Evaluating Static Source Code Analysis Tools

by

Thomas Hofer

B.S., École Polytechnique Fédérale de Lausanne (2007)

Submitted to the School of Computer and Communications Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

at the

École Polytechnique Fédérale de Lausanne

April 2010

© Thomas Hofer, MMX. All rights reserved.

The author hereby grants to EPFL and CERN permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part.

Author . . . .
School of Computer and Communications Science
March 12, 2010

Evaluating Static Source Code Analysis Tools

by

Thomas Hofer

Submitted to the School of Computer and Communications Science
on March 12, 2010, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science

Abstract

This thesis presents the results of an evaluation of source code analyzers. Such tools constitute an inexpensive, efficient and fast way of removing the most common vulnerabilities in a software project, even though not all security flaws can be detected.

This evaluation was conducted at CERN, the European Organization for Nuclear Research, in the intent of providing its programmers with a list of dedicated software verification/static source code analysis tools. Particular focus of these tools should be on efficiently finding security flaws. The evaluation covered close to thirty different tools for five major programming languages.

Thesis Supervisor: Philippe Oechslin
Title: Lecturer

Thesis Supervisor: Sebastian Lopienski
Title: Deputy Computer Security Officer at CERN

Thesis Supervisor: Stefan Lueders
Title: Computer Security Officer at CERN

Acknowledgments

I feel particularly grateful to the supervisor with whom I shared offices, Sebastian Lopienski, for his many insightful comments over the months, for his friendly guidance and for his enthusiasm. I would also like to thank Dr. Stefan Lueders for his continued ability to suggest improvements and for a very interesting tour of CERN. My thanks as well to Dr. Philippe Oechslin for his support and his helpful comments. I am indebted to my parents for bearing with me this long and for so many other reasons. Erin Hudson has my undying gratitude for her very thorough and cheerful comments. Renewed thanks to Luis Muñoz and Lionel Cons for their help sorting out the Perl::Critic policies. Finally, many thanks to the Computer Security Team at CERN for creating such an enjoyable working environment!

Contents

1 Introduction **1**
1.1 Current State of Research . . . 1
1.2 Current Status at CERN . . . 2
1.3 Goals and Deliverables . . . 3
1.4 Organization of this Thesis . . . 3

2 Understanding the Tools **4**
2.1 Model Checking . . . 5
2.2 Control and Dataflow Analysis . . . 5
2.3 Text-based Pattern Matching . . . 6

3 Defining the Metrics **8**
3.1 Basic Considerations . . . 8
3.2 Metrics Used . . . 9
3.2.1 Installation . . . 9
3.2.2 Configuration . . . 10
3.2.3 Support . . . 10
3.2.4 Reports . . . 10
3.2.5 Errors Found . . . 11
3.2.6 Handles Projects? . . . 11

4 Presenting the Results **12**
4.1 C(++) Source Code Analysis Tools . . . 12
4.1.1 Astree . . . 13
4.1.2 BOON . . . 14
4.1.3 C Code Analyzer . . . 14
4.1.4 Code Advisor (HP) . . . 14
4.1.5 Cppcheck . . . 14
4.1.6 CQual . . . 15
4.1.7 Csur . . . 15
4.1.8 Flawfinder . . . 15
4.1.9 ITS4 . . . 16
4.1.10 Smatch . . . 16
4.1.11 Splint . . . 17
4.1.12 RATS . . . 17
4.2 Java Source Code Analysis Tools . . . 18
4.2.1 CodePro Analytix . . . 18
4.2.2 FindBugs . . . 19
4.2.3 Hammurapi . . . 19

4.2.4 JCSC . . . 20
4.2.5 IBM Rational AppScan . . . 20
4.2.6 PMD . . . 20
4.2.7 QJPro . . . 21
4.3 Perl Source Code Analysis Tools . . . 21
4.3.1 B::Lint . . . 21
4.3.2 Perl::Critic . . . 21
4.3.3 RATS . . . 22
4.3.4 Taint Mode . . . 22
4.4 PHP Source Code Analysis Tools . . . 22
4.4.1 Sandcat.4PHP . . . 23
4.4.2 Pixy . . . 23
4.4.3 RATS . . . 24
4.5 Python Source Code Analysis Tools . . . 24
4.5.1 PEP8 . . . 25
4.5.2 PyChecker . . . 25
4.5.3 Pylint . . . 25
4.5.4 RATS . . . 26

5 Delivering the Results **27**
5.1 CERN Computer Security Team Web site . . . 27
5.1.1 Packaging the Tools . . . 27
5.2 Post-C5 Presentation / IT Seminar . . . 28
5.3 Article in the CNL . . . 29

6 Conclusion **31**
6.1 Results . . . 31
6.2 Achievements . . . 31
6.3 Main Challenges . . . 32
6.4 Outlook and Future Developments . . . 32
6.4.1 At CERN . . . 32
6.4.2 Further Improvements to Software Security . . . 33

A Tools HOWTO **I**
A.1 Security Analysis Tools - Recommendations for C/C++ . . . I
A.2 Security Analysis Tools - Recommendations for Java . . . III
A.3 Security Analysis Tools - Recommendations for Perl . . . V
A.4 Security Analysis Tools - Recommendations for PHP . . . VI
A.5 Security Analysis Tools - Recommendations for Python . . . VIII

B Sample Output **X**

C Additional contributions **XI**
C.1 Wrapper for Pixy . . . XI
C.2 Script to Retrieve Wiki Pages . . . XX

List of Tables

3.1 The Metrics Table. 9

4.1 Evaluation Results For C/C++ Tools 13
4.2 Evaluation Results For Java Tools 18
4.3 Evaluation Results For Perl Tools 21
4.4 Evaluation Results For PHP Tools 23
4.5 Evaluation Results For Python Tools 25

1 Introduction

1.1 Current State of Research

Over the past fifteen years, computer security has been getting growing attention and publicity. Security flaws can lead to disclosure of confidential information, loss of functionality or damaged image of a company, amongst other issues. Computer security is relevant and should be considered whenever computers are used, *e.g.* when creating and sharing of documents, proving services or developing software and web applications. This work is concerned with aspect of the latter: development of software and web applications.

There are various complementary ways to approach computer security where development is concerned. Training the programmers is most effective before starting the development, yet still meaningful later in the process. Source code analysis can be performed whenever the code can be compiled and helps identifying potential risks early on. Once the software is almost ready to be released, it can be reviewed manually by experts, preferably external to the project. Lastly, dynamic analysis can be used to try and discover if the program misbehaves on unexpected input. Source code analysis is the main focus of this thesis.

Research on software analysis is quite extensive and aims either at verifying a program's correctness [1] or at optimizing its performance [2]. This thesis will exclusively focus on correctness related research and more particularly where security aspects are involved.

In 1936, Alan Turing proved that the *halting problem* is undecidable [20]. This has tremendous repercussions on the field of software verification. Indeed, if an essential and apparently simple property such as termination can be impossible to prove for some programs, how can one expect to prove much more complex statements about security considerations?

It is important to note that the undecidability of a property does not imply that it is impossible to prove it for *any* program. There are infinitely many programs that can be proven to terminate on an infinite set of inputs. Therefore, theoretical and formal approaches such as abstract interpretation [3] or model checking [17] still hold their place and have a role in this area. However, simpler and less thorough methods tend to yield faster results while requiring less effort. The most trivial example

is the use of the Unix `grep` utility to find all calls to library functions that have been flagged as dangerous.

Over the past decade, there has been an effort to find a middle ground between these extremes. While some attempt to build up from simple tools, to offer developers a light interface to identify the most common errors, such as **ITS4** [22], others work on heuristics and abstractions to make formal methods more efficient and reliable.

1.2 Current Status at CERN

CERN, the European Organization for Nuclear Research, was founded in 1954 as one of Europe's first joint ventures. Its purpose is research in fundamental physics, gaining a better understanding of the Universe. While fans of fiction know it as a place where antimatter is created [1], it should be noted that it is also the place where the web was born [2].

Of the many faces CERN has to offer, the one which will be most relevant to this thesis is its very particular software development environment. While CERN only employs about 2500 people, about three to four times that many visiting scientists spend part of their time at CERN and need computer accounts to be able to conduct their research there [3]. Each of those people has the opportunity to create a Web site to host a personal project, present information about their latest research or run a Web application. Any Web site can be restricted to internal usage at CERN or open to the rest of the world. While the vast majority of such Web sites only contains static content, *i.e.* no code, diversity and freedom make this a difficult environment to control.

Furthermore, a few hundred software projects are being developed for various purposes, from assistance with daily HR tasks to control and monitoring of the particle accelerators. While key projects such as these are developed with some concern for security, many of the smaller ad-hoc software projects are developed by people with little or no training in secure software development – although this number is diminishing as regular courses on the topic are scheduled.

This implies that vulnerabilities are bound to appear in programs used in production at CERN. While it is not feasible in any reasonable time at reasonable cost to get rid of all the vulnerabilities – Secunia[4] releases multiple security advisories daily for reputable and security-aware products –, one should always try and make the best out of the time available for computer security.

At CERN, computer security requirements are often complicated by the necessity for academic freedom, which is essential to the advancement of fundamental research. Also, the Computer Security Team has limited direct authority over users; except for a few exceptional measures in case of immediate threat, matters are handled through hierarchical channels. On top of this, due to the size of the organization, it is impossible to control and verify all of the software in use at CERN.

[1] Read `http://angelsanddemons.cern.ch/` for information about the "science behind the story".
[2] `http://cern.ch/public/en/About/Web-en.html`
[3] Numbers taken from: `http://cern.ch/public/en/About/Global-en.html`
[4] Check `http://www.secunia.com/advisories`.

Therefore the responsibility for security always lies with the developers and providers of the services, rather than with the Computer Security Team.

In consequence of this, if any measures are proposed to improve the security of the software developed at CERN, programmers can not easily be ordered to, coerced into or made to use them. They need to be convinced that it makes sense for them to invest part of their time into improving their code. This is the key consideration in the definition of the metrics detailed in Chapter 3.

1.3 Goals and Deliverables

The global goal of this project is to improve software security at CERN. More particularly, there is interest in improving the security of the many pieces of software developed at CERN. It was agreed that this needs to be approached in a breadth-first manner, in the sense that the focus should be finding a tool that as many developers as possible would use, even if find finds fewer errors, rather than a tool that would find (almost) all of the errors for only a few developers.

The purpose of this project is to evaluate existing source code analysis tools, according to metrics prioritizing ease of use and "quick gains" over more efficient but more complex solutions.

The deliverables requested by the CERN Computer Security Team consisted mostly of documentation on how to install, configure and use the tools selected. At least one tool should be recommended and documented for each of the main programming languages in use at CERN, namely: C/C++, Java, Perl, Python and PHP. Those languages were selected according to recommendations of the Computer Security Team, and based on the proportion of standard lines of code (SLOC) in the source code database built from all accessible SVN and CVS repositories at CERN, as well as from the Web sites hosted on AFS. Finally, the tools should be packaged along with a default configuration, so as to make installation easier for the developers.

1.4 Organization of this Thesis

Chapter 2 gives some theoretical background on source code analysis. Chapter 3 explains the metrics used to evaluate the tools. Chapter 4 presents the tools that have been evaluated and the results of the evaluation. Chapter 5 discusses the means used to propagate information about source code analysis tools at CERN. Finally, Chapter 6 gives some perspective on further improvements to Computer Security, both about what is done in parallel and in extention of this project as well as more general considerations and concludes this thesis, discussing the results obtained, presenting some of the challenges faced and what this work has achieved.

2 Understanding the Tools

There are many ways to look at a piece of software to try to improve it. This chapter will discuss where static analysis fits in and some of the main forms it can take. Categories of software analysis tools can be established according to various criteria. One of the most relevant and interesting criteria is whether a process is "manual" or "automatic". Manual software improvement processes involve a developer or a quality analysis expert to actively evaluate the piece of software, make use of their previously acquired knowledge to understand it and identify its shortcomings. Automatic methods consist of a tool that performs a predetermined series of tests on the piece of software it is given and produces a report of its analysis.

Another criteria commonly used in differentiating the various existing methods of analyzing software is the approach they take. A method can either use a "black-box" approach, considering the whole piece of software as an atomic entity, or an "white-box" approach, examining the inner workings of the software, e.g. its source code. The "black-box" approaches most often consist of trying to give the piece of software being analyzed unexpected input parameters or unexpected values, thus hoping to trigger incorrect behavior. The "white-box" approaches vary significantly in their implementations and can for instance simply look for certain patterns in the code, reconstruct the logic or data-flow of the software or even attempt to simulate all possible executions of the program.

