The Evolution of a Security Control or *Why Do We Need More Qualitative Research of Software Vulnerabilties?* (Transcript of Discussion)

Olgierd Pieczul[1,2(✉)] and Simon N. Foley[2]

[1] Ireland Lab, IBM, Dublin, Ireland
olgierdp@ie.ibm.com
[2] Department of Computer Science, University College Cork, Cork, Ireland
simon.foley@imt-atlantique.fr

Hi, my name is Olgierd Pieczul and this is a joint work with Simon Foley. Inspired by the theme of today's workshop we decided to look at evolution of security controls and vulnerabilities.

Today, evolution of software vulnerabilities tends to be researched mostly by using various types of quantitative analysis. These studies often take large numbers of software components, or security advisory records, and process them automatically. Based on that they make broad claims about the health of software security, identify trends, and so forth. These results are, however, somewhat expected, if not entirely, obvious findings. Although quantitative analysis provides some insight into general trends of vulnerability evolution, it does not really help to understand how and why software vulnerabilities and protection mechanims evolve. This is due to the fact that the studies are often based on data that is easy to acquire and process, for example, synthetic metrics such as CVSS. They are straightforward to analyze at a large scale and draw conclusions.

However, if you look at vulnerabilities in more detail, you fill find plenty of valuable information that is much harder to process. Unstructured, data such as bug reports, mailing list discussions or blog posts, that may give some insight into what is really going on and may help you with "why" and "how".

The problem is, how do we process this kind of data? Inspired by qualitative research we attempted to apply qualitative-style techniques to software vulnerabilities analysis. Typically, these techniques take raw, unprocessed data, such as interview transcripts and systematically analyze it in search for emerging themes. An example of that process has been presented at this workshop in 2009, where common expectations related to privacy of sharing photographs were established through a qualitative analysis of interviews.

Of course performing qualitative analysis at the scale seen in quantitative studies is not practical. We decided to perform a small scale experiment. Unlike in quantitative research, where simple metrics are analyzed for a large number

J. Anderson et al. (Eds.): Security Protocols XXIV, LNCS 10368, pp. 85–89, 2017.
DOI: 10.1007/978-3-319-62033-6_10

of security problems, we focused on a single component and looked much deeper into the available information.

The subject of our analysis was Apache Struts, a very popular MVC web framework that is used by a number of products, including enterprise. We scoped our analysis to a specific part of Struts, responsible for dealing with request parameters and cookies. This component automatically extracts parameter and cookie data from HTTP requests and passes it to the underlying application using the OGNL language processor (Apache's Object-Graph Navigation Language used for getting and setting properties of Java objects). This component has had a number of security vulnerabilities over the timeframe of our investigation which was 12 years, from 2004 to 2015. Effectively this was just a single security problem whereby Struts failed to handle the inputting of this data in a secure way. As a result attacker could manipulate the application, execute custom code and so forth.

How did we collect the data? The first, obvious source was official advisories. Many (but not all, as we later found) of these vulnerabilities have been published with CVE numbers and brief descriptions. This provided a base line and gave us an idea how many and what kind of security problems have been fixed by Struts developers. ¿From that we looked at the source code changes. We reviewed the entire source code evolution over the 12 years that related to the security control under study. That process was not particularly easy as the code moved between repositories and within the application structure number of times during the timeframe. Eventually we identified each code change related to every vulnerability. Having done that, we aggregated all the code changes into key categories. The details are included in the paper, but effectively there has been number of distinct areas of the code in which the security control operates: number of white lists and black lists, and different additional mechanisms and so forth. While identifying and aggregating code changes was the first step, we had already observed that the key code changes did not necessarily matched the actual vulnerability reports. This inaccuracy would make us question the veracity of (quantitative) studies that are based on this data alone for tracking vulnerability evolution.

Further, we looked at the bug reports in the project's bug tracking system, these often contain much more information than the public advisories. In particular, the developers comments on the reported issues were of interest. In many cases, we observed lengthy discussions regarding whether a particular issue should be considered a vulnerability, whether it requires a fix and what it should it be. Sometimes, bug reports on security issues were dismissed, only to become acknowledged as vulnerabilities later, and in some cases, years later.

We also investigated a number of blog posts that discussed Struts security. Sometimes the vulnerabilities were discussed into high detail. Usually the authors of the posts were the people who discovered the vulnerabilities or people who performed some post-advisory analysis. An interesting element of the blog posts were the reports on author interactions with the Struts developers, the process they followed, their reactions to the report, and so forth.

Having all of these key code changes identified and correlated with findings from other raw data we identified common phenomena related to vulnerability fixing and security control development. There's not time to go through all of them now, but I will just give you an example of the repeating patterns we have observed. All of the phenomena are discussed in the paper.

One of the findings is that security metrics are notoriously assigned incorrectly. For example, the (effectively) same vulnerability, with almost the same title has been reported three times over four years. Each of the security advisories describes a vulnerability with exactly the same impact and almost identical attack vector. First is for the original vulnerability discovered, while the other two for incomplete fixes that did not remediate the bug completely. Even though, each time the vulnerability has been reported with different severity score, higher each time.

