Vulnerability Cause Graphs: A Case of Study

Nicolas Chaufette Tommie Haag
Linköpings universitet, Sweden
Email: {nicch907,tomha490}@student.liu.se

Abstract

Vulnerability Cause Graphs is a method for modeling vulnerabilities and their causes in software products. This paper aims at studying and evaluating this method.

Vulnerability Cause Graphs give the developer a visual representation of relationships between vulnerabilities and their causes, providing a better understanding of the vulnerabilities. However, the use of this method might also bring some disadvantages.

In this paper the Vulnerability Cause Graphs are introduced shortly, followed by three case studies where this method has been applied to. Finally, we show some of the advantages and disadvantages found in the method.

1. Introduction

Even though security is an important aspect when designing software, software engineers do not often take those aspects into account when designing the software product. With increasing complexity and distribution of software, the importance of heeding to security is becoming more and more critical.

While the vulnerabilities the developers are facing are often very similar, the lack of knowledge about their causes often lead to reintroduction of the vulnerabilites in new software. Modeling vulnerabilities has been described as a way to organize information about vulnerabilities both to improve the developers understanding and to reuse the analysis. Ardi et al. [1] introduce the Vulnerability Cause Graphs as a method to describe the vulnerabilities, based on a formal graph representation.

With the aid of security modeling, such as those with Vulnerability Cause Graphs (VCGs), security problems in software can be avoided at an early stage in the engineering process. The application of VCGs has been demonstrated on well-known vulnerabilities [2]. The goal is to develop a set of methods and tools, based on the analysis results of VCG, to improve the prevention of vulnerabilities in new or existing software.

The VCG method is developed, but not widely used. This paper aims to conclude what the advantages and disadvantages there are with the VCG method. We will first describe how the method works and illustrate it with three case studies, which treat vulnerabilities under the class of cross-site scripting vulnerabilities. Finally, we analyze the advantages and disadvantages of the method.

2. Background

This chapter will discuss the basics of vulnerabilities, cross-site scripting and VCGs.

2.1 Vulnerabilities

Gollmann defines vulnerabilities as "…*weaknesses of a system that could be accidentally or intentionally exploited to damage assets*" [3]. This means that if vulnerabilities are exploited, it could violate the confidentiality, integrity or availability of the system.

Vulnerabilities are, as mentioned, a weakness of a system. As a designer you would want to mitigate the vulnerabilities of the system. A mitigation of vulnerabilities is accomplished by first analyzing what the causes for the vulnerabilities are, and then designing the software according to secure practices with the aim to eliminate those causes.

The vulnerabilities can be classified in various categories, for example cross-site scripting, buffer overflows, buffer underwrites, bad privilege assignments, insecure default configurations (passwords/permissions) etc.

The classes of vulnerabilities we have chosen to use as case studies in this report are cross-site scripting (XSS) vulnerabilities.

2.2 Cross-Site Scripting Vulnerabilities

Web sites nowadays make extensive use of web scripts like JavaScripts to improve the user experience. However, this also increases the risk of being attacked by a cross-site scripting attack.

A XSS vulnerability allows an attacker to inject malicious code into web pages generated by web services. The *same-origin policy*, which states that scripts loaded from a web page can only access data belonging to the same domain, is circumvented by attackers by injecting the malicious code into the web page of a normally genuine web site. Then, the cookies and other sensitive data stored by this site in the victim's browser can be stolen by the malicious code and sent to any web site controlled by the attacker.

According to Vogt et al. [5], cross-side scripting attacks can be separated in two different methods. Using the first method of injecting malicious code into a victim's browser, called *stored XSS*, the attacker injects malicious code into a web application, such as a database. If the web site does not filter the input data, a script can be stored in a database and later retrieved by the victim's browser.

The second cross-site scripting method, called *reflected XSS* is used when the malicious code is not stored. The code can be for example hidden into a specially-crafted link contained in an email sent to a victim. When the victim clicks on the link, the code is send to the vulnerable web script of a genuine web server, and the malicious code is written into the web page.

Since there are only two types of XSS attacks and the vulnerabilities they use are very likely to be similar, it is important to model cross-site scripting vulnerabilities to improve the developer's understanding and highlight the vulnerabilities similarities.

2.3 Vulnerability Cause Graphs

The purpose of a VCG is to relate causes to a vulnerability in a software product. The causes and the vulnerability are depicted in a directed acyclic graph with four kinds of nodes: simple, compound, conjunction and exit nodes. The VCG method has a stable mathematical foundation which allows transformation of the graph itself.

