The Effect of Pre-written Scripts on the Use of Simple Software Security Analysis Tools

Matti Mantere[1], Kaarina Karppinen[1] and Mika Rautila[2]

Software Technologies, Security
VTT Technical Research Centre of Finland
[1]Kaitoväylä 1, 90571 Oulu, Finland
[2]Vuorimiehentie 3, 02044 Espoo, Finland
matti.mantere@vtt.fi
kaarina.karppinen@vtt.fi
mika.rautila@vtt.fi

Abstract: In this paper we study the effect of integrating lightweight, open source, static code security analysis tools using Ruby and shell scripts. Particular emphasis is placed on the effect of tool usability by this approach. By scripts simple analysis methods could be created so that used tools themselves were able to remain completely hidden from the end user. Scripts were used for automatically fetching the relevant source packages, patching them to the right versions and running different analysis tools on the target. Analysis cycle was fully automated and produced rough results of the nature of flaws present in the source material. The overall user experience and ease-of-use of the tools were improved considerably with the pre-defined scripts. This improvement was distinct especially on the analysis phase. With the scripts it was easy to have a cursory estimation of a general risk-level of the target application. This estimation could later be used for deciding further security analysis priorities or other things, dependent of the tools and heuristics used.

1 Introduction

Use of even simple tools designed for software security analysis usually requires some skill and time. In this paper the effect of pre-written scripts to run these tools and analyze their output is studied in relation to their overall usability. A selection of three simple, open source, static analysis tools is used as an example case. The scripting language used to integrate the execution and analysis of the tools is Ruby.

This approach of using simple static code analysis tools that are run by scripts was tested while going through a very large database of software packets during the Sameto (Security Assurance Methods and Tools) project [Sa10]. The approach of writing scripts to integrate and automate executions of a set of tools is not a new idea in itself but its effect on usability has not been noted before. We saw the script writing to be a very useful way to accomplish the needs of our project, where the analysis of the amount of packages involved was not feasible using manual analysis, but rough results could be obtained by automating the process.

Analysis can combine the output of various tools to produce a single report of the target. In our opinion this approach is best suited for simple lightweight tools that could benefit from grouping them with other similar tools. Related research, as described in [CW07], shows that in the case of very polished, easy to use and efficient security analysis tools such as some of the leading commercial solutions this approach loses most of its core idea, as those tools produce the needed output by themselves.

2 The Case Study Setup

In our case study three static code analysis tools were loosely integrated to by the scripts. These tools were assigned to scan the source code for security related programming errors such as buffer overflows and race conditions. The target packages consisted of a large amount of source code obtained from Ubuntu Linux distribution [Ub10]. Scripts were used 1) to automatically fetch the relevant source packages from the Ubuntu project repositories, 2) to patch them to the version to be studied, and after that 3) to run different analysis tools on the target. The whole analysis cycle was automated and produced the sought rough results of the nature of flaws present in the source material. The output from scripts fed to the database was then used to produce statistical information concerning the security flaws present in the Ubuntu distribution.

The tools used were CppCheck [Cp10], RATS [Fo10] and Flawfinder [Fl10], with RATS and Flawfinder being rather obsolete by themselves but serving an auxiliary purpose in the analysis. RATS and Flawfinder provide similar output, listing found risky structures and functions listed in their databases. Extensive use of these risky structures is usually a thing to be avoided. Even if the structures are used safely, they leave extra room for classic errors such as buffer overflow. Cppcheck is a more sophisticated tool, tuned to produce as few false positives as possible, but prone to producing false negatives in the process. They are all open source software, used from command line, and produce their results in a very simple textual format. The command line interface was used for the scripts, and in the case of these tools there was no need for any further APIs.

The main scripting language used was Ruby; both version 1.8 and version 1.9 were used. Ruby offers an interface for calling the system commands that is very simple to use. A few Linux shell scripts were also used in combination with the Ruby scripts when action needed was easily accomplished using the standard Linux tools. The text output produced by the tools was very easy to parse and input to the heuristics was used to calculate relevant attributes.