Where "black-box" approaches are concerned, automatic methods tend to produce better results. This is because it is easier, and faster, for a computer program than for a human being to list a large number of inputs likely to lead to faulty behavior. On the other hand, in most cases manual analysis yields much better and more accurate results when it comes to "white-box" approaches, because automated tools can only understand concepts that have been built-in by their developers, whereas human beings can learn to detect new types of errors more easily. However, manually reviewing millions of lines of code is a rather time consuming task, even if one focuses on the critical routines or methods, which have to be identified first. A comparison of manual versus automatic code review is presented by Miller et al. [16]. While it is clear from their results that manual code review can reveal key issues that are missed by some of the best tools, it is worth noting that the manual

reviewing process took the authors about two years. The time consumption of manual code review is so important that over the last few years much work has been invested into improving the results of automatic source code analysis. The following sections will present three of the major trends that have emerged in the area.

2.1 Model Checking

Model checking labels a group of formal methods for software and hardware verification. It aims to test whether a given system model complies with a specification. Applied to source code analysis, it verifies if a given implementation meets a set of requirements. Whether model checking actually fits within the automatic source code analysis category is debatable. Indeed, this technique requires the definition and description of properties that should be checked for a piece of software, most often in a tool specific format. However, once this specification has been established no additional human input is needed to perform the analysis.

The major class of Model Checking methods is based on Temporal Logic. Initial work on this topic is attributed to Emerson and Clarke [11], [6], [7] and to Queille and Sifakis [18]. While most of the research about Model Checking assumes that the model analyzed is given as a Finite State Machine, there are some Model Checkers available that are able to analyze `C` or `Java` source code directly. Former EPFL Professor T. A. Henzinger et al. [14], [5] have developed BLAST [1], a tool to check safety properties of C code.

Model Checking is perfectly fitted to the needs of software developers who need high confidence, or even guarantees, about critical programs, such as flight control systems for airplanes or docking systems for automated space vessels. However Model Checking remains an expensive and complex technique, not suited for integration in nightly builds of non security-critical projects, despite the efforts of the scientific community to make it more accessible and simpler to use.

2.2 Control and Dataflow Analysis

One representation that is widely used in formal methods for optimizing and verifying source code is that of Control Flow Graphs (CFG). In a CFG, each node represents a basic bloc of code, *i.e.* lines of code that will not be executed independently during normal program execution. In other words, the nodes do not contain any jumps or labels (jump targets). The jumps are represented by directed edges. Usually CFG implementations have two special nodes: the “entry point”, which is the starting point of the execution; and the “exit point”, where the execution exits the graph.

Based on this representation, some observations resulting from graph reachability are trivial: if a subgraph is unreachable from the entry point, the nodes that form this subgraph consist of *unreachable code*; if the exit point is unreachable from the entry point, there is an *infinite loop* in

[1] `http://mtc.epfl.ch/software-tools/blast/index-epfl.php`

the code.

Control Flow Graphs have little more practical use on their own, but they are used in various other techniques as a representation of all the possible paths of execution. One such technique is dataflow analysis [15]. Dataflow analysis can be performed by defining how each block modifies the data and then performing fixed-point iteration across the whole CFG.

This approach is particularly useful for optimization of the software. More relevant to the scope of this project, dataflow analysis can facilitate the identification of interdependencies in data and track the effects of user input.

Input tracking is essential in software security, as anything coming from outside the program should be considered as potentially malicious, including configuration files and inter-thread communication. If an untrusted value is used in a critical place in code without having been verified, it can lead to an exploitable bug. Therefore, dataflow analysis can be quite helpful in determining a system's security vulnerabilities. However, dataflow analysis suffers from high computational complexity and even with advanced heuristics, applying dataflow analysis to a large program can be very time consuming.

Coverity [4] developed a static analysis tool that uses dataflow analysis. Running this tool on one of the projects developed at CERN – the ROOT framework[2] – takes about 28 hours for slightly over 2 millions lines of code. While the developers of this project are very satisfied with the results they obtained from this run, this kind of tool requires some meddling into the compilation process and more time and effort than most CERN developers would be able and willing to invest into software security.

2.3 Text-based Pattern Matching

The simplest and quickest way to find some trivial vulnerabilities in C source code is to `grep` through the files for weak and potentially vulnerable functions calls like the `gets` or the `strcpy` library functions, amongst others. Indeed, `gets` is always vulnerable to buffer overflows and is nowadays even the cause of compiler warnings. This problem is actually so well-known that this function is likely not to be specified in the upcoming version of the C standard, dubbed C1X. A note available at `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1420.htm` mentions its removal from the standard, and the current draft (`http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1425.pdf`) of C1X does not include any mention of it.

While this method fails to be directly useful to non-specialists as such, for one has to know beforehand what particular patterns should be searched, some tools have been developed on this principle. Those tools come with a list of patterns that are known to lead to vulnerable code, or to be commonly misused, and sometimes contain short explanations of what the risk of using the identified function or construct could be. ITS4 [22] is an early example of such a tool. It looks

[2] `http://root.cern.ch`

for calls to functions typically involved in a security vulnerability in C source code, such as `gets`, `strcpy`, `strcat` or a wide variety of string formatting functions.

From a theoretic point of view, such an approach can seem somewhat limited, as tools of this category will only detect flaws that have been described to them explicitly, as opposed to the formal approaches discussed above, which could discover weaknesses in a more analytic fashion. However, the vulnerabilities found by these tools are the ones most commonly exploited and the difference in time and effort invested before obtaining results is abysmal in comparison to the methods presented earlier in this chapter.

3 Defining the Metrics

3.1 Basic Considerations

Based on the context presented in Section 1.2, key elements can be defined and prioritized. The end goal of this research is to increase software security at CERN by finding tools that make it as easy as possible for the programmers to identify and understand some of the security flaws in their software. Since the deadlines leave little margin to squeeze security in, developers need to be motivated to use the tools suggested, which is discussed in Chapter 5, and then it is important to ensure that their experience goes as smoothly as possible and that they do not encounter excessive difficulties in the process of using the tools. While it is impossible to offer any guarantees, the user experience with a tool can be divided in various phases, each of which will be cautiously examined and evaluated.

The first place where issues can appear is during installation of the tool. It is important that developers get a good first impression, as it will be a lasting one. The amount and availability of prerequisites, the accessibility of the tool itself and the complexity of the installation process ought to be considered.

Secondly, the effort and time spent to configure the tool should be given some attention as well. This can be split in two aspects: the configuration absolutely needed to get results from the tool – the less the better; and the configuration options to optimize the quality, readability or usability of the results – the more the better.

Next, the information given by the tool should be easy to understand, useful and trustworthy. In order to be of any use, the reports must be self-explanatory at first sight, yet be complete enough so that they can lead to good understanding of the vulnerability reported and, eventually, to getting it fixed. The utility of the reports also depends on the trust the users have in them. If a tool were to report thousand warnings for each real threat, the effort of sorting through the *false positives* would eventually discourage developers from using it.

There are two other obvious items on the list of things to consider, although in this context they are of lesser importance than the elements presented above. Clearly, it would be optimal to leave as few errors as possible unhandled, so the ratio of *false negatives* should be taken into account.

However, this is clearly a difficult and time-consuming measurement to take, quite beyond the time that was available for this project. Lastly, if commercial products are considered, it implies that pricing will come into play.

3.2 Metrics Used

Table 3.1 shows the criteria used to evaluate the tools, along with the importance each criterion was given and the "grades" that were attributed to the tools.

This was intended to be used as a first filter to determine which tools should be investigated further. As it turns out, the results obtained using this simpler set of metrics were clear enough to allow for a choice to be made without needing to go back to the original set of metrics.

Title	**Weight**	**Values**
Installation	High	Eliminating, Cumbersome, Average, Easy
Configuration	Low	[N/A], Complex, Standard, Not required
Support	Medium	[N/A], None, Slow replies, Helpful
Reports	High	[N/A], Unparsable, Parsable, Readable
Errors found	High	[N/A], None, Common, Uncommon
Handles projects?	Medium	[N/A], Single Files, Import, Yes

Table 3.1: The Metrics Table.

3.2.1 Installation

This metric encompasses the different aspects of a tool's installation process. If a tool is too complex to install, or has too many prerequisites, then developers will have a negative first impression of it and are likely not to be willing to put more effort in getting it set up. While the installation process can be simplified to some extent by packaging, compilation errors and numerous prerequisites are much more complex to compensate.

The tools that would not compile, despite attempts to fix them, as well as the tools having too heavy or numerous prerequisites have been discarded without further investigation. In those cases, installation was declared "Eliminating" and all further metrics were labeled "[N/A]".

If tools had their own custom installation process, they would have been given a "Cumbersome" grade. However, no such tool was evaluated.

For some of the tools, the installation process required previous knowledge of a specific, yet standard, mechanism. One such example is the common series of commands: `./configure`, `make` and `make install`. Tools falling into this category where graded "Average".

Ultimately, some tools make things even easier on their users and provide an installer file or package that takes care of everything. This includes `.exe` files for Windows, and RPMs or `.deb` files for Linux. This was deemed "Easy" installation.

3.2.2 Configuration

Here the complexity of configuring the tool before running it is measured. This is considered as mildly important, as long as a default configuration can be provided with the tool. This differs from the configuration considerations about model checkers or some of the codeflow analysis tools in the sense that for those tools configuration needs to be altered in significant ways from one project to another, and thus providing a customized configuration is not sufficient to make the tool easy to use.

If the only way to configure a tool was through command-line options, or if the configuration could not be easily propagated or shared with developers once it had been created, it was classified as "Complex". One specific case of "Complex" configuration encountered during the evaluation appeared when tools would crash when run on their own, albeit running successfully when wrapped with `valgrind`. Usually used for debugging purposes, `valgrind` works very similarly to a virtual machine, inserting itself between the original program and the host machine. This is originally intended to examine and analyze programs memory calls, but in this context its most interesting feature was its graceful handling of segmentation faults, allowing the programs to continue their execution and issuing a warning instead of merely halting the execution. In those cases, the more specific "valgrind" designation was used.

On the other hand, if the configuration was easy to share with the developers, *e.g.* as a configuration file, it was marked as "Standard".

For some tools however, no configuration was required to get a list of security risks for a given program. This was awarded a "Not required" mention.

3.2.3 Support

While in most cases there was no need for it, support teams were contacted for the tools that were troublesome to install. This is not critical metric, but it was envisioned as giving a second chance to the tools that failed to install in the testing environment but could lead to interesting results with additional effort from the tester.

If the contacted support teams failed to answer, "None" was used.

When replies took longer than a month to obtain, the tool was given a "Slow replies" for this metric.

Lastly, the tools for which replies were quick and complete enough were marked "Helpful".

3.2.4 Reports

This next metric is concerned with the quality of the reports given by the tools. This measures how easy it would be to offer programmers understandable results.

Those tools that would have required unreasonable effort in packaging to convert their reports into something that could be read and understood directly by developers would have been evaluated

as “Unparsable”.

On the other hand, the tools whose output was reasonably easy to convert into a clear and legible format were marked as “Parsable”.

The tools that had readable and informative output out of the box were said to be “Readable”.

3.2.5 Errors Found

Here the range of errors found is evaluated. The focus of this project is on security related flaws and this is what this metric determines.

If the focus of the tool was on coding style and best practices and no verification related to security is performed, the result was “None”.

The tools that had fairly few checks related to security and only pointed to errors most of the other tools found as well were marked as “Common”.

The few tools that did find errors missed by most of the others were listed as finding “Uncommon” errors.

3.2.6 Handles Projects?

This metric determines whether projects are handled by the tool or not. This is relevant for two reasons. First and more importantly, if a tool does not recognize the interactions between source files, it is unlikely to understand anything but the most simple errors. Secondly, if a project consists of thousands of source files, it would be quite cumbersome to have to run the tool manually on each of those files. The latter is not that big an issue, as it can be solved with simple packaging of the tool.

If the tool only analyzes files one at a time, this metric will be set to “Single Files”.

Tools that only accept single files as arguments but try to import the files referenced in the source will get an “Import” mention.

A full blown “Yes” will reward the tools which can be given a source directory as argument and then figure out on their own which files they should process.

4 Presenting the Results

All of the evaluation runs were conducted on a dual-core Intel Pentium IV @ 3.00GHz, running Scientific Linux CERN 5, which is based on Red Hat Enterprise Linux and uses Linux kernel 2.6.18. The source code analyzed either came from CERN **SVN** and **CVS** repositories, was directly shared by groups willing to have their code reviewed or was retrieved from **AFS** hosted Web sites. As some of the projects analyzed are in production and not all the flaws found have been fixed yet, I will not disclose detailed bug reports herein, but rather mention which categories of flaws each of the tools was able to find. Detailed reports have been sent to and discussed with some of the people responsible for maintaining the projects or Web sites. Also, the source code base was so large – over a hundred millions of lines of code – that only a small subset of the hits reported by the tools have been confirmed or invalidated "by hand".