Simon Foley: You could say that as they repeatedly tried to fix the same problem their understanding increased and as a result it was that increased their subjective opinion on the CVSS measure.

Reply: Yes. There were two vulnerabilities, one after another, one being a more general and having an incomplete fix, and the other having fixed to the gap. As you can see the CVSS metrics and the actual severity on the advisory were completely different.

Another phenomena is opportunistic fix. At the slide, you can see several code changes over a number of years related to the same security control. As the request parameters in Struts are interpreted as OGNL statements, and OGNL allows executing arbitrary code, it has to be restricted. Rather than properly disabling certain OGNL features, initially the restriction was enforced using regular expression on the parameter string. Over time, a number of security vulnerabilities related to this was fixed using the same solution that was easily available. However, it was rarely the right way to fix. For example, at one point the developers discovered that an expression for invoking constructors was allowed by the filter. The fix was to remove the white space from the white list as it effectively prevents the constructor expression that requires a space after "new" keyword. While this makes it technically impossible to invoke constructors it definitely is not the proper solution for such a critical vulnerability. However, it only required a single character change in the existing code. We observed that often this, opportunistic of way of fixing bugs led to more security problems later on. Another effect was that, over time, the regular expression became completely incomprehensible. Eventually after number of incomplete fixes the developers decided to apply a proper remediation, that is, disable expression execution at OGNL processor level.

The phenomena we discovered tend form a particular pattern in themselves. Once developers gain more understanding of what the problem is, what is the vulnerability that they are dealing with, what should be the proper solution, they would make different type of mistakes, than they had done at the beginning. It evolves over time and as larger software components are built from smaller, the issues repeat. The improper mechanism implementation on one side increases

the misunderstanding of what the software can do on the other side. We find that this repeating nature in the phenomena particularly useful because it may be possible to turn it into a checklist that may be used during the software development process. This helps to understand more on how and why software vulnerabilities appear and evolve.

Simon Foley: For those of you who were at the workshop last year, you might remember the presentation that we gave. We claimed that developers introduce flaws into their code as a result of using APIs in a complete way, and we called this the *dark side of the code*. In that presentation we gave a simple example that nicely illustrated the problem (a developer forgets that a URL 'web address' can have the file method). The example was conjecture in that it seemed reasonable but we didn't have systematic evidence that these kinds of vulnerabilities truly happen, *in the wild*. Today's paper provides the systematic evidence that developers really do make these kinds of mistakes: we looked at the detail of over 300 code changes in struts and found repeated concrete evidence that developers were programming in the dark side. This qualitative result is not note something that can be easily discovered using quantitative techniques.

Reply: Our experiment, at a small scale, provided evidence to support some of the hypotheses of our previous work. Similarly, the developers blind spots phenomena, has been proposed in the past based on a controlled experiment involving interviewing developers. In our study we have seen this developer's blind spots phenomenon on real life made by real people in the real software.

As a closing point, I would like to remark why we think that using metrics is not fit for purpose for looking at security evolution. Over time, gaps in previous fixes appear as new vulnerabilities. This already creates a bias because one only counts vulnerability records they may think of them as independent issues, while in fact, often there are the same issues, just not fixed correctly in the past. Unless you spend time and effort to correlate them, you cannot really draw conclusions about evolution of the software security.

Also, security problems and vulnerabilities are often misunderstood by those who write advisories. Often, they only describe developers' interpretation of what the problem is rather than describe the actual problem. In the paper we have number of examples of advisories with too narrow scope or incorrect impact.

Finally, the severity metrics which we have shown are all often assigned incorrectly and not corrected when true nature of the issue is discovered later on. As we have shown in the example, subsequent instances of the same problem usually have higher severity because they are better understood. It does not mean that software has more severe vulnerabilities, it just means that the previous vulnerabilities had incorrectly assigned low severity which never got corrected.

Any questions? Comments?

Simon Foley: We should consider the debate about the relative values of qualitative versus quantitative research; both can be equally useful or lacking. If you would display again the picture depicting the phenomenon (Fig. 3 in the paper). Remember that these phenomena emerged as a consequence of looking at very

large amount of code, vulnerability reports, and so forth, in great detail. Carrying out this qualitative analysis we can point to concrete evidence of developers programming in the dark side, that developers make opportunistic fixes rather, and so on. Now compare that evidence, with the examples that you gave at the start of the talk on results from quantitative studies based for example on CVSS scores. One example was *"In this paper, we examine how vulnerabilities are handled in large-scale, analyzing more than 80,000 security advisories published since 1995. Based on this information, we quantify the performance of the security industry as a whole.* and I ask you, which is the more useful?

Reply: Yes. My message is do less of these (quantitative studies) and do more of that other one (qualitative studies). Of course qualitative studies can be much harder to do, but even doing them in a small scale can give you much better results. Thanks.