to classify the kind of vulnerabilities that are most commonly found in today's

[1] People like Gary McGraw have been long insisting that software security and security software are two different problems, the latter being a necessary but far from sufficient condition to the former, see e.g. [4].

applications, and to find ways to prevent them. One such group, focusing on Web applications, is the Open Web Application Security Project (OWASP, http://www.owasp.org). This group has been maintaining a list of what they perceive as the top ten most common vulnerabilities in Web applications. This list has been updated over the years, the latest installment to date has been produced in 2007 [7]. Another group is the MITRE corporation (http://www.mitre.org), who maintains a list of vulnerabilities trends [8], and recently co-organized with SANS (http://www.sans.org) the project "2009 CWE/SANS Top 25 Most Dangerous Programming Errors" [9]. One common issue with these attempts at classifying vulnerabilities is that problems of very different nature are combined, for example weak cryptographic algorithms, flawed security protocols, unsecure usage of cryptographic algorithms and/or security protocol, software security vulnerabilities, unsecure application configuration, unsecure application architecture etc. Each of these problems leads to security breaches, but the skills required to understand and resolve the problem are vastly different. In the context of this article, we focus on software security vulnerabilities (as opposed to more classical security domain such as cryptographic systems, secure protocols, intrusion detections etc.), and to a lesser extent on application architecture. We show that when looking at this aspect of the security of a software system, what are in fact often faced with feature interactions problems. We advocate that the academic research that as been performed in the domain of feature interactions should be applied to the much needing domain of software security, to the benefit of both communities.

In Section 1, we illustrate our point with three simple, classical software vulnerabilities that we show can be seen as feature interactions problems. Then, in Sections 2 and 3, we do a more systematic evaluation by looking at the "top ten" listing of OWASP [7]. Of these ten vulnerability categories in Web applications, arguably only three are really software security issues, while the other seven are more along the lines of improper configurations, improper usage of cryptographic systems or unsecure architecture. We show that all three software security categories can in fact be explained in terms of feature interactions problems. We also show in Section 4 that the latest uncovered serious generic vulnerability, "ClickJacking", is also a case of feature interactions! In conclusion, we discuss our findings in Section 5, where we insist on our main point: the software security community is in dire need of academic formalism and generic solutions, and is faced with an entire category of problems that are an instance of feature interactions. The feature interaction community should apply its results to that domain.

1. Three Classical Vulnerabilities Explained As Feature Interactions

In this Section, we look at three simple and classical software vulnerabilities that are not part of the topmost vulnerabilities any more but can be used to illustrate our point: in each case, one way of understanding the problem is the interaction of features which, in isolation, are fine (and sometimes are even meant to enhance the security of the system) but when combined do lead to serious security breaches.

1.1. Race Conditions When Accessing Temporary Files

Instances of this problem have plagued Unix systems throughout the 90s. The first feature involved is actually a security feature! On Unix systems, much like on most modern

operating systems, access to files is controlled. In order to read, execute, write or delete a file, a user must have sufficient access rights. Thanks to this feature, among other things it is not possible for a simple user to create havoc in the entire system by modifying critical system files: one must be "root" to modify files such as the password file. Another feature that Unix offers is symbolic links to file, which allows the creation of what looks like a file in a directory, but is actually a reference to another file, possibly in another directory. Accessing the former is in fact accessing the latter. Yet another common feature of Unix (at the time) was a shared temporary directory. This was a mean to avoid having programs writing temporary files (and possibly not removing them) all over the place. Instead, the "/tmp" directory was used.

The vulnerability that these three features combined opened was the following: a program running with high privileges (typically root) would create a temporary file in the "/tmp" directory by first checking if the file existed and, if not, creating it. The program would then use the file to write some data, and would eventually delete it. An attacker would use a low privilege account to automatically create files in the "/tmp" directory having the (often easily predictable) name used by the root program for its own temporary files. The goal was to create the file after the root program had checked for its existence, but before it created it itself (hence the "race condition"). This was quite easily automated. The file created by the attacker was not an actual file, but rather a symbolic link to a critical system file. Once the attack succeeded, the root program would proceed by adding its own data to the critical file, and more often than not would end deleting the file entirely, and thus make the system unusable.