This chapter will present the results obtained with the tools evaluated. While some of the tools yielded interesting results, others had to be abandoned for various reasons, *e.g.* not installing properly. As the categories of flaws found varied slightly from one language to the other and the aim was to select the most suitable tools per language, the evaluations will be classified by language. Some of the tools provide support for various languages and will therefore appear multiple times, once for each of the languages they were tested for. The tools are ordered in alphabetical order within each language group. Each language section contains a table summarizing the results and a detailed evaluation description for each of the tools. The tools that have been selected to be documented and suggested to CERN developers are highlighted with a light grey background in the table.

4.1 C(++) Source Code Analysis Tools

Where security is concerned, C can be a tricky language. Indeed, it leaves much freedom to the developers, notably concerning memory management. This much freedom, along with some inherently unsafe library functions (such as `gets`), means that the burden of ensuring security is made heavier on a programmer. The most common vulnerabilities in `C` code are also the easiest to find and fix.

Many of the unsafe or commonly misused library functions have a "secure" counterpart which too many developers are unaware of. The fact that some of those errors are so common makes them easy to search for and they formed a large percentage of the errors found by the tools tested in this project.

Table 4.1 shows the results of the evaluation of tools for C and C++. They are explained in detail below.

Tool Name	Installation	Configuration	Support	Reports	Errors found	Project support
Astree	N/A	N/A	N/A	N/A	N/A	N/A
BOON	Eliminating	N/A	None	N/A	N/A	N/A
CCA	Eliminating	N/A	None	N/A	N/A	N/A
HP Code A	Eliminating	N/A	N/A	N/A	N/A	N/A
cppcheck	Average	Standard	N/A	Readable	Common	N/A
CQual	Average	Not required	N/A	Parsable	Common	Single files
Csur	N/A	N/A	N/A	N/A	N/A	N/A
Flawfinder	Easy	Not required	N/A	Readable	Uncommon	Yes
ITS4	Easy	valgrind	N/A	Readable	Uncommon	Single files
Smatch	Average	valgrind	N/A	Parsable	Common	Import
Splint	Easy	Complex	N/A	N/A	N/A	Import
RATS	Easy	Not required	N/A	Readable	Uncommon	Yes

Table 4.1: Evaluation Results For C/C++ Tools

4.1.1 Astree

Tool Name	Installation	Configuration	Support	Reports	Errors found	Project support
Astree	N/A	N/A	N/A	N/A	N/A	N/A

Astree is a static analysis tools, which originated from the Ecole Normale Supérieure and the Centre de Recherches Scientifiques. Since December 2009 it is commercialized by AbsInt (`http://www.absint.de/astree`). Unfortunately no public or demonstration version was available during the evaluation phase of this project. An evaluation version was released on January 15th 2010, which includes various checks for `C`. It uses abstract interpretation to find possible run-time exceptions. It currently supports `C` according to the `C99` standard, with the exception of dynamic memory allocation and recursions. While it is regrettable that this late release prevented the evaluation of this tool, those limitations would be prohibitive for most of CERN projects. However, earlier versions of this tool have been successfully used by Airbus and the European Space Agency to prove the absence of run-time exceptions in critical systems.

While this tool sounds promising and interesting for targeted analysis of very specific pieces of code, it is oriented towards the Software Analysts trying to prove a set of given properties and does not fit the needs of the target audience defined in Section 1.2. Therefore, it does not meet the requirements defined for this project.

4.1.2 BOON

Tool Name	Installation	Configuration	Support	Reports	Errors found	Project support
BOON	Eliminating	N/A	None	N/A	N/A	N/A

BOON stands for Buffer Overflow DetectiON. It has been developed at Berkeley and is available from `http://www.cs.berkeley.edu/~daw/boon`. It seems to have many prerequisites not directly available at CERN, which meant a lengthy and difficult to package installation process. Furthermore, the tool comes without any support and is no longer maintained. It was thus discarded, because CERN developers cannot be expected to invest this kind of time and effort into Computer Security.

4.1.3 C Code Analyzer

Tool Name	Installation	Configuration	Support	Reports	Errors found	Project support
CCA	Eliminating	N/A	None	N/A	N/A	N/A

C Code Analyzer is written in `C`, `Perl` and `OCAML`. It can be downloaded from `http://www.drugphish.ch/~jonny/cca.html`. The webpage claims that C Code Analyzer can fully track user input and detect quite a few other errors. However and despite several attempts and meddling into the Makefile, I was unable to get it to compile on any of the machines available to me. This tool does not seem to be supported any longer and no update has been made to the webpage in four years.

4.1.4 Code Advisor (HP)

Tool Name	Installation	Configuration	Support	Reports	Errors found	Project support
HP Code A	Eliminating	N/A	N/A	N/A	N/A	N/A

Hewlett Packard's Code Advisor takes advantage of its close relationship with the HP C/C++ Compiler to maximize its understanding of the source code analyzed. It uses a client / server architecture, and while clients are available for most platforms, the server can only run on an HP-UX system. This made evaluating the tool impossible in our setting and would not be suitable for CERN programmers.

4.1.5 Cppcheck

Tool Name	Installation	Configuration	Support	Reports	Errors found	Project support
cppcheck	Average	Standard	N/A	readable	Common	N/A

Cppcheck is a community developed and maintained static analysis tool, available from Source-Forge at `http://sourceforge.net/apps/mediawiki/cppcheck/index.php?title=Main_Page`. It

is written in C++ and is available as a `msi` file for Windows users. Its source code is available on a public git repository.

Compilation and installation worked flawlessly. It can be run either from the command-line or through a GUI, which is quite intuitive to understand. Runs on various C/C++ projects yielded a fairly comprehensive subset of the checks listed at `http://sourceforge.net/apps/mediawiki/cppcheck/index.php?title=Main_Page#Checks`. In particular some memory leaks were detected as well as improperly closed file handles.

4.1.6 CQual

Tool Name	Installation	Configuration	Support	Reports	Errors found	Project support
CQual	Average	Not required	N/A	parsable	Common	Single files

CQual uses type qualifiers to analyze source code. It takes advantage of type inference techniques to limit the number of annotations required from the programmer.

It is available from `http://www.cs.umd.edu/~jfoster/cqual/` and it installed flawlessly by running standard Unix installation commands. Its command-line interface is self-explanatory, and the default configuration yields some quick results. However, the flaws found without requiring additional effort from the user are quite limited. Actually, unless it is provided with additional information about the program architecture and the constraints that should be enforced, there is little that type inference can achieve. On the other hand, flaws such as vulnerabilities to string formatting attacks can be detected, for instance by marking all user input with a “tainted” type qualifier and requiring the first argument of all functions in the `printf` family to be “untainted” [19]. While a few other possible usages have been researched by Wagner [9], Foster et al. [13] and Foster [12] amongst others, those techniques still require a good understanding of the type inference mechanisms used by CQual in order to place the proper type qualifier annotations at the right place in the source code.

4.1.7 Csur

Tool Name	Installation	Configuration	Support	Reports	Errors found	Project support
Csur	N/A	N/A	N/A	N/A	N/A	N/A

Csur is developed under copyright license and a beta version is announced for the near future since 2004. It could therefore not be evaluated.

4.1.8 Flawfinder

Tool Name	Installation	Configuration	Support	Reports	Errors found	Project support
Flawfinder	Easy	Not required	N/A	readable	Uncommon	Yes

Flawfinder is a security oriented static analyzer for `C/C++`, which was specifically developed by its author to be easy to install and to use. It uses text-based pattern matching to detect calls to inherently vulnerable or commonly misused library functions.

It is available as a source rpm for Fedora based Unix systems and only requires Python. It is therefore easy to run from source on Windows, *e.g.* using Cygwin. When invoked directly on a source file or directory, it will report a warning for each call to functions it considers as a potential risk. The text output is easy to understand and includes a short description of the reason for which the function is considered insecure and in some cases suggests a possible fix. The reported hits are given a threat level, ranging from 0 (benign) to 5 (high risk). Flawfinder will also print a short summary of the analysis, displaying the number of lines of code analyzed and the number of hits reported for each threat level. Command-line arguments include the ability to set the minimal threat level reported, directives to output the report in html or xml and "header-/ footerless, single ligne per error" output, which is useful for editor integration.

While Flawfinder does yield some false positives – after all, not every single call to `strcpy` results in a security vulnerability – it also comes with a mechanism to report false positives, by including a comment on the falsely accused line. It will then ignore the error(s) found on that line, unless the `--never-ignore` option is used.

It was decided to keep this tool because of its noteworthy ease of use and distinct ability to find a wide range of simple, yet common, errors. It also has a few options to manage and customize the output.

4.1.9 ITS4

Tool Name	Installation	Configuration	Support	Reports	Errors found	Project support
ITS4	Easy	valgrind	N/A	readable	Uncommon	Single files

ITS4 was developed and maintained by Cigital, but has been discontinued since 2000. It uses an approach fairly similar to that of Flawfinder, in the sense that it also uses text-based pattern matching to detect potential security-related issues.

Installation and configuration did not cause any problem and could be achieved with standard Unix installation commands. However, the first run attempted lead to a segmentation fault. Running ITS4 through valgrind to attempt and pinpoint the error actually seemed to circumvent the segmentation fault.

Once this error was bypassed, the results obtained were similar to those reported by Flawfinder.

4.1.10 Smatch

Tool Name	Installation	Configuration	Support	Reports	Errors found	Project support
Smatch	Average	valgrind	N/A	parsable	Common	Import

Smatch is a project led by the people at `http://janitor.kernelnewbies.org`. It is uses the sparse C parser and is mostly oriented at finding bugs in the Linux kernel. The source can be obtained on the git repository `http://repo.or.cz/w/smatch.git`. Building the binaries is simple enough, but the installed version crashed on various of the files it was given, and similarly to ITS4, when run inside valgrind, the problem would disappear. Since testing it for this project, a new version has been released, but according to the author himself, it is still buggy and not quite as reliable as he would like it to be.

While it has some detectors for buffer overflows, it did not detect them in the projects it was tested on, and the only relevant issue it reported was an array out of bounds error.

4.1.11 Splint

Tool Name	Installation	Configuration	Support	Reports	Errors found	Project support
Splint	Easy	Complex	N/A	N/A	N/A	Import

Splint is a variant of lint, developed at the University of Virginia, concerned mainly with Secure Programming. Installation is achieved through standard Unix commands.

It can be used with relatively little effort to perform basic checks, but the more advanced functionalities and verifications require the developers to include annotations in their code. This is fine for projects that will be verified multiple times in the long run, and for which the developers can afford to spend more time initially to get the testing parameters setup. However it does not suit the unconvinced developer who wants to just give a try to a Source Code Analysis Tool. Also, Splint only accepts pure C code and cannot handle C++.

4.1.12 RATS

Tool Name	Installation	Configuration	Support	Reports	Errors found	Project support
RATS	Easy	Not required	N/A	readable	Uncommon	Yes

RATS stands for Rough Auditing Tool for Security. It is an open source tool distributed under GPL, which can scan C, C++, Perl, PHP and Python source code. It uses text-based pattern matching to look for potential risks, based on the entries in its language-specific dictionaries. It can be freely downloaded from `http://www.fortify.com/security-resources/rats.jsp`.

A Windows executable and a source tarball are available from the site mentioned above, and packaged binaries are available for many Linux distributions. It can be run directly on the root directory of a project and will analyze the files written in each of its target languages, based on the file extension. Its C/C++ dictionary of potentially vulnerable patterns has 334 entries. The results of an analysis are similar to Flawfinder's, with the advantage of being faster and the limitation of missing the source files with unknown extensions, which Flawfinder recognizes using heuristics.

It was decided to keep this tool because of its ease of use, its speed and the interesting, albeit not impressive, quality of its results. Furthermore, it has the advantage of being easily extensible.

4.2 Java Source Code Analysis Tools

Aside from the vulnerabilities common to all the languages considered, *i.e* errors related to file handling and system calls, Java code is also susceptible to leak improperly secured confidential information.

Table 4.2 summarizes the results of the evaluation of Source Code Analysis Tools for Java.

Tool Name	Installation	Configuration	Support	Reports	Errors found	Project support
CodePro	Easy	not required	N/A	Readable	Uncommon	Yes
FindBugs	Easy	Standard	N/A	Readable	Uncommon	Yes
Hammurapi	Standard	Complex	N/A	Readable	Common	Yes
JCSC	Easy	Complex	N/A	Parsable	Common	Single Files
IBM	Complex	Standard	Helpful	Readable	Uncommon	Yes
PMD	Standard	not required	N/A	Readable	Common	Yes
QJPro	Easy	Standard	N/A	N/A	N/A	Yes

Table 4.2: Evaluation Results For Java Tools

4.2.1 CodePro Analytix

Tool Name	Installation	Configuration	Support	Reports	Errors found	Project support
CodePro	Easy	not required	N/A	Readable	Uncommon	Yes

CodePro Analytix is a commercial static analysis tool for Java, developed by Instantiations. It is available as an Eclipse plugin from `http://www.instantiations.com/codepro/analytix`. Installation is performed through the Eclipse plugin manager, from a local directory where the software must be downloaded first.