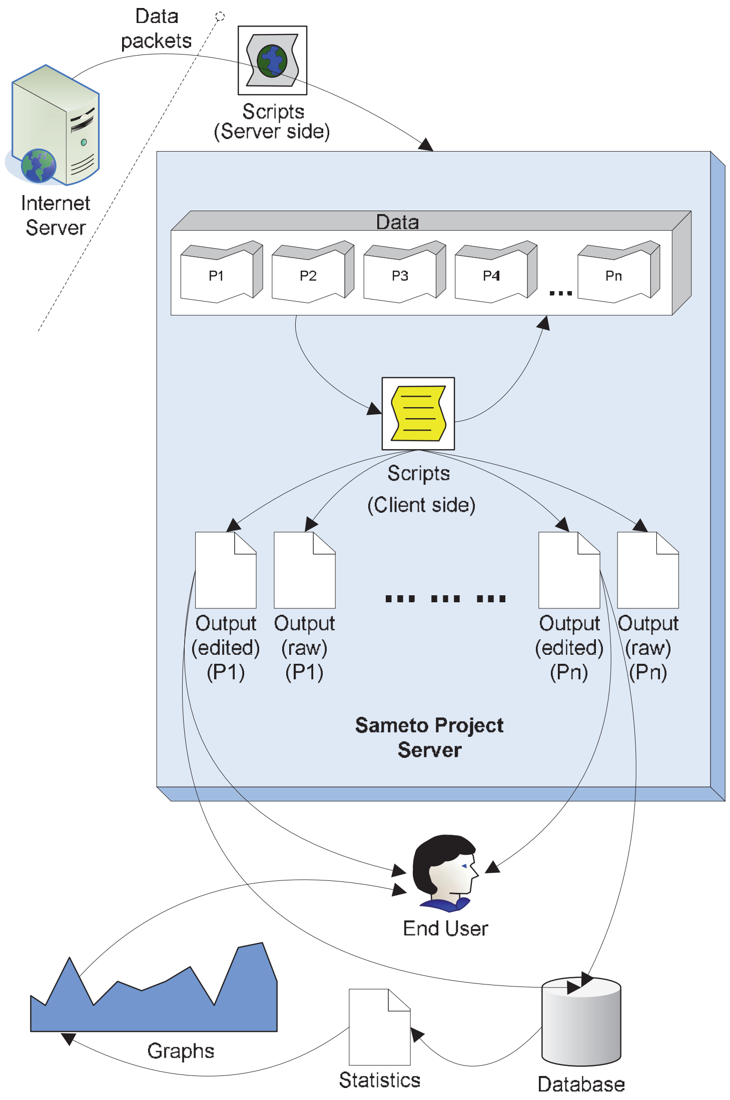

Figure 1: The case study setup

A Linux server was used in the example. The server was configured to fetch the relevant source packages and to perform the analysis on them. All this was accomplished by using Ruby scripts and simple Linux services, such as cron.

Our case study setup is illustrated in Figure 1. The server side scripts are programmed to fetch the relevant data packets from an Internet server, in our case Ubuntu, unzip the packets, and patch them to the version to be studied. After that, the client side scripts run different analysis tools on the target data packets which consist of source code. A closer look at the client side scripts is given in Figure 2.

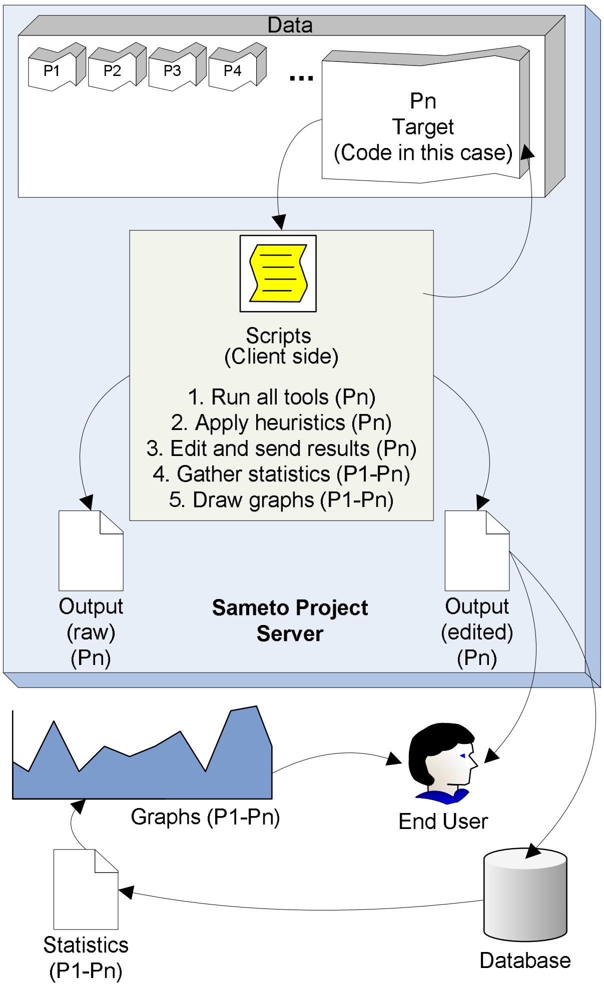

Figure 2: The activities of client side scripts

First the scripts run the assigned tools and receive the output from all of them. After the tools have been run the scripts take care of making the output files. Heuristics are then applied to the collected output. Much of the quality of the output relies on the proper formulation of the heuristics, which are still very much under work. With poor heuristics the output from the integrated tools will be only the output from each individual tool without any added value. The raw output files are still also saved for a possible later use. Then the results are edited to a report consisting of the edited output files, a summary and an analysis of the output and the heuristic calculations, which is sent to the end user and saved to the database. These results can later be used to gather statistics and form graphs according to the needs of the user.