Note that this problem is rarely seen nowadays. Temporary directories are not system-wide any more, most administrators avoid running programs as root unless required, and more secure ways of creating files are advocated to programmers. Still, what we have here is a feature interaction problem.

1.2. Administrator Trojan

This attack used to be a favorite of computer science students in the 80s and 90s. The first feature involved is the access control system already mentioned, which prevents low privilege accounts to modify higher privilege files. The second feature is the simple notion of an environment variable, "path": when one wants to execute a program, for example listing the content of the current directory using the "ls" command, it would be particularly cumbersome to have to indicate to the shell where to find the "ls" executable file on the system. Instead, since most of these programs are actually within few directories, an environment variable called "path" is created, containing a list of directories that the shell will explore to find a program. Similarly, when you compile a program, it is handy to be able to invoke the program by simply typing its name from the command line, assuming that you are in the directory that contains the program. So, the "path" contains the relative directory ".", meaning that the shell should also explore the current directory when a program is invoked. Finally, if you type the name of a program that is in the current directory, you typically expect this program to be executed, not another program with the same name in another directory of the "path". So the "." is first in the "path".

This naturally leads to the following vulnerability: create a program, named "ls", in your low privilege home directory. Have this program do something you want (such as

adding your account to the administrator group), then delete itself and display the result of the "real" ls command on the current directory. Obviously, you cannot run your own program, lacking the necessarily privileges. Now complain to the administrator about something in your home directory, such as a file with a space in its name that you don't know how to delete. The administrator goes to your home directory, lists its content by typing "ls", which executes your local program because of the dot at the beginning of the administrator's path. You are added to the administrator group, your attack is deleted, and the administrator sees nothing wrong with the result of the ls command.

Again, this problem should not happen any more. Administrators have learned to not use their "root" account when doing mundane tasks (such as servicing students), shell prompts have yield place to graphical explorers anyways, and well administrated system have dropped the dot from the path.

1.3. Second Order Injections

Unlike the previous examples, this attack is still very relevant today. It is in fact an instance of the category of vulnerabilities discussed in Section 3. Assume for example that a software provides a logging feature, which logs in a file all entries it receives from users, before processing these entries. Most servers will have a variation of such a feature. In particular, any web server, such as Apache, will log any URL request it receives before processing them. These log file are typically used by the system's administrator to monitor the usage of the software. In the case of a web server, the logs are usually used to analyses the behavior of the users on the web site, tracking the popular page, finding the most requested URLs and so on. However, a marginally popular web site will receive thousands of requests per day, making it very difficult for anyone to peruse the raw log file and make any sense out of it.

To avoid this problem, web site owners use software that create reports out of the log files, and produce easy to read, interactive graphics that can be used to analyze the activity on the web site. These software can typically create their reports as HTML file, to be in turn displayed by the very web server it monitors. And, finally, web page are seen with Web browser, that usually have a scripting feature that enables scripting code to be embedded in an HTML page and run in the client's browser.

Putting all of this together, what happens when an attacker sends an URL request which is actually not a valid URL but some malicious scripting code? The web server will log the request, as it is expected to do. It will log it verbatim, since the site administrator wants to know what really happened, not some "cleaned-up version" of the truth. The reporting software, working from the log, will incorporate the script as part of the report. And finally, the administrator of the site will review the report in a web browser, thereby executing the script that was planted by the attacker.

This kind of attacks has been called "second order injections" [10]. Note that many variations are possible, for example sending an URL request containing a carriage return symbol, in order to get one request displayed as two separated ones when looking at the resulting log files. Again, this is a case of feature interactions, where the vulnerability is the result of individual features that just don't coexist gracefully.