CodePro Analytix integrates with Eclipse and all the functionalities are available from the IDE. The list of bugs found was fairly comprehensive and can be expanded through activation of rules not figuring in the default analysis configuration, either individually or by categories. The "Security" category includes detectors for faulty authentication, issues with temporary files, missing catches for exceptions (particularly relevant for Web applications) and usage of tainted input values, amongst others. Each reported error comes with an explanation of its security implications.

Unfortunately, a major investment would have been necessary to serve CERN's 200 Java developers. However, a link to this tool has been provided to CERN developers, so that any group that would be willing to invest more in software security would have the necessary information.

4.2.2 FindBugs

Tool Name	Installation	Configuration	Support	Reports	Errors found	Project support
FindBugs	Easy	Standard	N/A	Readable	Uncommon	Yes

FindBugs is free software, distributed by the University of Maryland under the LGPL terms. It is available from `http://findbugs.sourceforge.net`. It can be installed either as a plugin to Eclipse (from version 3.3 onwards) or as a standalone application with a Swing interface. The Eclipse installation is handled by the Eclipse plugin manager. The standalone application contains a jar file with launchers both for Linux and Windows.

There is a slight subtlety in the configuration of the standalone application, which needs to be given the path to your compiled classes, libraries and source files – the latter is optional but allows for viewing the error in context. The Eclipse plugin can be run directly from the IDE. Similarly to Code-Pro Analytix, FindBugs features an extensible dictionary of vulnerabilities, grouped in categories. By default, the "Security" category is not activated, nor is the "Malicious Code Vulnerability". This can be changed easily though – detailed explanations are available in Appendix A.2. Bugs found include vulnerabilities to SQL injections or Cross-Site Scripting, ignored exceptional return values, exposed internal representations and more.

Reports can be exported to XML files, including classification of individual hits as "not a bug", "I will fix", . . .

4.2.3 Hammurapi

Tool Name	Installation	Configuration	Support	Reports	Errors found	Project support
Hammurapi	Standard	Complex	N/A	Readable	Common	Yes

Hammurapi is a platform to review code quality. It has seen a major shift in focus between version 5.7 and 6.1.0. Version 5.7 is available at `http://www.hammurapi.biz/hammurapi-biz/ef/xmenu/hammurapi-group/products/hammurapi/index.html` and is usable out-of-the box to analyze Java source code. Version 6.1.0 is available at `http://www.hammurapi.com/` and is more of a framework rather than a tool per se. It provides interfaces for language modules to parse source code into a model, for inspector sets to review the model and produces observations, and for waivers to render the observations in a user readable format. The version evaluated in this project was version 5.7.

The installer is available as an executable file for Windows and as a jar file for cross-platform compatibility. It can then be used as an Eclipse plugin.

Many of the major security flaws identified by other tools were missed by Hammurapi, which seems to focus on coding style practices and "code smell".

4.2.4 JCSC

Tool Name	Installation	Configuration	Support	Reports	Errors found	Project support
JCSC	Easy	Complex	N/A	Parsable	Common	Single Files

JCSC is a free tool available from `http://jcsc.sourceforge.net/`. It is available as a standalone application with a Swing GUI, or as a plugin for IntelliJ. It is inspired by lint and mostly checks for compliance to a coding style. The default configuration is in line with the Sun Code Conventions, and has a couple of additional checks for potentially dangerous code, such as empty catch blocks. It also comes with a rule editor, which can be used to implement detection of custom patterns.

4.2.5 IBM Rational AppScan

Tool Name	Installation	Configuration	Support	Reports	Errors found	Project support
IBM	Complex	Standard	Helpful	Readable	Uncommon	Yes

AppScan is the static analysis tool of IBM's Rational line. It is commercially distributed at `http://www-01.ibm.com/software/awdtools/appscan/`.

Installation is performed by ad-hoc binaries, which failed to work on the Linux machine used for the evaluation, because they need to set up a specific user to run the background services upon which the analyzer is built, using a command not compatible with CERN's SLC commands to create users. However, installation could be completed on a Windows platform. It required the installation of three different binaries and the obtention and configuration of a license key for each of them. Furthermore, the tool installs to a directory that must then be set as an external Eclipse plug-in location (which cannot be done anymore from Eclipse 3.4 onwards). Fortunately, support from IBM was outstanding.

While this tool did find a few more bugs than others of the tools we analyzed, it was not trivial to sort those bugs according to some sort of risk rating, which would have made it easier for an average CERN programmer to find out what he should focus on.

The bottom line is that the slightly increased complexity in installing / managing / running the tool is currently not compensated by some extra features in comparison to other tools.

However, this is a very interesting tool and should be recommended to anyone willing to go the extra mile and on a budget that can afford it.

4.2.6 PMD

Tool Name	Installation	Configuration	Support	Reports	Errors found	Project support
PMD	Standard	not required	N/A	Readable	Common	Yes

PMD is a fairly popular free Java source code scanner. It is available from http://pmd.sourceforge.net and integrates with most major IDEs. Installation can be handled directly through the plugin manager of major IDEs or downloaded from the tool's homepage.

No additional configuration is required to run the tool, but the reported flaws are mostly related to usual bad coding practices and not directly related to software security.

4.2.7 QJPro

Tool Name	Installation	Configuration	Support	Reports	Errors found	Project support
QJPro	Easy	Standard	N/A	N/A	N/A	Yes

QJPro is a free tool for Java Source Code Analysis and coding standards enforcement. It is available from http://qjpro.sourceforge.net either as a standalone application or as a plugin for Eclipse or other IDEs.

However, its latest version predates Generic Java and code written for Java 1.5 or more recent will cause it to crash.

4.3 Perl Source Code Analysis Tools

Tool Name	Installation	Configuration	Support	Reports	Errors found	Project support
B::Lint	Standard	Complex	N/A	Readable	Common	Yes
Perl::Critic	Easy	Standard	N/A	Readable	Uncommon	Yes
RATS	Easy	Not required	N/A	Readable	Common	Yes
Taint Mode	Easy	Not required	N/A	Readable	Uncommon	Import

Table 4.3: Evaluation Results For Perl Tools

4.3.1 B::Lint

Tool Name	Installation	Configuration	Support	Reports	Errors found	Project support
B::Lint	Standard	Complex	N/A	Readable	Common	Yes

B::Lint is a Perl module available on CPAN at http://search.cpan.org/~jjore/B-Lint-1.11/lib/B/Lint.pm. Installation can be carried out through a CPAN shell.

B::Lint can be extended to find more errors, but it is only shipped with some fairly simple detectors. The errors it finds are some of the most common ones, but are fairly limited.

4.3.2 Perl::Critic

Tool Name	Installation	Configuration	Support	Reports	Errors found	Project support
Perl::Critic	Easy	Standard	N/A	Readable	Uncommon	Yes

Perl::Critic is a Perl module that can be used to enforce over 100 policies, most of which come from Damian Conway's Perl Best Practices [8]. It is available on CPAN at `http://search.cpan.org/~elliotjs/Perl-Critic-1.105/lib/Perl/Critic.pm`. Installation is done via a CPAN shell. The installation includes an executable, "`perlcritic`", on top of the Perl::Critic module.

While some of the security-related checks are disabled by default, if the level of verification is set to "brutal", most of the risky Perl idioms will be identified. This setting will also produce many coding style warnings though, thus increasing the length of the output and making it more difficult for the developers to identify the most crucial warnings.

4.3.3 RATS

Tool Name	Installation	Configuration	Support	Reports	Errors found	Project support
RATS	Easy	Not required	N/A	Readable	Common	Yes

General information about RATS is available on Section 4.1.12. RATS has 33 entries in its Perl database. It might trigger more false positives for Perl than for other languages, as some secure libraries use the same function names as the insecure libraries they intend to replace.

While RATS only really shines for C/C++ code analysis, it is still a very quick and efficient tool for Perl and can be recommended to programmers wishing to use a single tool for a variety of languages.

4.3.4 Taint Mode

Tool Name	Installation	Configuration	Support	Reports	Errors found	Project support
Taint Mode	Easy	Not required	N/A	Readable	Uncommon	Import

While Perl's Taint Mode is not a source code analysis tool strictly speaking, it is a key language feature as far as security is concerned. When taint mode is activated, through the `-T` flag, every data that comes from outside the program itself, including configuration files, user input and environment variables, will be marked as tainted until it is sanitized. Sanitization is done through pattern matching. Taint mode will not ensure that the values used are secure, only that they have been verified, thus it is still up to the programmer to make sure that those values do not contain possibly harmful data.

This does require some additional effort from the developer, but its benefits are not to be neglected.

4.4 PHP Source Code Analysis Tools

There are not that many source code analysis tools available for `PHP`. It seems more effort has been put into "black-box" approaches for Web applications, probably because it puts the analyzer in the

position of an attacker, thus detecting the same kind of bugs a malicious user is likely to find and exploit. However, it is always possible for an attacker to find a weakness a vulnerability scanner missed. Therefore, if some bugs can be caught easily using a “white-box” approach, there is no reason not to try it!

Tool Name	Installation	Configuration	Support	Reports	Errors found	Project support
Sandcat	Easy	Not required	N/A	Readable	Uncommon	Yes
Pixy	Standard	Not required	N/A	Parsable	Uncommon	Yes
RATS	Easy	Not required	N/A	Readable	Common	Yes

Table 4.4: Evaluation Results For PHP Tools

4.4.1 Sandcat.4PHP

Tool Name	Installation	Configuration	Support	Reports	Errors found	Project support
Sandcat	Easy	Not required	N/A	readable	uncommon	Yes

Sandcat.4PHP is developed and commercialized by Syhunt, as an extension to their “black-box” testing software, Sandcat. It is available in a rather limited demonstration version from Syhunt’s Web site at `http://www.syhunt.com/?section=sandcat4php`. This tool is available only on Windows.

While it does yield some interesting results on simple test cases, the demonstration version is limited to two source files, thus restricting the usefulness of the tests. At the time of the evaluation phase of the project, its professional version was publicly priced 1,099 USD per seat. As explained in Section 1.2, any CERN user is potentially a Web site owner/developer/maintainer. Therefore, this pricing model would be totally inadequate for CERN.

It seems however that the distribution options have changed in the meantime. Sandcat.4PHP is now exclusively available in the Hybrid version of the Sandcat software, for which price quotations are offered on demand.

The pricing of the tool at the time along with the availability being limited to Windows were considered severely hindering factors.

4.4.2 Pixy

Tool Name	Installation	Configuration	Support	Reports	Errors found	Project support
Pixy	Standard	Not required	N/A	Parsable	uncommon	Yes

Developed in Java by people from the International Secure Systems Lab (`http://www.iseclab.org/`), Pixy scans `PHP` files looking specifically for Cross-Site Scripting and SQL-Injection vulnerabilities, two of the most commonly exploited weaknesses in Web applications according to OWASP’s 2005 Top 5 PHP Vulnerabilities list[1].

[1] See `http://www.owasp.org/index.php/PHP_Top_5`.

Pixy's latest version, dating back to July 2007, is available from `http://pixybox.seclab.tuwien.ac.at/`. As it is based on `Java` and packages both a `Perl` script and a batch file to set the environment variables properly and invoke the Java application, Pixy can easily be run on Linux or Windows. The test runs led to the discovery of various vulnerabilities in CERN user Web sites, which have since then been verified and fixed by the Web site owners. To facilitate the identification and correction of the vulnerabilities it reports, Pixy outputs call graphs in the `DOT` language. An example of a such graph is included in Appendix B.

The main drawbacks of Pixy are that the graphs are created in text format, which would require additional effort from the developers to understand or to view, and the fact that it only accepts single files as input. Another issue is the the reports seem to have been designed to be parsed rather than read.

4.4.3 RATS

Tool Name	Installation	Configuration	Support	Reports	Errors found	Project support
RATS	Easy	Not required	N/A	Readable	Common	Yes

General information about RATS is available on Section 4.1.12. RATS has 55 entries in its PHP database, many of which are concerned with (possibly) improper retrieval of user input.

However, this does not include the commonly used `$_GET[key]` shorthand notation, only the functions used to achieve the same effect.

While RATS only really shines for C/C++ code analysis, it is still a very quick and efficient tool for PHP and can be recommended to programmers wishing to use a single tool for a variety of languages.

4.5 Python Source Code Analysis Tools

It seems that the number of source code analysis tools for Python is rather limited. There are a few possible explanations for this, some of which will be presented below. First and foremost, the developer community is quite proactive about pushing security patches for Python, thus limiting exposure of up-to-date systems. However, while Python does not have any exploitable library function as well-known as `C`'s `gets`, some of its functions can be used incorrectly by developers, *e.g.* not properly sanitizing user input before using it in a system call. Another possible explanation being that `C` and `Java` have been used in security-critical software for decades, and `PHP` and `Perl` are very commonly used in the development of Web applications, which are very exposed to large scale attacks. This would explain more effort and attention being put into developing source code analysis tools for those languages, or vulnerability scanners in the case of `PHP`.