The process of developing the complete VCG for a specific vulnerability consists of five steps (for each node) [2]:

1. Determine the validity of the node
2. Determine if the node needs to be split
3. Determine if the node needs to be converted to a compound node
4. Find candidates for predecessors in the VCG
5. Organize predecessor candidates in the VCG

All the VCGs that are created are then put into a Vulnerability Analysis Database (VAD). The purpose of the VAD is to contain the relationships between vulnerabilities and causes, thus making it possible to extract information in future projects or upcoming analysis of new vulnerabilities [2]. We will not describe the VCG method in detail here. The reader is encouraged to read the paper "*Modeling Software Vulnerabilities With Vulnerability Cause Graphs"* [2] for a thorough description of the method itself.

3. Evaluation the VCG Method – Case Studies

This chapter will describe three case studies where the VCG method is used. All cases concern cross-site scripting vulnerabilities.

3.1 Case Study 1: CVE-2002-0902

The first case is chosen from the *Common Vulnerabilities and Exposures* (CVE) database and describes a vulnerability in phpBB, which is a well-known open-source bulletin board. This vulnerability is described as follows [4]:

> *Cross-site scripting vulnerability in phpBB 2.0.0 (phpBB2) allows remote attackers to execute JavaScript as other phpBB users by including a http:// and a double-quote (") in the [IMG] tag, which bypasses phpBB's security check, terminates the src parameter of the resulting HTML IMG tag, and injects the script.*

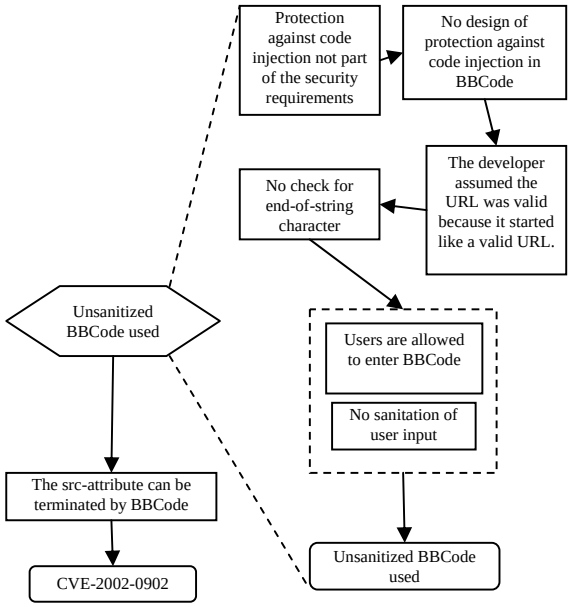

Figure 1. VCG of the CVE-2002-0902 vulnerability

PhpBB is a powerful and customizable open source bulletin board package written in PHP. Boerwinkel [8] found this cross-site scripting vulnerability in phpBB 2.0.0, which allows an attacker to inject arbitrary web scripts into a web page.

BBCode is a special simple language invented for the phpBB users to allow both the users to format their messages without knowledge of HTML tags, and the administrators to disable the use of HTML tags without preventing the users to format their message. To display an image using BBCode, the users just write the URL of

the image between "`[img]`" and "`[/img]`", and phpBB takes care of the HTML tags produced when the message is displayed.

Before the message is stored in the database, the URL of the image should be checked. To do so, the script just checks that the URL begins with "`http://`", and assumes that it is a valid URL. To inject HTML code into the web page, the attacker just adds a quote to the URL of the image, and anything after the quote will be interpreted as HTML code on the user web page. For example the following BBCode

[img]http://n.i/l" onError="javascript:alert(document.cookie)[/img]

will be interpreted as

<IMG SRC="http://n.i/l" onError="javascript:alert(document.cookie)">

informing the user about its own cookie. It is clear that the developer made a wrong assumption about the URL presuming it is valid as long as it begin with "`http://`", or did not think about HTML injection at all.

We modeled CVE-2002-0902 using vulnerability cause graphs. The result is shown in figure 1.

3.2 Case Study 2: CVE-2006-0437

The second case that we have studied concerns another vulnerability found in phpBB. The description of the vulnerability is as follows [4]:

> *Cross-site scripting (XSS) vulnerability in admin_smilies.php in phpBB 2.0.19 allows remote attackers to inject arbitrary web script or HTML via JavaScript events such as "onmouseover" in the (1) smile_url or (2) smile_emotion parameters, which bypasses a check for "<" and ">" characters*[1].