2. Cross Site Request Forgery (OWASP TOP TEN: A5)

We now focus on generic categories of security flaws, due to unsecure programming and unsecure software architecture for non security software. As already mentioned, groups like OWASP have been monitoring the most commonly found types of security errors in software on a yearly basis (web application, is this case). We first look at "Cross Site Request Forgeries", which is number five in OWASP's top ten for 2007. Cross Site Request Forgery (CSRF), sometimes known as session ridding, XRSF, "One Click Attacks" or even "hostile linking", as been called "the sleeping giant" of web vulnerabilities by Jeremiah Grossman [11]. It was listed as one of "the most exploited types of vulnerabilities" for web application in 2007 by SANS [12]. See e.g. [13] for a detailed technical description of the attack and possible lines of defense.

The first feature involved in this attack is a simple browser "cookie". Cookies are a simple way for web server to store small amount of data on the client side. The data being stored is sent back to the server by a cookie compliant browser along with each request to the originator site. Cookies have a variety of purpose, but the most important one in our case is that it is the most common work around for the state-less nature of the HTTP protocol. In order to be able to track a user throughout a session, in the absence of the notion of state, web application routinely issue "session IDs" that are unique to the client and stored as a cookie. Thus, when a new request is received, the server gets the session ID from the cookies being sent by the client and retrieves the state information for that client in some local database. As far as the web application is concerned, any request received with a particular client's session ID is coming from the client, and this is in fact the only way to reconcile a received request to a client session. This particular session management mechanism is the second feature at play here. Another important feature is the heterogeneity of an HTML page. One page can be made of a combination of various sources, coming from different locations, all mashed up into one particular page. And the way to gather the data from these sources is uniform: http requests (mostly GET) are used to fetch the main page, the frames, the images etc. So, from the receiving end (the web server) a request for a page and a request for an image inside a page are not distinguishable.

A CSRF attack puts all of these features "at work" together in the following way: a victim user is logged on to a web application, say a banking application. Thus, a cookie has been set by the application, containing the victim's session ID. Every request to the application sent by the victim is sent along with the cookie, and processed by the server application as a valid request from the victim. The attacker crafts an HTML page in which, among other unrelated data, a link back to the bank application is hidden. For example, this link is given in lieu of an image, so when the page is viewed in a browser, the browser rendering engine automatically follows the link in the background in order to fetch the image. In other words, no intervention from the user is required, no scripting is involved. Now, if the attacker manages to get the victim to just look at the malicious page, the victim's browser will automatically follow the link to the victim's bank application, and will send the victim's cookie containing the session ID along with the request. If for example the link to the bank application leads to a request of money transfer to another account, a common functionality for such applications, the request will be sent from the victim's browser, along with the victim's session ID, and will be processed as a legitimate

request by the bank application (in fact, the bank application just cannot tell apart this attack from a legitimate money transfer request from the victim's part)[2].

It should be noted that CSRF attacks are still quite difficult to fend off. It usually involves adding unique user tokens to each URLs of the application, and raise the bar high enough so that a CSRF attack require a successful XSS attack as well (see Section 3.1), in which case the CSRF attack doesn't provide much more to the attacker anyway.

3. Injections vulnerabilities

Another very large class of software vulnerability is "injections". A rapid overview of the software vulnerabilities published within the last few of years shows that a number of them belong to the category of "command injections". In OWASP's 2007 top ten listing, two of the top ten vulnerabilities belong to the injection category (down from three in the 2004 listing, because several problems have been merged together into a large "Injections Flaws" category).

We will first review the two categories as defined by OWASP, and we will then provide a more formal framework for describing injection vulnerabilities. In the formal framework, we will again link this type of flaw to feature interactions problems.

3.1. Cross Site Scripting (OWASP TOP TEN: A1)

According to SANS, "*Cross site scripting, better known as XSS, is the most pernicious and easily found web application security issue. XSS allows attackers to deface web sites, insert hostile content, conduct phishing attacks,* [and] *take over the user's browser using JavaScript malware*". XSS vulnerabilities are routinely reported, and most of the major web sites have had instances of publicly disclosed XSS problems at one time or another.