Tool Name	Installation	Configuration	Support	Reports	Errors found	Project support
PEP8	Standard	Not required	N/A	Readable	None	Yes
PyChecker	Standard	Not required	N/A	Readable	Common	Yes
Pylint	Standard	Not required	N/A	Readable	None	Yes
RATS	Easy	Not required	N/A	Readable	Common	Yes

Table 4.5: Evaluation Results For Python Tools

4.5.1 PEP8

Tool Name	Installation	Configuration	Support	Reports	Errors found	Project support
PEP8	Standard	Not required	N/A	Readable	None	Yes

PEP8 is a community developed source analysis tool for Python. Its source is available on a read-only git repository: `git://github.com/jcrocholl/pep8.git`. It checks for compliance to some of the style conventions in PEP8 [2].

PEP8 is written in Python and can thus be run directly from source on any Operating System for which a Python interpreter is available.

The absence of any security-related checks is clearly an issue in this context.

4.5.2 PyChecker

Tool Name	Installation	Configuration	Support	Reports	Errors found	Project support
PyChecker	Standard	Not required	N/A	Readable	Common	Yes

PyChecker is a community driven source code analysis tool for Python. While PyChecker mainly targets "problems that are typically caught by a compiler for less dynamic languages", it also includes a couple of checks related to security. Its source code is available from SourceForge, at `http://sourceforge.net/projects/pychecker`. On Linux, it can be installed by running `Python setup.py` directly from the source directory. As it is written in Python though, it is easy to make it run on a Windows installation with a Python interpreter. Running PyChecker against sample code permitted the identification of various errors, most of which were not security risks, but included a vulnerability to arbitrary code execution..

4.5.3 Pylint

Tool Name	Installation	Configuration	Support	Reports	Errors found	Project support
Pylint	Standard	Not required	N/A	Readable	None	Yes

Pylint is an open source static source code analysis tool for Python. It is available as a source tarball from `http://www.logilab.org/857`.

[2]See `http://www.Python.org/dev/peps/pep-0008/`.

The installation process is similar to PyChecker's. However, it does not include any security related checks and was therefore not considered any further for this project, despite being a quite popular tool.

4.5.4 RATS

Tool Name	Installation	Configuration	Support	Reports	Errors found	Project support
RATS	Easy	Not required	N/A	Readable	Common	Yes

General information about RATS is available on Section 4.1.12. RATS has 62 entries in its Python database, many of which are concerned with possible race conditions. It also checks for insecure random number generators and system calls. Not all dangerous system calls are detected though, and calls to `commands.getstatusoutput`, while highly risky, are missed.

While RATS only really shines for C/C++ code analysis, it is still a very quick and efficient tool for Python and can be recommended to programmers wishing to use a single tool for a variety of languages.

5 Delivering the Results

The end goal of this project was to increase computer security at CERN. In order to ensure a real usefulness of the results presented so far, it was important to propagate information about static analysis tools to CERN developers. Due to the organization size and the limited human resources available to the Computer Security Team, dedicated communication channels are needed. In order to maximize the visibility and usage of this project, a combination of means have been used. The first phase consisted of the creation of a reference Web site, containing information about the selected tools as well as recommendations for the creation of secure source code, both language specific and with a broader impact. This ensured information availability and persistence.

A talk to which developers and group leaders were invited and an article written for an internal publication contributed to increase the visibility of the results, which was the objective of the second step.

5.1 CERN Computer Security Team Web site

The Web site created for this project is available at `http://cern.ch/security/codetools`. It contains installation instructions, configuration guidelines and sample execution commands for each of the selected tools. These are mostly oriented to Linux users, more specifically to users of the Scientific Linux CERN distribution, since this is the operating system most commonly used by CERN developers. However, hints are also provided for users of other Linux distributions as well as for Windows users, whenever applicable. A snapshot of the status of the documentation is available in Appendix A.

5.1.1 Packaging the Tools

An important part of the process of making the tools available for CERN users was to make their installation as simple as possible. Scientific Linux CERN, the distribution most widely used by CERN developers, is based on Red Hat Enterprise Linux and uses `yum` to manage its packages. Fortunately, most of the tools were already available as source `RPMs`, and required very little effort to

integrate into the CERN software repositories. On the other hand the tool selected for PHP source code analysis, Pixy, requested slightly more effort. Furthermore, the original Pixy implementation only accepts single files as input and cannot process a project globally, although it does analyze all the files included or required by the file it is given. Taking advantage of this feature, a wrapper was developed, which will create a temporary PHP file, consisting of a call to the `include` built-in for each of the PHP files found in the directory given as argument. Also, in its original form, Pixy's output can be rather cumbersome to understand for the naked eye. The result is postprocessed by the wrapper and can be printed in straight text form, in an XML structure or as HTML code. Last but not least, Pixy creates digraphs providing insight into each of the errors it reports. One such graph can be found in Appendix B. However, those digraphs are created in DOT (text) format. The wrapper uses GraphViz to convert those digraphs into PNG files, which can then be directly viewed and are linked from the HTML-formatted reports.

With the same idea of making the whole process as simple as possible for the users and allowing them to obtain a quick hands-on experience with the tools, a configuration file for Perl::Critic has been provided. It creates two custom groups of errors, the first one – `cerncert` – featuring all the detectors identified as directly concerned with security while the second set identifies bad-practices that can lead to vulnerabilities on the longer run, by making the code more difficult to maintain. These rules have been identified and categorized in collaboration with members of the Computer Security Team.

5.2 Post-C5 Presentation / IT Seminar

The C5 is a weekly meeting of managers of main services offered within CERN's IT Department. It is attended by representatives of all the IT groups. Sometimes those meetings are followed by a presentation, where a member of the Department has the opportunity of speaking about a project of general interest. Another semi-regular set of talks within the IT Department is a series of computing seminars, often featuring external speakers, which all are welcome to attend.

In order to maximize the impact of the talk, it has been organized as a joint venture and advertised as both a Post-C5 Presentation and an IT Seminar. As a result, more than a hundred people attended the talk and not everyone was able to take a seat. The talk was built around three key points: motivate the attendees to use the tools; show how easy it is to run some of the tools and to get some results; insist that while Source Code Analysis Tools are a nice way to get *some* results quickly, using them will not guarantee security.

The slides used are available at `http://documents.epfl.ch/users/t/th/thhofer/public/C5-CS-presentation.pdf`.

5.3 Article in the CNL

The last approach used to propagate the results of this project to CERN developers is the writing of an article in the Computer Newsletter (CNL). The CNL is the internal publication within CERN's IT Department. It is however distributed throughout the organization, and outside it as well, and thus has a wide coverage. It is also available on `http://cern.ch/cnl`. At the time of writing, the January-March 2010 issue – the one featuring the article *Find your security vulnerabilities before attackers do* – has not yet been released.

This article includes a few paragraphs on source code analysis tools, whose aim is to persuade developers to use the tools evaluated, packaged and documented during this project. The author's contribution to the article was the following:

> **Finding your bugs before attackers do...!**
>
> Another important key point to improving computer security at CERN is to make the attackers' job harder by minimizing the exploitable flaws in the software that you develop. While it is very hard to write perfectly secure software, there are some simple means allowing you to improve the security of your piece of software. One of the easiest ways is using Static Source Code Analyzers. Those tools look at your code without executing it, but point out what they consider to be potential weaknesses. The most typical example of what those tools can find probably is calls to the `gets` function in the C programming language: this function is inherently insecure and can lead to buffer overflows. Specially crafted user input values can for instance allow an attacker to access or modify confidential data or even take control of any computer executing that piece of software. Because these tools need to understand your code, they are necessarily very language specific. The Security Team has evaluated a number of such tools, for various programming languages (C/C++, Java, PHP, Perl and Python) and has compiled a short list of analyzers, selected because of their ease of use, their simple configuration and their established benefit in pointing out weaknesses.
>
> This list of tools is available at http://cern.ch/security/CodeTools, along with advice on configuration and recommendations on how to fix the most common errors, as well as pointers to Web sites and books containing more information on the matter. RATS, shorthand for Rough Auditing Tool for Security, covers all of the above languages, with the exception of Java. It targets mainly calls to commonly misused or exploited library functions. It is a very fast tool, available on Linux and Windows. For C/C++, David Wheeler, a renowned IT security expert, provides Flawfinder, which will look for risks of buffer overflows and race conditions. It is unfortunately only available on Linux. Coverity is a security company with extensive experience in C/C++ static analysis, responsible for finding many bugs in major open source projects such as the Linux kernel or implementations of samba and is contracted by the U.S. Department of Homeland

Security and Yahoo among others. Currently, the PH/SFT group is arranging an agreement with Coverity, which will allow for CERN to use their tool. For Java, FindBugs will find various security-related and non security-related errors, vulnerabilities to SQL injection for instance. It is available both as an Eclipse plug-in and as a stand-alone java application. Pixy will review your PHP code and warn you against risks of SQL injection and Cross-Site Scripting. It consists mostly of Java code with a Perl wrapper. It is therefore cross-platform compatible. The Perl::Critic CPAN module will raise warnings for many risky Perl idioms, and used in conjunction with Perl's tainted mode, it should help to produce secure Perl code. As for Python, the most important thing is simply to keep your version up-to-date with security patches (for SLC machines, the security patches are back-ported to older versions, which is why Python still appears to be at version 2.4!). You can also use rats (see above) to detect some of the potential security issues. Pychecker and pylint are also nice static analysis tools, even if they do not focus on security aspects.

6 Conclusion

6.1 Results

Though the results obtained when analyzing programs with the tools selected during this project are not on level with the state of the art methods for static analysis, they constitute a compromise between completeness of results and investment of resources (*i.e.* time and effort). This implies that they are more likely to be used in the context relevant to this project. In practice, the results seems to have been welcomed by CERN developers, with attendance for the tool presentation talk upwards of one hundred people and approximately one hundred unique visits to the documentation webpage over its first three weeks online.

It is difficult to precisely measure the impact of this research because computer security is not directly quantifiable without proof of absolute security, which is neither currently available nor expected to be in the future. However, careful and diligent use of those tools will help eliminate a large number of common errors and to educate the developers by increasing their awareness to security issues. For instance, the Cross-Site Scripting Vulnerabilities that have been found during this research could have been entirely avoided with little effort.

6.2 Achievements

The most satisfactory aspect of the results of this projects are the fact that the product delivered to the CERN Computer Security Team met the highest expectations that had been set for it. Not only where all of the tools documented and configurations provided where necessary, only one of them could not be made available on CERN software repositories.

The following two items were not stricly related to the evaluation of the tools, but can be viewed as interesting by-products of this research. The most closely related to the topic is the wrapper written for Pixy which is already mentioned in Chapter 5. The second one is a script written to retrieve pages from a wiki over an HTTPS connection to create a snapshot fit for distribution (detailed in Appendix C.2. The development of those (rather small) pieces of software were a good opportunity

to put secure programming guidelines in practice and to use some of the tools recommended on familiar code.

6.3 Main Challenges

The primary challenge faced during this project was social, rather than technical, in nature. The need to adapt to a practical and professional environment with developer deadlines implied finding a balance between the theoretical optimum and the solution most applicable to practice, rather than aiming solely for the optimal solution as a purely academic project would require. In the words of Einstein,

> In theory there is no difference between theory and practice, but in practice there is!

The most significant technical challenge of this project concerned the variety of programming languages targeted. While familiar with some of the languages considered, at the inception of this research I was by no means a security expert for any of them. Therefore analyzing and identifying the various categories of security vulnerabilities for each language was a substantial undertaking.

6.4 Outlook and Future Developments

This section will discuss the possible improvements and extensions to this project. These are grouped in two different subsections. Subsection 6.4.1 will describe what is already being done at CERN as well as some ideas and suggestions that are envisioned for the near future. Subsection 6.4.2 describes more global improvements that could be made to Computer Security in general, based upon the reasoning that Source Code Analysis is not sufficient, at least in its current form.

6.4.1 At CERN

A possible extension for this project consist of running those tools on all of the source code accessible on the various repositories hosted at CERN and discussing the results with the affected developers.

Running those tools without knowing the internal structure of a project can present a few issues. For instance, the tools should not be run against the root of a repository, but rather on branches or on the trunk, or maybe even on a specific source directory. Some of the tools might crash, but will definitely run erroneously, if they are facing different versions of the same file.

Another point one should be very careful with is the interpretation of the results. Large code bases will lead to considerable output, and a basic understanding of the code is required to check whether each of the errors is a false positive or not.

The last thing one should be careful with is in case of multi-language projects. Some of the tools might ignore files based on extensions, while others might misinterpret a file, because it is not in the expected language.

If good care is taken of these points, it would be interesting to have a thorough analysis of the code repositories.

6.4.2 Further Improvements to Software Security

While it is good that such simple means as the tools presented in Chapter 4 help to improve software security, those means are clearly not sufficient to offer any guarantees about software security. There are many types of errors those tools are unable to detect including, but not limited to, flaws in design or failure to properly implement a protocol.