PhpBB gives the user the ability to add, modify or remove its own smilies. The URL of the smiley to add can be given as an argument of the script that manages it. To ensure that the URL does not contain any HTML code, the script just checks for the existence of the '<' and '>' characters by replacing them by their HTML code. That would be enough if the code produced by this stored field was part of an HTML text node, however being a displayed smiley, the URL is the value of the SRC attribute of an IMG tag and the special character to be tested is the quote character.

An attacker can exploit this vulnerability by inserting a quote in the URL of the smiley and start writing attributes of the IMG tag, e.g. DOM events that execute web scripts. Then, any user that just opens the page containing this smiley executes the injected malicious code. Thus, we believe that the developer forgot about the HTML code produced by the script that displays the smiley, or if he did not forget, he did not think about the possibility to write DOM events.

We modeled CVE-2006-0437 using vulnerability cause graphs. The result is shown in figure 2.

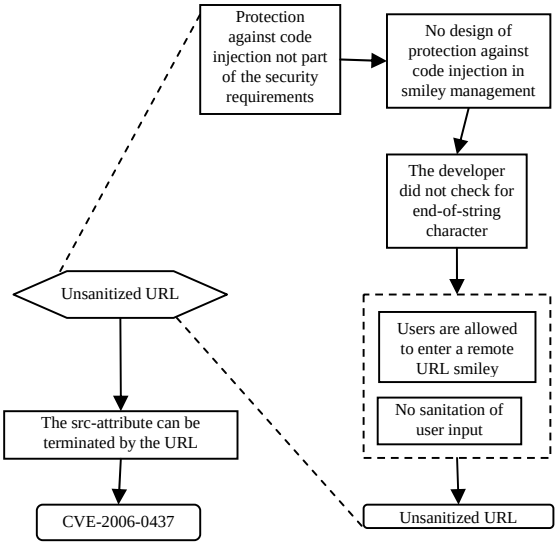

Figure 2. VCG of the CVE-2006-0437 vulnerability

3.3 Case Study 3: CVE-2006-0806

The third and final case studied is a vulnerability found in ADOdb. The description of the vulnerability is as follows [4]:

> *Multiple cross-site scripting (XSS) vulnerabilities in ADOdb 4.71, as used in multiple packages such as phpESP, allow remote attackers to inject arbitrary web script or HTML via (1) the next_page parameter in adodb-pager.inc.php and (2) other unspecified vectors related to PHP_SELF.*

ADOdb is a widely used database abstraction library for PHP. The version 4.71 and below has a vulnerability that allows an attacker to inject arbitrary HTML code including web scripts into a victim's browser.

In addition to unifying the query language and the management of databases, ADOdb provides a number of features to facilitate their use. Among others, ADOdb has the ability to paginate and display the retrieved database records. However, ADOdb 4.71 and below contains several cross-site scripting vulnerabilities that allow an attacker to inject malicious code. We address

[1] *Note*: The description is wrong. There is indeed a check for "<" and ">" – the vulnerability does not allow that check to be bypassed. In fact, the "<" and ">" characters are the only characters checked!

one of these vulnerabilities, concerning the use of an unchecked variable.

The pagination class detects the number of the actual page using a session variable which is overwritten by a GET argument of the script, allowing the user to change page by following a link, or a wise user to access an arbitrary page number by modifying the URL.

Although this is a common practice in a pager system, this variable is here unchecked, and therefore any HTML code contained in this argument will then be displayed where the current page number should be displayed.

We show the VCG of this vulnerability in figure 3. While using GET variables simplify the attack, using POST variables would not only be inconvenient, but it would not stop a wise attacker. A solution could be to simply check that the given page number is a valid integer, which is what the developers did in the following version of ADOdb [7].