3 Simple Tool Integration and Automation

Integration of simple tools by writing scripts in languages such as Ruby can be used to create simple analysis methods for users uninformed of the tools or their usage. In this way, the tools themselves can remain completely hidden from the end user. The analysis performed by the scripts is naturally not at the same depth as that of an expert doing analysis, but is light-weight and very easy to use and can be done even by a novice. Thus rough results can be obtained with very little effort after the scripts have been brought to a level where they work at a sufficient efficiency.

If needed, even complicated tools can be used in the same fashion as the simple ones, but their integration can be far more demanding. The heuristics used can be as complicated as desired, or they can be configured only to look for some single instance in the tool output. When tools are integrated to be run by scripts with preordained flags, the benefit of receiving output from multiple tools by a single call of the script is gained. This output is then collected and edited to form a single report with arbitrary metrics calculated from the data pool generated by the used tools. By automating the running of these scripts a relatively up-to-date database of information concerning the target codebase is always at hand.

Adding new light-weight tools to an existing script is easy. One of the strong points of this approach is the possibility to use nearly any number of light-weight tools.

4 The Effect of the Tool Usage

The main goal in integrating and automating tools in this approach is to achieve decreased time consumption and the ease of later use of the tools. After the scripts have been written, they can either be automated to run at certain time intervals, or be assigned to run by the end users. If the scripts are written and automated by a party other than the end user, the end user merely needs to use the script output, without even needing to run the script himself. This can be beneficial in cases when the code base is large and the user needs to access the script output without extra effort. In those cases when the integrated tools have long execution times, it is beneficial that the system automatically checks the most recent versions of target files and ensures that recent reports are available.

4.1 Effects Related to User Experience

Human factors and issues have traditionally played a limited role in security research and secure systems development and testing [CG05]. In the past couple of years, researchers around the world have begun to notice that there really is a trade-off situation between security and usability; e.g., the first book [CG05] about these issues was published a few years ago. One aspect of this trade-off is the existence of security analysis tools that are designed to be used merely by senior developers who have a sound technical experience of using different kinds of analysis tools and interpreting their results. However, there is a need for easy-to-use security analysis tools, also within the open source software range.

Usability is usually seen as a quality attribute that affects the end user. Nielsen [Ni93a] stresses that usability is not a single, one-dimensional property of user-interface. According to Nielsen [Ni93b] usability consists of five dimensions, namely: (1) Learnability: Ease of learning; the user can quickly go from not knowing the system to getting work done, (2) Efficiency: High level of productivity, (3) Memorability: Ease in remembering how to use the system, (4) Low error rate: Users do not make many errors during use of the system, or if they do, make errors they can easily recover from, and (5) User satisfaction: The system is pleasant to use, users are subjectively satisfied, and they like it.

We evaluated the three static code analysis tools within the project group. Each of the three tools was used individually with predefined tasks. We noted down the user experiences while running the tools designed for software security analysis. When completing the tasks the project group also wrote down their impressions about the overall look and feel that they got from using the tool. Afterwards the characteristics of the tools were compared with the above mentioned five usability dimensions.

Here are the results of the comparison: (1) Learnability: it takes a considerable amount of time for a novice to learn to use the basic interface, (2) Efficiency: it takes time to analyze the results in a correct way and overall use of the tools is not efficient, (3) Memorability: manuals and help files are needed to run these tools and manuals are not extensive as is the case with many open source programs, (4) Low error rate: tools are all used from command lines which in addition to memorability problems causes also errors based on typing or spelling mistakes, and (5) User satisfaction: the users were not subjectively satisfied in using the tools. The conclusion was that usability is obviously not very well thought of when designing any of these tools, and the overall ease-of-use of the three tools in our experiment is not very good.

Our evaluation continued with the system with the pre-written scripts. Based on our experiences, these pre-written scripts to run these tools and analyze their output saved a considerable amount of time and effort, and with them the above mentioned five usability dimensions were better fulfilled. Learnability and memorability were easier as there were only a few commands which the user needed to remember. Efficiency was improved extremely as the scripts took care of the hard work of going through and analyzing the results. In consequence of all these results the user satisfaction was improved substantially with the pre-written scripts. Actually, the tools and their usage were completely hidden from the user; all the end user needed to know was how to start the analysis and how to interpret the results received in a form of a report. Thus the scripts had a considerable positive effect on the user experiences.