There are a few ways one could go further down the path of creating more secure software. First and foremost, it is vital that developers be made aware of the risks and vulnerabilities inherent to their programming language. There is an appalling number of posts across developer forums asking why compilers throw a warning when the `gets` function is used. This issue can only be resolved through adequate training.

Another important point is the integration of security into the Software Development Life Cycle. While some effort has already been made in that direction – *e.g.* Microsoft uses the **ASAP**[1] process internally and the U.S. Department of Homeland Security hosts an article by N. Davis [10] – it has not quite managed to become standard practice in the industry yet.

An option less demanding on the developers would be to have more thorough tools available than those selected during this project. This is partially achieved already, as some critical pieces of software have been verified with formal methods. One such example is Airbus verifying its A380 flight system using Astre – the results are alluded to in [21].

All of the above suggestions are, however, only applicable if people can be convinced to put time and effort into software security. Dan Lohrmann, Michigan State's CTO, is currently working on a series of articles entitled *Why Do Security Professionals Fail?* As of this writing, the first five articles in this series are available on his CSO Online Blog `http://blogs.csoonline.com/blog/dan_lohrmann`. Out of the various issues he mentions, two seem particularly interesting. Firstly, he mentions that security experts are generally ill-considered, as a consequence of always bringing up issues no one else wants to hear. There is no simple way around this, but paying attention to the ways problems are brought up and trying to suggest a solution might go a long way. Secondly, he encourages to taking a "realistic" approach to security. Instead of trying to push for the pristine, clean and totally secure option, which would entail a radical increase in costs, a few options should be studied and suggested, offering various levels of efficiency and implied costs.

It is often difficult for software engineers to grasp how (in)secure their code is. An option some testing tools or security analysts offer, which seems to be quite popular, is some sort of grading of the source evaluated. This makes it easier for the non-specialist to get an idea of the improvement a patch has brought, and for team leaders to set targets for their teams. However, it is not trivial

[1] Application Software Assurance Program

to create such a metric in a way that makes sense and is coherent across new versions of a piece of software.

Bibliography

[1] Almeida, J. B., Barbosa, M., Sousa Pinto, J., and Vieira, B. Verifying cryptographic software correctness with respect to reference implementations. In *FMICS '09: Proceedings of the 14th International Workshop on Formal Methods for Industrial Critical Systems* (Berlin, Heidelberg, 2009), Springer-Verlag, pp. 37–52.

[2] Alt, M., and Martin, F. Generation of Efficient Interprocedural Analyzers with PAG. In *SAS'95, Static Analysis Symposium* (September 1995), A. Mycroft, Ed., vol. 983 of *Lecture Notes in Computer Science*, Springer, pp. 33–50.

[3] Bagnara, R., Hill, P. M., Pescetti, A., and Zaffanella, E. On the design of generic static analyzers for imperative languages. Quaderno 485, Dipartimento di Matematica, Università di Parma, Italy, 2008. Available at `http://www.cs.unipr.it/Publications/`.

[4] Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., and Engler, D. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM 53*, 2 (2010), 66–75.

[5] Beyer, D., Henzinger, T., and Théoduloz, G. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *CAV: Computer-Aided Verification*, Lecture Notes in Computer Science 4590. Springer, 2007, pp. 509–523.

[6] Clarke, E. M., and Emerson, E. A. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop* (London, UK, 1982), Springer-Verlag, pp. 52–71.

[7] Clarke, E. M., Emerson, E. A., and Sistla, A. P. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst. 8*, 2 (1986), 244–263.

[8] Conway, D. *Perl Best Practices.* O'Reilly Media, Inc., 2005.

[9] David, R. J., and Wagner, D. Finding user/kernel pointer bugs with type inference. In *In Usenix Security Symposium* (2004), pp. 119–134.

[10] Davis, N. Secure software development life cycle processes, 2006.

[11] Emerson, E.and Clarke, E. Characterizing correctness properties of parallel programs using fixpoints. In *Automata, Languages and Programming* (1980), Springer Berlin / Heidelberg, pp. 169–181.

[12] Foster, J. S. *Type Qualifiers: Lightweight Specifications to Improve Software Quality.* PhD thesis, University of California, Berkeley, 2002.

[13] Foster, J. S., Terauchi, T., and Aiken, A. Flow-sensitive type qualifiers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (2002), ACM Press, pp. 1–12.

[14] Henzinger, T., Jhala, R., Majumdar, R., and Sutre, G. Software verification with Blast. In *SPIN: Model Checking of Software*, Lecture Notes in Computer Science 2648. Springer, 2003, pp. 235–239.

[15] Khedker, U. P. Data flow analysis. In *The Compiler Design Handbook*. CRC Press, 2002, pp. 1–59.

[16] Kupsch, J. A., and Miller, B. P. Manual vs. automated vulnerability assessment: A case study. In *The First International Workshop on Managing Insider Security Threats* (2009), West Lafayette.

[17] Peled, D., Pelliccione, P., and Spoletini, P. Model checking. In *Wiley Encyclopedia of Computer Science and Engineering*, B. W. Wah, Ed. John Wiley & Sons, Inc., 2008.

[18] Queille, J., and Sifakis, J. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming* (1982), Springer Berlin / Heidelberg, pp. 337–351.

[19] Shankar, U., Talwar, K., Foster, J. S., and Wagner, D. Detecting format string vulnerabilities with type qualifiers. In *In Proceedings of the 10th USENIX Security Symposium* (2001), pp. 201–220.

[20] Turing, A. On computable numbers, with an application to the Entscheidungsproblem. In *Proceedings of the London Mathematical Society* (1936).

[21] Venet, A. A practical approach to formal software verification by static analysis. *Ada Lett. XXVIII*, 1 (2008), 92–95.

[22] Viega, J., Bloch, J. T., Kohno, Y., and McGraw, G. Its4: A static vulnerability scanner for c and c++ code. In *ACSAC '00: Proceedings of the 16th Annual Computer Security Applications Conference* (Washington, DC, USA, 2000), IEEE Computer Society, p. 257.

A Tools HOWTO

This appendix contains a snapshot of the documentation that was provided to CERN developers. The documentation was originally provided as a webpage, and converted with html2latex to figure here for reference purposes only. Despite an effort at cleaning the results up, some formatting issues might still appear, and all the links have been lost in the conversion.

A.1 Security Analysis Tools - Recommendations for C/C++

Flawfinder

FlawFinder is a simple yet efficient and quick tool that scans your C/C++ source code for calls to typical vulnerable library functions. It was developed by David Wheeler, a renowned security expert. It is run from the command line. Its output can easily be customized.

- Typical error types found:
  - Calls to library functions creating buffer overflow vulnerabilities (gets, strcpy, sprintf, ...)
  - Calls to library functions potentially vulnerable to string formatting attacks (sprintf, printf, ...)
  - Potential race conditions in file handling.

Installation

SLC
Run the following as root:

```
yum install flawfinder
```

Debian
Available on most Debian based distributions:

```
sudo apt-get install flawfinder
```

Others
Check Flawfinder's webpage, get the sources from there and follow the installation instructions!

Usage

Basic run
To obtain a complete (and possibly lengthy) report on your code, simply run:

```
flawfinder <path_to_your_source_directory>
```

Note: It doesn't properly check all files when run on the . directory, just run it on ./ instead. Alternatively, you can pass a list of files as argument.

Useful options

Setting the –help / -h option will provide a list of the possible options.

```
$ flawfinder --help
```

Flawfinder can provide the output in an html format (potentially easier to parse if you need to), disable header and footer of the report, . . . The following example would only output the hits (*of risk-rating at least 2*), in an html format:

```
$ flawfinder -m 2 --html --quiet --dataonly
```

Furthermore, if you wish to view only the flaws introduced in a patch, you can save the hit-list history and run a differential analysis.

```
$ flawfinder --savehistfile=prepatchhits.ffh <pre_patch_directory_or_files>
$ flawfinder --diffhistfile=prepatchhits.ffh <patched_directory_or_files>
```

False positive ignoring/reporting

Some times, Flawfinder will report items that are not bugs. In that case, you can avoid having them reported again as shown below.

Inline comments

Directly on the line which you have identified as a false positive, include:

```
strcpy(largebuffer, smallconstantbuffer) /* Flawfinder: ignore */
```

RATS

The Rough Auditing Tool for Security is an open source tool developed by Secure Software Engineers. Since then it has been acquired by Fortify, which continues to distribute it free of charge (here). It scans various languages, including C, C++, Perl, PHP and Python.

- Typical errors found (C/C++):
  - Buffer overflows
  - TOCTOU race conditions
- Typical errors not found (C/C++):
  - Design flaws
  - . . .

It is very fast and can easily be integrated into a building process without causing noticeable overhead.

Installation

SLC

```
yum install rats
```

Linux

```
wget http://www.fortify.com/servlet/download/public/rats-2.3.tar.gz
tar xfz rats-2.3.tar.gz
cd rats-2.3
./configure && make && sudo make install
```

Other systems

The latest version is available here. . .

Usage

Basic run

```
rats --resultsonly <path_to_source_directory>
```

Advanced config

```
rats --quiet --xml -w 3 <path_to_source_directory>
```

- –xml, –html generate output in the specified format
- -w <1,**[2]** ,3> set the warning level:
  - 1 will only include high level warnings (*i.e.* less false positives, but more false negatives)),
  - 2 is the medium and default option,
  - 3 will produce more output and miss less vulnerabilities, but might also report many false positives.

A.2 Security Analysis Tools - Recommendations for Java

FindBugs

FindBugs is written in Java, distributed under LGPL and although not focused on security vulnerabilities, it does find quite a few of those. It is available both as a standalone application and as an Eclipse plugin.

- Typical error types found:
  - Risks of SQL Injection, Code Injection, ...
  - General Bad practices
  - Exceptional return values not checked
  - Hard coded database passwords
  - *Customizable rulesets*

Standalone Application

Installation
The following is valid as of Jan 29th 2010, for the current version of FindBugs (1.3.9).

```
wget http://switch.dl.sourceforge.net/project/findbugs/findbugs/1.3.9/findbugs-1.3.9.tar.gz
tar xfz findbugs-1.3.9.tar.gz
```

Usage
The standalone version of FindBugs can be started with the following command (from the directory where you installed it):
Unix

```
bin/findbugs
```

Windows

```
bin\findbugs.bat
```

Once it is started, you should configure an analysis project:

- Select the File > New Project menu option (Ctrl+N works as well)
- Give the project a name
- Add the location of the classes to be analyzed to the corresponding list. (typically your bin or class folder, or a jar file.)

- (Optional) To improve FindBugs understanding of your code, you can add the required libraries to the next list (all jars have to be added, including the lib folder containing the jars is not sufficient)!
- (Optional) For better identification of the errors and to be able to get the line numbers and code extracts for the reported vulnerabilities, add the root of your source directory to the last list.

NOTE: To the best of our knowledge, the Wizard for the project creation isn't fully functional, the above steps are recommended instead.

You can then save the project configuration and analysis results for later use or evolution analysis.

Configuration

In the stand-alone version you can change the configuration via filters (Preferences –¿ Filters) to hide those issues which you do not consider relevant to your analysis.

Particular attention should be given to the bugs in the Security or in the Malicious Code Vulnerability categories. Also, the `BadUseOfReturnValue` detector should be activated.

Eclipse Plugin

Installation

NOTE: The version of Eclipse available on SLC5 is Eclipse 3.2, which not supported by the FindBugs plugin! It requires Eclipse 3.3 (Europa)...

The url for the Eclipse plugin update site is: `http://findbugs.cs.umd.edu/eclipse` for the official releases. From version 3.4 onwards (Ganymede & Galileo), the plugin installation has been simplified, just go to Help > Install New Software and enter the update site's address. Once the features have been loaded, select the one you want. (In our case, simply check the FindBugs box.) In version 3.3 (Europa), go to Help –¿ Sofware Updates –¿ Find and Install, then select Search for new features to install. Afterwards, hit the New Remote Site button, enter a name (e.g. FindBugs) and the update site url.

Usage

Once the plugin is installed, you can right click your project and select FindBugs –¿ Find Bugs. This will start the analysis of your source and offer you to switch to the FindBugs perspective.

Configuration

In the Eclipse Plugin, the configuration can be done via Window –¿ Preferences –¿ Java –¿ FindBugs.

False positive ignoring/reporting

Reported potential vulnerabilities can be categorized (e.g. *I will fix* , *not a bug* , *need further study* , etc.) either directly on the stand-alone application or through the Bug User Annotations view in the Eclipse plugin. Analysis results can be saved to an xml file, which can be shared and then loaded by collaborators, including the annotations.