A Cross Site Scripting attack requires a scripting enabled web browser (most of the major browsers are by default, and disabling scripting is in fact a major inconvenience, since a large amount of legitimate web site and web applications do require scripting). The script language of choice is JavaScript, although any other browser-supported scripting language works just as well for the attack. The goal is for the attacker to embed some script into a web page that will be viewed by the victim. There are a few different ways to achieve this, but one fairly common one is to use a web application that will accept user input and display that input back to other users. This widespread feature can be abused if the inputs are not thoroughly validated and sanitized, which is often not the case (most web applications nowadays do provide some level of data validation and neutralization, but not enough to completely prevent XSS attacks).

[2] One could argue that it seems pretty difficult to get the victim to look at a page while having an active session with the attacked application. In fact, it is often possible to embed the malicious page inside the application itself, thus almost guarantying that the victims will have an active session. This is because a large number of web applications implement some sort of user feedback/comments functionalities, which are integrated in the application and displayed back to other users. Thus using one such feature is a "natural" place to plant the attack with a very high success rate.

3.2. Injections Flaws (OWASP TOP TEN: A2)

OWASP propose another category called "Injection flaws" in which many other types of injections are grouped. This category is described as follows: "*injection occurs when user-supplied data is sent to an interpreter as part of a command or query*". The two most common injections flaws (in Web application, and beside Cross Site Scripting), are SQL injections and HTML injections.

An SQL command injection vulnerability exists whenever an application uses an SQL database and constructs unfiltered (or improperly filtered) SQL commands based on untrusted input. As an example, consider a web application that logs in users by querying 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'') + ''';'';
```

Malicious users can 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[3]

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

If this command is run, it will always return the record having `administrator` in the field `UserId`, regardless of the associated password.

HTML injections, much like XSS attacks, target HTML browsers, but do not require any scripting capacity. As a simple example, assume that an application displays the nicknames of the users currently on. If the application has stored a user nickname into the `NickName` variable and constructs its HTML page using the following server side ASP code:

```
<B><%NickName%></B> is currently logged in!
```

then a malicious can use a nickname that includes HTML tags. These tags will be embedded into the page sent to other users, and will be interpreted by the victim's browser as HTML tags of the page. For example, the attacker can use those tags to insert an entire fake login form similar to that used by the application and steal valid userids and passwords.

3.3. A Generic Description of Injection Attacks

Cross site scripting, SQL injections and HTML injections are all variation of a large category of software vulnerability, which is not at all limited to Web applications. Injections

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

attacks are in general the ability for an attacker to use an application as a vector to inject commands into another application, directly or indirectly. In the examples seen to far, the "vector" application is the web application, while the target application is the user's HTML browser for XSS and HTML-injections, and the SQL database in the case of the SQL-injection. The fact that the target of the attack is not the application used to launch the attack makes it particularly difficult to deal with injection attacks in general. In the tradition of feature interactions problem, the challenge is all in the fact that the application's feature on its own is fine, but is not once combined with another feature of another application.

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 [14], LDAP servers, word-processing and spreadsheet embedded application macros, interpreted programming languages, and shell commands. More importantly, more applications will be introduced in the future that will lead to more injection vulnerabilities down the road. In the worst case scenario, an application deployed today will tomorrow be coupled with a new application, leading then to an injection vulnerability that doesn't even exist at the time of the application's deployment. Securing applications under these circumstances is impossible unless a formal approach is followed, and technology-independent solutions are provided.

In [15], we proposed a definition of injection attacks using formal languages and virtual machines, and drew from this definition a general framework to protect against this type of vulnerabilities. But this problem can also be explained in a generic manner using feature interactions. We need two applications, application A which is the "vector" through which the attack will be carried out, and application B which is in fact the target of the attack.

The feature required on application A is anything allowing the users to inject data that will be eventually transmitted to application B, under a particular form. The transmission does not need to be verbatim, nor direct. Application A can (and usually will) modify the data, process it in various ways, and the result might not reach B directly. It may require several more applications along the way. In fact, it may well be the case the A doesn't particularly "know" that B will eventually receive the data[4].

Application B must process inputs and execute some actions based on these inputs. The key feature for B is that it should have the possibility to receive what amount to *instructions* as input. Note that most applications do not do that: inputs are rather parameters that are processed by existing instructions. Let us illustrate the difference on an example, an SQL database. Typically, it can receive two kinds of inputs: direct SQL queries, which are then what we call "instructions". What it does depends on the received query, which is in fact SQL code. Or, another way to use an SQL database is to use stored procedures, in which case the SQL code is stored inside the application, and the input is simply the parameter to use for the stored queries. In this case, a user cannot instruct the SQL database to run a particular code, it can only provide the parameters for existing code. The ability for B to accept code directly and interpret it is the feature that we need.

[4] As in the example of Section 1.3, where the web server log file ended up formatted as HTML is the administrator browser: A is the Web server, with the logging feature, B is the HTML browser, and the Web server did not intentionally create the log to be displayed in the user's browser.

Once we have both applications with the required features, we have a potential for an injection vulnerability. We do not necessarily have a vulnerability, though, it all depends on what *A* does to the user inputs and the steps it takes to ensure that the part coming from the user is under control when sent to *B* (see [15] for a detailed discussion), but at the core of it, we have two independent features whose interaction create the potential security vulnerability.

4. "Clickjacking"

The last software security vulnerability we want to survey in this paper is the newly discussed "ClickJacking" vulnerability. This vulnerability as widespread consequences, including in some instances the possibility to remotely turn on and monitor a victim's computer camera and microphone. In general, it is about the possibility to subvert victim's mouse clicks and redirect these inputs toward some other location of the attacker's choice [16]. More importantly, at the time of writing, no real protection exists. Microsoft as announced that the new version of Internet Explorer will have built in protection against ClickJacking [17], but the effectiveness of this solution is still being debated.

Looking at ClickJacking from a high level, we see that it is again a question of feature interactions. Several features are at play: the first one is the framing feature of HTML: it is possible for a page to create several frames, and load into these frames other pages, possibly coming from different locations. That in itself creates a large security problem because of another feature, scripting, and the ability to manipulate and interact with displayed documents using script. This particular feature interactions problem was foreseen, and restrictions are enforced by browsers to avoid that a script running in one frame access the document being displayed in a different frame if this document has a different origin. ClickJacking is actually one of the ways to get around this restriction.

The second feature, although strictly speaking not necessary for ClickJacking, greatly increases its impact: it is a feature that most browsers offer, to "remember" a site's user ID and password. When this feature is used, the browser will automatically pre-fill the login information with the correct values. Many people do use this feature.