4.2 Effects Related to Analysis Phase

In the Sameto project the approach presented in this paper was used to analyze is a large database of code. The results gained were used to determine whether the methods found any of the errors known to exist in the code, whether the output differed between the patched and unpatched versions of the same source files, and if any new and unknown errors could be discovered. Several possible errors were discovered with the automated analysis, which will require further investigation. The environment is still running and producing more statistical information concerning security notices and patches emerging concerning the Ubuntu Linux distribution.

It must be noted that the main advantage with the pre-defined scripts was gained within the analysis phase when going through the results of the tools, regardless of the skill level of the user. The analysis of the vast amount of packages involved in our case study was not feasible using manual analysis, but by automating the process rough results could be obtained. Even with a smaller amount of packages, when manual analysis would be possible, without the scripts the tools should be used one by one and each of them would give their own set of security related programming errors. With the scripts the findings were received in one file including assorted results of all the tools used, and what is most important, an automated analysis with a checksum summing up the severity of the errors based on the predefined heuristics which are emphasized based on their relevancy. With this result it is extremely easy to have a cursory estimation of a general risk-level of the target application. This estimation can then be used for deciding further security analysis priorities or other things, dependant of the tools and heuristics used.

4.3 Quality of Results

The approach of integrating multiple light-weight tools by scripts adds to the overall quality of results when used properly and compared to the results obtained from any of the integrated tools by itself. This requires that the scripts are correctly written, correct data is collected from the individual tool reports, and properly designed heuristics are used when computing the metrics used. Even then, the results from these automated lightweight tools are only indicative of the overall general quality of the code. The approach can be used to aid the work of programmers not well-versed in the finer points of software security. Automating lightweight tools by simple scripts can in no case be used to replace the auditing of the code by experts. Naturally, all the tools should be developed to be so easy to use that these kinds of intermediary helpers would not be needed, but as this is not the case at present these kinds of aids can be of use.

There are a couple of restrictions in our experiment. Firstly, our study was based on a novice user executing the tests. The outcome could be different if the user was a very experienced person who would instinctively know how to use the interfaces of various tools and what issues to consider when analyzing the results.

Secondly, at this point our scripts include only example heuristics that are used for calculating the relevant attributes for the analysis of the software packets. There are still various issues to be considered with the values as well as the weighing of different coding errors and their severity before these results can be exploited more widely. We will examine these issues further in our future projects.

5 Conclusions

In this paper the effect of pre-written scripts to run a selection of simple static security analysis tools and analyze their output was studied in relation to overall usability of these tools.

The overall user experience is improved considerably with the scripts, as there is no longer any need for the end user to directly interact with the tools, requiring only very rudimentary knowledge of running the scripts in question. This improvement was mostly focused on the analysis phase as the large amount of packages involved could not be analyzed manually but could provide rough analysis results by automating the process, so the scripts were well worth the effort of writing.

The increase in usability and significant decrease in time consumption came at a price of depth and accuracy of analysis. In situations where this trade-off is acceptable, this approach can be selected.

The utility of the results can well be increased by substituting the lightweight tools for more advanced tools and further developing any heuristics used. The added cost with more advanced tools generally includes either increased expenses, added difficulty writing the scripts or both.

Acknowledgments

This paper was written as a part of the SAMETO project at the VTT Technical Research Centre of Finland.

References

[CG05] Cranor, L.F., Garfinkel S.: Security and Usability – Designing Secure Systems That People Can Use. O'Reilly Media Inc, 2005.
[Cp10] Cppcheck, http://sourceforge.net/projects/cppcheck/, last accessed 08 February 2010.
[CW07] Chess, B., West, J.: Secure Programming with Static Analysis, Addison-Wesley, 2007
[Fl10] Flawfinder, http://www.dwheeler.com/flawfinder/, last accessed 08 February 2010.
[Fo10] Fortify, RATS Rough Auditing Tool for Security, http://www.fortify.com/security-resources/rats.jsp/, last accessed 08 February 2010.
[Ni93a] Nielsen, J.: Usability Engineering. Academic Press Inc, Boston, 1993.
[Ni93b] Nielsen, J.: Iterative User-Interface Design, J. Computer. 26, no. 11, 1993; pp. 32-41.
[Sa10] Sameto project, http://www.vtt.fi/vtt_show_record.jsp?target=tutk&form=se&search=9228/, last accessed 08 February 2010.
[Ub10] Ubuntu Linux, http://www.ubuntu.com/, last accessed 08 February 2010.