CodePro Analytix

CodePro Analytix is a commercial static analyzer, built in Java and available as an Eclipse plugin. It can import vulnerability dictionaries designed for FindBugs, and comes with a rather complete set of rules of its own. Its advantages over FindBugs seem to be an increased ease of configuration and a rather complete set of collaborative features. Since it is a commercial application and CERN does not have a site wide license, the decision on whether purchase it or not is left to the interested teams.

- Homepage
- Documentation
- Free trial

A.3 Security Analysis Tools - Recommendations for Perl

Perl::Critic

Perl Critic checks whether your code complies with best practices based on Damian Conway's *Perl Best Practices.* However, it also contains a few policies relevant to security.

Typical errors found:

- Use of backtick operators
- Unsafe open / select calls
- Unchecked exceptional return values...

Errors missed:

- Calls to system or exec (those are found by RATS)
- Uncleaned/unverified user input (for detecting this, use Perl's taint mode `--T` option)

Installation

SLC5 (sorry, not available on SLC4)

```
# RUN AS ROOT
yum install perl-Perl-Critic
```

Then download the configuration file and put it into your home directory:

```
# RUN AS REGULAR USER
wget -O $HOME/.perlcriticrc http://cern.ch/security/codetools/files/.perlcriticrc
```

Other systems

Perl::Critic is available as a CPAN module, so you can use usual installation procedure. For instance:

```
# RUN AS ROOT
perl -MCPAN -e shell

cpan> install Perl::Critic
```

Then download the configuration file and put it into your home directory:

```
# RUN AS REGULAR USER
wget -O $HOME/.perlcriticrc \
http://security.web.cern.ch/security/codetools/files/.perlcriticrc
```

More information on the Perl Critic CPAN page.

Usage

```
perlcritic <directory_or_file>
```

- âseverity <1,2,3,4,5> set the severity level (from 5: gentle to 1: brutal)
- âman for a man page

Additional info

It is worth noting that `http://www.activestate.com/` distributes perlcritic with a GUI in their Perl Development Kit (more details here) NOTE: be warned though... if your perlcritic profile has colors activated, the gui will not recognize any of the colored errors!

For **vim** users, vim perlcritic compiler script is available. Additionally, perl-support plugin supports Perl::Critic.

For **emacs** users, emacsWiki has a script to interface perlcritic.

RATS

The Rough Auditing Tool for Security is an open source tool developed by Secure Software Engineers. Since then it has been acquired by Fortify, which continues to distribute it free of charge (here). It scans various languages, including C, C++, Perl, PHP and Python. In Perl code, it will mostly raise a flag when finding calls to risky built-in functions.

It is very fast and can easily be integrated into a building process without causing noticeable overhead.

Installation

SLC

```
yum install rats
```

Linux

```
wget http://www.fortify.com/servlet/download/public/rats-2.3.tar.gz
tar xfz rats-2.3.tar.gz
cd rats-2.3
./configure && make && sudo make install
```

Other systems
The latest version is available here...

Usage

Basic run

```
rats -l perl --resultsonly <path_to_source_directory>
```

Advanced config

```
rats -l perl --quiet --xml -w 3 <path_to_source_directory>
```

- –xml, –html generate output in the specified format
- -w <1,**[2]** ,3> set the warning level:
  - 1 will only include high level warnings (*i.e.* less false positives, but more false negatives)),
  - 2 is the medium and default option,
  - 3 will produce more output and miss less vulnerabilities, but might also report many false positives.

LC's lint

This script, written by Lionel Cons, checks for compliance to his Perl Programming Guide.

It is publicly available on AFS (for example from LXPLUS):

```
/afs/cern.ch/user/c/cons/public/scripts/lint <path to your script>
```

For usage information and options type `lint -h` for the perldoc page, `lint -m`.

A.4 Security Analysis Tools - Recommendations for PHP

Pixy

Installation

We are providing a wrapper for Pixy that allows handling of multiple files and parsed output.

SLC5 (sorry, not available on SLC4)

```
yum install pixy
```

Other systems

Note : This requires Java 1.6, perl and Lionel Cons' perl modules. Also, GraphViz is required to be able to view / convert the generated dependency graphs.

```
wget http://pixybox.seclab.tuwien.ac.at/pixy/dist/pixy_3_03.zip
unzip pixy_3_03.zip
cd Pixy
rm -rf run-all.pl run-all.bat scripts testfiles test src
wget http://cern.ch/security/codetools/files/pixy
sed -i "s|/media/thomas/tools/php/Pixy|.|" pixy
chmod u+x pixy
```

The latest version of Pixy can also be obtained from Pixy download page.

Usage

Warning Unfortunately, Pixy may sometimes throw a Java exception. These errors are not deterministic, so don't get discouraged and just try again running Pixy with exactly the same arguments as before.

Basic Usage

Just point Pixy to the directory with your PHP code.

```
pixy <path_to_directory>
```

Advanced Usage

```
pixy -c --xml -o report.xml -t report_directory <path_to_directory>
```

Run pixy -h for help.

RATS

The Rough Auditing Tool for Security is an open source tool developed by Secure Software Engineers. Since then it has been acquired by Fortify, which continues to distribute it free of charge (here). It scans various languages, including C, C++, Perl, PHP and Python. Unfortunately its utility is rather limited for PHP as it does not find Cross-Site Scripting or SQL Injection vulnerabilities.

* Typical errors found (PHP): * TOCTOU race conditions * Calls to system functions (warning) * Typical errors not found: * Design flaws

It is very fast and can easily be integrated into a building process without causing noticeable overhead.

Installation

SLC

```
yum install rats
```

Linux

```
wget http://www.fortify.com/servlet/download/public/rats-2.3.tar.gz
tar xfz rats-2.3.tar.gz
cd rats-2.3
./configure && make && sudo make install
```

Other systems

The latest version is available here. . .

Usage

Basic run

```
rats --resultsonly <path_to_source_directory>
```

Advanced config

```
rats --quiet --xml -w 3 <path_to_source_directory>
```

- –xml, –html generate output in the specified format
- -w <1,**[2]** ,3> set the warning level:
  - 1 will only include high level warnings (*i.e.* less false positives, but more false negatives)),
  - 2 is the medium and default option,
  - 3 will produce more output and miss less vulnerabilities, but might also report many false positives.

A.5 Security Analysis Tools - Recommendations for Python

RATS

The Rough Auditing Tool for Security is an open source tool developed by Secure Software Engineers. Since then it has been acquired by Fortify, which continues to distribute it free of charge (here). It scans various languages, including C, C++, Perl, PHP and Python. As far as python is concerned, RATS is fairly basic and will only check for risky built-in/library function calls.

It is very fast and can easily be integrated into a building process without causing noticeable overhead.

Installation

SLC

```
yum install rats
```

Linux

```
wget http://www.fortify.com/servlet/download/public/rats-2.3.tar.gz
tar xfz rats-2.3.tar.gz
cd rats-2.3
./configure && make && sudo make install
```

Other systems

The latest version is available here...

Usage

Basic run

```
rats --resultsonly <path_to_source_directory>
```

Advanced config

```
rats --quiet --xml -w 3 <path_to_source_directory>
```

- –xml, –html generate output in the specified format
- -w <1,**[2]** ,3> set the warning level:
  - 1 will only include high level warnings (*i.e.* less false positives, but more false negatives)),
  - 2 is the medium and default option,
  - 3 will produce more output and miss less vulnerabilities, but might also report many false positives.

pychecker

Pychecker is a lint-like tool for python, which will mostly find bugs that would be found by compilers for less dynamic languages. However, it has very few checks concerning security.

Installation

SLC

```
yum install pychecker
```

Others

Check the pychecker homepage or

```
wget http://surfnet.dl.sourceforge.net/project/pychecker/\
pychecker/0.8.18/pychecker-0.8.18.tar.gz
tar xfz pychecker-0.8.18.tar.gz
cd pychecker-0.8.18
python setup.py install
```

Usage

Basic run

```
pychecker --quiet file1.py file2.py ...
```

Advanced config

```
pychecker --quiet -# 100 -e warning file1.py file2.py ...
```

By default, pychecker will only report the 10 first hits. This can be changed with the -# option. The -e options allows you to set the minimal level of errors that will be reported, *i.e.* internal errors will always be reported and style checks will only be performed if you ask for them. The available levels are (in decreasing order): internal, error, security, warning, unused, deprecated, style. We recommend using the **warning** level. Many options are configurable for pychecker, such as toggling on or off some of the detectors, managing output, . . . It is probably best to have a look at pychecker -h and decide for yourself which ones suit you best.

B Sample Output

The figure below presents an example of a graph output by Pixy. It clearly shows how an untrusted value controlled by the user, a `GET` / `POST` value, is returned by a function (`getarg`), stored in a local variable `$q` and after two concatenations "." is then used in the body of the page rendered.