The last feature is HTML IFrames, and more precisely the ability to position frames arbitrarily on a browser, including one on top of the other, as well as controlling the transparency of the frames (thus deciding what is seen and what is hidden).

In isolation, these are useful features, and seemingly armless. Combined, they allow ClickJacking. A simple way to "clickJack" a site is to load the target attacked application in one frame, and overlay on top of the frame other frames with content that is presented to the victim. The key idea is to leave uncovered some part of the frame containing the target application, but give the ensemble a consistent look. Thus, the part seen from the target application seems to belong to the frames displayed on top. For example, assume that the target application is the banking application of the victim. On the back frame, load the login page, and but leave uncovered the button that sends the login information. Built the top frameset around this button, and somehow integrate the button to the top page in a realistic way. If the victim has the pre-filled option enabled, the user ID and password for the banking application will be automatically filled by the browser, while the victim will submit the data by clicking on the button (not realizing that the button is actually connected to another frame). Continue doing this page after page, "walking" the

victim through the banking application while showing something else on the top layer, until you have achieved your goal as an attacker.

Obviously, the schema is very difficult to fight against at the application level, without some kind of collaboration with the browser. At this point, the only partial solution seems to be "frame busting", basically adding scripting code to the application to prevent it from running inside a frame.

5. Conclusion

We have reviewed several examples of class of software security vulnerabilities, and in all of these cases, we have shown that the problem was actually an instance of feature interactions. The software security community is struggling with a lot of issues, and does not seem to be in a position to take a step back and provide formal frameworks and generic solutions to the issues. For the time being, the problems are dealt with one at the time, on a case by case basis. A telling example of the situation is the question of injections: for years, different types of injections have been reported, first system command executions, then SQL injections, then Cross Site Scripting issues, HTML injection, etc. But somehow, the connection between these problems was not really done by the community, and no effort was initially made to analyze the root cause of the problem, and anticipate its occurrence with other technologies. Thus, it was "discovered" half a dozen times, given different names, and provided ad-hoc solutions every time.

The security community does not seem to realize the importance of feature interactions when it comes to software security. A lot more attention is given to securing the applications themselves, even though, as we have shown, the actual vulnerabilities are in majority the result of interactions. It is obviously necessary to secure the application, but that will not provide software security! This divide can easily be seen on input data validation. In the traditional security circle, input validation is done when the data is received, and aims at protecting the application itself. Other layers are added as various injections vulnerabilities are uncovered, but this is not done in a systematic and formal way. In [18,15], we argue that what is usually seen as "input validation" should actually be split into a "validation" step, as the data gets in, and a "neutralization" step, as it gets out. The validation protects the application, while the neutralization protects other application from receiving our input. The neutralization process is far more difficult than the validation process. In the interaction lies the real difficulty.

The feature interactions problem has been studied by academic researched for almost two decades, and a wealth of results has been gathered. Different solutions, different formalisms have been evaluated, with different results. The academic community involved in this research should look at the issues of software security in the light of the analysis provided here, and apply the theories and results accordingly. This should provide a "jump start" to the much needed formalization of the very important matter of software security, and will provide the feature interactions community with a large set of very relevant applications.

Acknowledgements

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

[5] I. Sommerville, *Software Engineering*, 8th ed. Addison Wesley, 2007, ISBN 0-321-31379-8.

[6] H. van Vliet, *Software Engineering: Principles and Practice*, 3rd ed. Wiley, 2008, ISBN 978-0-470-03146-9.

[7] The Open Web Application Security Project (OWASP), "The ten most critical web application security vulnerabilities, 2007 update," *http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project*, 2007.

[8] The MITRE Corporation, "Vulnerability type distributions in cve," *http://cwe.mitre.org/documents/vuln-trends/index.html*, May 2007.

[9] The MITRE Corporation and SANS, "2009 CWE/SANS top 25 most dangerous programming errors," *http://cwe.mitre.org/top25/*, January 2009.

[10] G. Ollmann, "Second-order code injection, advanced code injection techniques and testing procedures," *http://www.ngssoftware.com/papers/SecondOrderCodeInjection.pdf*, November 2004.

[11] J. Grossman, "CSRF, the sleeping giant," *http://jeremiahgrossman.blogspot.com/2006/09/csrf-sleeping-giant.html*, September 2006.

[12] SANS, "SANS top-20 2007 security risks," *http://www.sans.org/top20/2007/*, November 2007.

[13] A. Barth, C. Jackson, and J. C. Mitchell, "Robust defenses for cross-site request forgery," in *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*. New York, NY, USA: ACM, 2008, pp. 75–88.

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

[15] G.-V. Jourdan, "Data validation, data neutralization, data footprint: A framework against injection attacks," *The Open Software Engineering Journal*, vol. 2, pp. 45–54, December 2008.

[16] R. Hansen, "Clickjacking details," *http://ha.ckers.org/blog/20081007/clickjacking-details/*, October 2008.

[17] Microsoft Corporation, "Ie8 security part vii: Clickjacking defenses," *http://blogs.msdn.com/ie/archive/2009/01/27/ie8-security-part-vii-clickjacking-defenses.aspx*, January 2009.

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