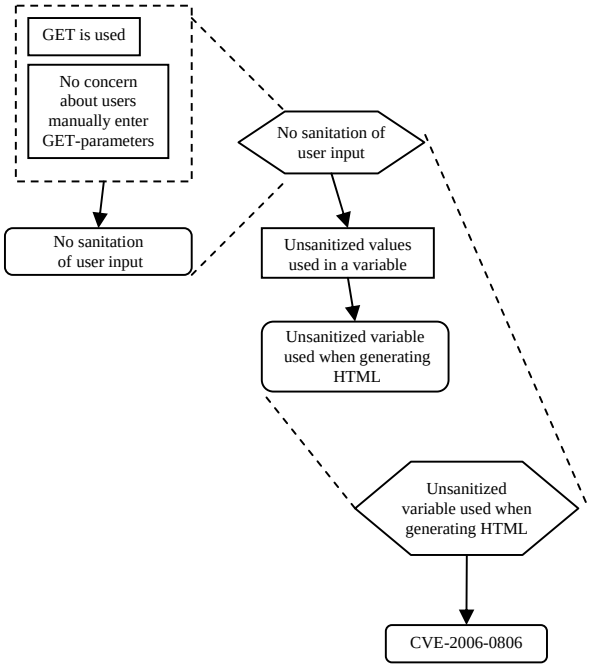

Figure 3. VCG of the CVE-2006-0806 vulnerability

4. Analysis

It appears that the cross-site scripting vulnerabilities are often very similar. For example, despite the fact that four years separate the vulnerabilities CVE-2002-0902 and CVE-2006-0437 and that they concern the same software, the flaw is basically the same. This indicates that vulnerabilities could be avoided be using some kind of vulnerability modeling.

Regarding the utilization of the method, it is a bit hard to use at first. The logic behind the method is uncomplicated and intuitive, but when you start using it you run into some problems addressed in chapter 4.2. One of the main problems encountered is to figure out at which abstraction level the causes should be modeled. Another problem is to determine if a node should be analyzed more or not. After some practice though, the method gets easier to utilize. We noticed that when making the third case study, we had a much better workflow compared to earlier tries.

4.1 Advantages

Some advantages with the method have been found. First of all, the method accomplishes to visualize *connections* between different kind of causes and vulnerabilities in a proper way. The use of the vulnerability analysis database also highlights the way various vulnerabilities are linked to each other. This allows for an easy access to information about what to be careful about when designing a system that must not exhibit a specific type of vulnerability.

The compound nodes in the VCG allows for visualization of several layers of abstraction. Depending on the level of details you want to indulge yourself in, you have a perfect choice of abstracting to a relevant level (assuming that enough analysis has been performed). For example, the figure 3 shows three levels of abstraction.

The VCG provides the developers a method to easily gather knowledge about vulnerabilities and causes and make use of this knowledge in upcoming software projects. For example, we are convinced that if the design team for phpBB had used VCGs to model the vulnerability CVE-2002-0902, then they could have avoided the CVE-2006-0437 vulnerability due to the knowledge to be found in the VAD (had it existed).

The VCGs provide an overview over the relationships between vulnerabilities and their causes, which is easier to comprehend than for example reading vulnerability reports. In fact, during the writing of this report, we thought about modeling the vulnerability CVE-2006-0438 along with the vulnerability CVE-2006-0437 because they usually are described together. But no matter how many times we read the vulnerability description, we were not able to understand it. We believe that if the vulnerability was described by a VCG, we would have been able to understand it, mostly because the causes are connected and provides the reader with an overview.

Another example would be the CVE description of CVE-2006-0437, which does not fully match the actual vulnerability. While the description says the '<' and '>' characters are unchecked, this is in fact the only characters that are checked, and this is not the cause of

the flaw. It is a reasonable assumption that a third-party person wrote this text from his own understanding of the vulnerability, based on the description of the person who found the flaw. This resulted in an incorrect description for the vulnerability. If it would have been described by a VCG instead, we think that this kind of misunderstanding could have been avoided.

4.2 Disadvantages

If the goal of modeling the vulnerabilities is that the VCGs replace the way the vulnerabilities are actually described, it could result in a loss of information about the vulnerabilities. Compared to the reports that are posted when vulnerabilities are encountered, the VCG method does not provide any *elaborate* descriptions of the vulnerability or its causes. In some reports and for some vulnerabilities, the author makes a great deal of describing the flaw in detail. In some cases, examples of how to attack the vulnerability is given along with the code containing the flaw. This gives a lot of information to the developer interested about how to prevent the vulnerability. The fact that this is lacking in the case of VCGs is natural, because the method strives for being general instead of being case-specific.

This generality property demands that the nodes in the graphs are perfectly clear and unambiguous as not to lead to misunderstandings or difficulties to interpret the graphs. The care and effort that has to be put into the work is thus set at a high level, hence it could otherwise lead to consistency problems within the VAD. We think that a good balance between generality and preciseness is difficult to accomplish.

Another concern with this method that we have found is that it is hard to know when to stop modeling. As you partition your VCG by splitting simple nodes or turning them into compound nodes, it is hard to decide if you should continue your analysis or not. This is maybe not so much a *disadvantage* of the method as it would probably be better described as a *difficulty* encountered when using the method. We think that the developer will overcome this difficulty with gained experience. The exit criteria just states that analysis of nodes should be performed until "… *no more changes or additions to the VCG can be found*" [2]. How does one know when no more additions to the VCG can be found? This problem is strongly connected to the next problem encountered when using the method: to know at what abstraction level the analysis should be held at.

When you are making your analysis of a vulnerability and its causes, you have the possibility to dig into deep details (and thus convert simple nodes to compound nodes) at almost every possible cause. We think that the method is not really clear about what abstraction level that qualifies a simple node or a compound node, respectively.

5. Discussion

The vulnerabilities we have chosen as test cases did not open up any opportunity to try the more advanced parts of the VCG method, as more complex graph transformations. Also, we did not have the opportunity to get the experience of using an existing VAD for our case studies.

It should be noted that it requires a lot of time to develop the graphs, mostly because the causes requires a lot of effort to find. This means that the method is expensive. As the original paper [2] suggests that an analyst team should review all graphs inserted into the database before confirming them. Then we could imagine that an automatic process ensures that the VAD is duplicate-free and consistent so there is no duplicates of the same causes in the database. However, these duties require a lot of time though, and the company utilizing this method must balance the cost against the use of implementing and using a VAD.

The big problem we see is how to design the manual or automatic processes that make sure that there are not any duplicates of causes or vulnerabilities in the VAD and how this database should be maintained.

6. Related work

From the web developer's point-of-view, the only way to prevent XSS attack is to make sure that their scripts do not contain vulnerabilities. While modeling XSS vulnerabilities can improve their understanding, the usual way to mitigate vulnerabilities in web services is to get experience from best practices [6]. Explicitly setting the character encoding, identifying the special characters, using filtering techniques and examining cookies are example of techniques that a web developer should always have in mind.

In Vogt et al. [5], the authors described and implemented a method to prevent a XSS attack from the client-side. While there exist several mechanisms to protect from cross-site scripting attack on the server side, few approaches has been developed to prevent leakage of sensitive data on the client side. The authors of the paper mentioned above designed and implemented a method to detect and prevent cross-site scripting. Based on the fact that the number of sensitive data to be stored is limited (cookie, location, referrer, ...), they designed a method to taint and track sensitive data into the JavaScript engines and even the DOM tree, using both a dynamic and static data tainting approaches, so that the browser is alerted when tainted data is sent to an untrusted web site. The results of their experiments show that most of the tainted data sent were sent to companies that collect statistics about traffic on the web sites of

their customers, which is most likely to be legal because it is specified in their privacy policy. While the use of web scripts to send sensitive information with the consent of the privacy policy of the web site is out of our field of interest, it is noticeable that this method does success to prevent real XSS attacks and only generates a small number of false warnings. As such we believe that in the future this method should be available in browsers.

7. Conclusions

Vulnerability Cause Graphs are visual representations of vulnerability modeling allowing the developer to get an overview of the connections between the vulnerabilities and their causes. While offering arbitrary level of abstraction and relating the causes together, the method provides an easier understanding of the vulnerability and allows a quicker comparison with other vulnerabilities. On the other hand, the use of this method may result in a loss of information about the vulnerability and it requires an additional effort in the modeling of the nodes so that the graph does not lead to misunderstandings.

Our experience with the method in the context of cross-site scripting shows that VCGs can be very useful for preventing vulnerabilities in web development.

A company or an organization could benefit from the use of VCGs by gathering knowledge about vulnerabilities and causes and make use of this knowledge in upcoming software projects.

References

[1] S. Ardi, D. Byers, and N. Shahmehri. ”Toward a structured unified process for software security”. In *Proceedings of the ICSE 2006 Workshop on Software Engineering for Secure Systems (SESS06)*, 2006.

[2] D. Byers, S. Ardi, N. Shahmehri, and C. Duma. ”Modeling Software Vulnerabilities With Vulnerability Cause Graphs”. In *22th IEEE International Conference on Software Maintenance (ICSM’06)*, 2006.

[3] D. Gollmann: “*Computer Security*”, 2nd Ed. ISBN 0-470-86293-9

[4] The common vulnerabilities and exposures list. `http://cve.mitre.org` (accessed April 22 2007)

[5] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel and G. Vigna. ”Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis”.

[6] CERT/CC Understanding Malicious Content Mitigation For Web Development. `http://www.cert.org/tech_tips/malicious_code_mitigation.html` (accessed April 22 2007)

[7] Differences between pager.inc.php 1.1 and 1.2. `http://phpesp.cvs.sourceforge.net/phpesp/phpESP/admin/include/lib/adodb/adodb-pager.inc.php?r1=1.1&r2=1.2` (accessed April 21 2007)

[8] Neophasis Archives Bugtraq, 2002-05 `http://archives.neohapsis.com/archives/bugtraq/2002-05/0234.html` (accessed April 25 2007)