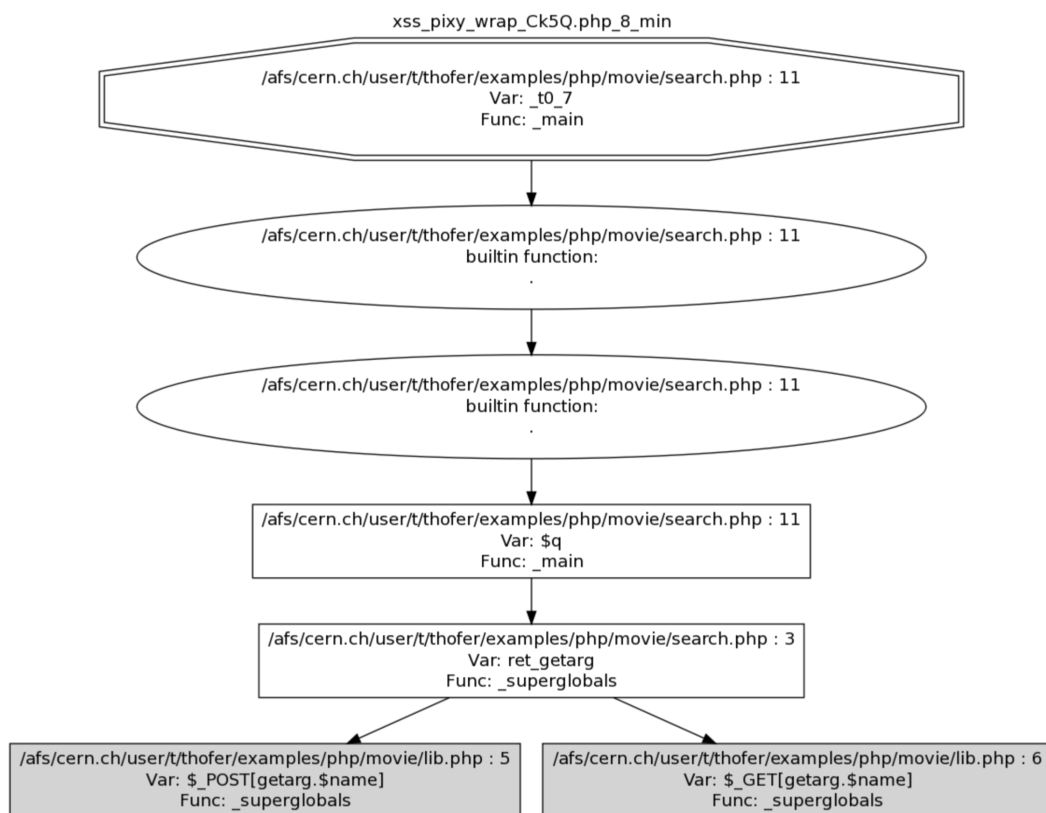

Figure B-1: An example Pixy graph

Additional contributions

C.1 Wrapper for Pixy

```
1  #!/usr/bin/perl -T
2
3  use strict;
4  use warnings;
5  use English;
6  use LC::Secure qw(environment);
7  use LC::Exception;
8  use LC::Process qw(execute);
9  use LC::Option;
10 use LC::Util qw($ProgramName);
11 use LC::Find qw(:FIND);
12 use HTML::Entities;
13 use File::Find;
14 use File::Temp qw(tempfile);
15 use Cwd qw(abs_path);
16
17 use Carp qw(croak);
18
19 #
20 # Constants
21 #
22
23 use constant JAVA_OPT1  => "-Xms256m";
24 use constant JAVA_OPT2  => "-Xmx1024m";
25 use constant PIXY_HOME  => "/usr/lib/pixy";
26 use constant CLASSPATH  => PIXY_HOME."/lib:".PIXY_HOME."/build/class";
27
28 use constant DESC => {
29     SQL     => "possible SQL injection vulnerability",
30     XSS     => "possible Cross-site scripting vulnerability",
31     File    => "possible remote file inclusion vulnerability"
32 };
33
34
35 #
36 # Global variables
37 #
38
39 our ($OS, @files, $dir, $tmpdir, $tmpfile, $targetdir, $outfile, $ignore);
40
41 #
42 # Initialize it all
43 #
44
45 sub init () {
46     $| = 1;
47     LC::Exception::Context->new()->will_report_all();
48     $OS = LC::Option::define("$ProgramName [OPTIONS] [--] [File or Directory]",
49         [ "help=h",                undef,  "show this help" ],
50 #       [ "manual=m",              undef,  "show the manpage" ],
51         [ "targetdir=t:string", undef,  "directory to store the report and graphs" ],
52         [ "context=c",             undef,  "print the faulty line" ],
53         [ "xml",                   undef,  "output in xml format" ],
54         [ "html",                  undef,  "output in html format" ],
55         [ "output=o:string",       undef,  "file for output" ],
56         [ "ignore=i:string",       undef,
```

```
57                  "files to ignore (unless included by other PHP scripts): e.g. \"test.inc,bluecat
                        *.php\"" ]
58     );
59     LC::Option::parse_argv($OS);
60
61     $OS->handle_help($ProgramName, q$Rev: 14 $, q$Date: 2010-02-17 17:23:58 +0100 (Wed, 17 Feb
           2010) $);
62 #     $OS->handle_manual();
63
64     if (scalar(@ARGV) > 1) {
65   print "Wrong arguments, or wrong order of arguments.\n\n", $OS->usage();
66   exit(1);
67     } elsif (scalar(@ARGV) == 1) {
68         ( $dir ) = $ARGV[0] =~ m|^([-\w\./]+)$|;
69     } else {
70         $dir = ".";
71     }
72
73     $targetdir = "";
74     if ($OS->value("targetdir")) {
75         $targetdir = $OS->value("targetdir");
76         $targetdir = LC::File::path_for_open($targetdir);
77         $targetdir = abs_path($targetdir);
78         ($targetdir) = $targetdir =~ m/^([-\w.\/]+)$/;
79         LC::File::makedir($targetdir, 0700) or
80             croak("Unable to create report directory: $!");
81     } else {
82         $targetdir = LC::File::random_directory("/tmp/pixy_report_XX", 0700);
83     }
84
85     $outfile = "";
86     if ($OS->value("output")) {
87         $outfile = $OS->value("output");
88         $outfile = LC::File::path_for_open($outfile);
89         $outfile = abs_path($outfile);
90         ($outfile) = $outfile =~ m/^([-\w.\/]+)$/;
91     }
92
93     $ignore = "";
94     if ($OS->value("ignore")) {
95         my ($ignoreitems) = $OS->value("ignore") =~ m/^([-\w.\/\*,]*)$/;
96         $ignoreitems =~ s/\./\\\./g;
97         $ignoreitems =~ s/\*/.\*/g;
98         my @ignorelist = split(",", $ignoreitems);
99         $ignore = "(".join("|", @ignorelist).")";
100     }
101 }
102
103 #
104 # Find matching files (called by $finder->find...
105 # See find_php_files below!
106 #
107
108 sub findfiles () {
109     my ($name, $path) = ($LC::Find::Name, $LC::Find::Path);
110     if ($name =~ m/^.*\.(php[3-5]?|inc)$/i) {
111         if ($ignore eq "" or $path !~ m/$ignore$/) {
112             push(@files, $path);
113         }
114     }
115 }
116
117
118 #
119 # Recursively find the php files at the given directory
120 #
121
122 sub find_php_files ($) {
123     my ($directory) = @_;
124     my ($finder);
125
126     $directory = abs_path($directory);
127
128     @files = ();
129
130     $finder = LC::Find->new();
131     $finder->flags(FIND_FORGIVING);
132     $finder->file_callback(\&findfiles);
133
134     $finder->find($directory) or
135         croak("Error running the find command: $!");
136 }
137
138 #
```

```
139 # Create the one file to import them all!
140 #
141 sub create_import_file () {
142     my (@lines, $contents);
143     $tmpdir = LC::File::random_directory("/tmp/pixy_tmp_XX", 0700);
144     (undef, $tmpfile) =
145         File::Temp::tempfile("pixy_wrap_XXXX", SUFFIX => ".php", DIR => $tmpdir);
146     $tmpfile = abs_path($tmpfile);
147     ( $tmpfile ) = $tmpfile =~ m|([-\w./]+)|;
148
149     @lines = ();
150     push(@lines, "<?");
151     foreach my $file (@files) {
152         push(@lines, "\tinclude_once(\"$file\");");
153     }
154     push(@lines, "?>");
155
156     $contents = join("\n", @lines);
157
158     LC::File::file_contents($tmpfile, $contents);
159 }
160
161 #
162 # Run Pixy
163 #
164
165 sub run_pixy ($) {
166     my ($input) = @_;
167     my (@cmd, $stdout);
168     @cmd = ();
169     $stdout = "";
170
171     push(@cmd, "java");
172     push(@cmd, JAVA_OPT1);
173     push(@cmd, JAVA_OPT2);
174     push(@cmd, "-Dpixy.home=".PIXY_HOME);
175     push(@cmd, "-classpath");
176     push(@cmd, CLASSPATH);
177     push(@cmd, "at.ac.tuwien.infosys.www.pixy.Checker");
178     push(@cmd, "-a");
179     unless ($targetdir eq "") {
180         push(@cmd, "-o");
181         push(@cmd, "$targetdir");
182     }
183     push(@cmd, "-y");
184     push(@cmd, "xss:sql:file");
185     push(@cmd, $input);
186
187     execute(\@cmd, "stdout" => \$stdout);
188
189     return \$stdout;
190 }
191
192 #
193 # Make the .dot file a .png!
194 #
195
196 sub convert_graph ($) {
197     my ($graph_ref) = @_;
198     my ($ori_path, $type, $num, @cmd, $stdout, $name, $dest_path);
199
200     if ($graph_ref =~ m/^(sql|xss|file)(\d+)$/) {
201         $type = $1;
202         $num = $2;
203     } else {
204         croak("Unexpected graph reference.");
205     }
206
207     (undef, undef, $name) = File::Spec->splitpath($tmpfile);
208
209     $ori_path = "$targetdir/$type\_$name\_$num\_min.dot";
210     unless ( -e $ori_path ) {
211         $ori_path = "$targetdir/$type\_$name\_$num\_dep.dot";
212     }
213
214     unless ( -e $ori_path ) {
215         croak ("Couldn't open the .dot graph file: $ori_path");
216     }
217
218     $dest_path = "$targetdir/$type-$num.png";
219     @cmd = ();
220     push(@cmd, "dot");
221     push(@cmd, "-Tpng");
222     push(@cmd, $ori_path);
```

```
223     push(@cmd, "-o");
224     push(@cmd, $dest_path);
225
226     execute(\@cmd) or croak ("Failed to convert the graph file: $ori_path");
227
228     return $dest_path;
229 }
230
231 #
232 # Extract the relevant information for a SQL injection vulnerability report.
233 #
234
235 sub extract_sql_vuln (@) {
236     my (@lines) = @_;
237     my ($file, $line, $graph, $cond, %vuln);
238     unless (scalar(@lines) == 3) {
239         croak ("Incorrect sql slice!")
240     }
241     foreach my $info (@lines) {
242         if ($info =~ m/^- ([-\w.\/]+):(\d+)$/) {
243             $file = $1;
244             $line = $2;
245         } elsif ($info =~ m/^- Graphs: (sql\d+)$/) {
246             $graph = $1;
247         } elsif ($info =~ m/^- (.*conditional.*)$/) {
248             $cond = $1;
249         } else {
250             croak ("Unexpected sql output from Pixy.");
251         }
252     }
253
254     if ($file !~ "" and $line != 0 and $graph !~ "" and $cond !~ "") {
255         $vuln{"file"} = $file;
256         $vuln{"line"} = $line;
257         $vuln{"graph"} = convert_graph($graph);
258         $vuln{"cond"} = $cond;
259         $vuln{"type"} = "SQL";
260         return \%vuln;
261     }
262
263     croak ("Pixy vulnerability sql report is lacking information.");
264 }
265
266 #
267 # Extract the relevant information for a XSS vulnerability report.
268 #
269
270 sub extract_xss_vuln (@) {
271     my (@lines) = @_;
272     my ($file, $line, $graph, $cond, %vuln);
273     unless (scalar(@lines) == 3) {
274         croak ("Incorrect xss slice!")
275     }
276     foreach my $info (@lines) {
277         if ($info =~ m/^- ([-\w.\/]+):(\d+)$/) {
278             $file = $1;
279             $line = $2;
280         } elsif ($info =~ m/^- Graph: (xss\d+)$/) {
281             $graph = $1;
282         } elsif ($info =~ m/^- (.*conditional.*)$/) {
283             $cond = $1;
284         } else {
285             croak ("Unexpected xss output from Pixy.");
286         }
287     }
288
289     if ($file !~ "" and $line != 0 and $graph !~ "" and $cond !~ "") {
290         $vuln{"file"} = $file;
291         $vuln{"line"} = $line;
292         $vuln{"graph"} = convert_graph($graph);
293         $vuln{"cond"} = $cond;
294         $vuln{"type"} = "XSS";
295         return \%vuln;
296     }
297
298     croak ("Pixy xss vulnerability report is lacking information.");
299 }
300
301 #
302 # Extract the information from a file injection vulnerability report.
303 #
304
305 sub extract_file_vuln (@) {
306     my (@lines) = @_;
```

```
307     my ($file, $line, $graph, $cond, %vuln);
308     unless (scalar(@lines) == 3) {
309         croak ("Incorrect file slice!")
310     }
311     foreach my $info (@lines) {
312         if ($info =~ m/^- File:\W+([-\w.\/]+)$/) {
313             $file = $1;
314         } elsif ($info =~ m/^- Line:\W+(\d+)$/) {
315             $line = $1;
316         } elsif ($info =~ m/^- Graph: (file\d+)$/) {
317             $graph = $1;
318         } else {
319             croak ("Unexpected file output from Pixy.");
320         }
321     }
322
323     if ($file !~ "" and $line != 0 and $graph !~ "" and $cond !~ "") {
324         $vuln{"file"} = $file;
325         $vuln{"line"} = $line;
326         $vuln{"graph"} = convert_graph($graph);
327         $vuln{"type"} = "File";
328         return \%vuln;
329     }
330
331     croak ("Pixy file vulnerability report is lacking information.");
332 }
333
334
335 #
336 # Parse the output from Pixy and make it and array of vulnerabilities.
337 #
338 sub parse_output ($) {
339     my ($output_ref) = @_;
340     my ($output, @lines, @vulnerabilities, $errortype, $counter);
341     $output = $$output_ref;
342
343     @lines = split("\n", $output);
344     @vulnerabilities = ();
345     $errortype = "";
346
347     $counter = 0;
348
349     while (defined($lines[$counter])) {
350         my ($msg, $file, $line, $graph, $cond, $inline);
351         $msg = $file = $graph = $cond = "";
352         $line = 0;
353         $inline = $lines[$counter];
354
355         if ($inline =~ m/^(\w+) Analysis BEGIN$/i) {
356             $errortype = $1;
357             $counter++;
358             next;
359         }
360
361         if ($inline =~ m/^(\w+) Analysis END$/i) {
362             if ($1 !~ m/^\Q$errortype\E$/i) {
363                 croak ("Ill-formated report.");
364             } else {
365                 $errortype = "";
366                 $counter++;
367                 next;
368             }
369         }
370
371         if ($errortype =~ m/^SQL$/i and $inline =~ m/directly tainted/i) {
372             if ($lines[$counter+3]) {
373                 push(@vulnerabilities,
374                         extract_sql_vuln(@lines[$counter+1..$counter+3]));
375             }
376             $counter = $counter+4;
377             next;
378         }
379
380         if ($errortype =~ m/^XSS$/i and $inline =~ m/^Vulnerability detected\!$/i) {
381             if ($lines[$counter+3]) {
382                 push(@vulnerabilities,
383                         extract_xss_vuln(@lines[$counter+1..$counter+3]));
384             }
385             $counter = $counter+4;
386             next;
387         }
388
389         if ($errortype =~ m/^File$/i and $inline =~ m/^Line:\W+\d+$/i) {
390             if ($lines[$counter+2]) {
```

```
391                 push(@vulnerabilities,
392                         extract_file_vuln(@lines[$counter..$counter+2]));
393             }
394             $counter = $counter+3;
395             next;
396         }
397 
398 
399         $counter++;
400     }
401 
402     return \@vulnerabilities;
403 }
404 
405 #
406 # Get the line accused by Pixy
407 #
408 
409 sub get_context ($$) {
410     my ($file, $line) = @_;
411     my @lines;
412     $file = LC::File::path_for_open($file);
413     @lines = split("\n", LC::File::file_contents($file));
414     if ($lines[$line-1]) {
415         return $lines[$line-1];
416     } else {
417         return "";
418     }
419 }
420 
421 #
422 # Print the contents in a xml structure
423 #
424 
425 sub print_xml ($) {
426     my ($ref) = @_;
427     my (@lines);
428     unless(ref($ref) eq "ARRAY") {
429         croak ("Not an array reference!");
430     }
431 
432     @lines = ();
433 
434     push(@lines, "<?xml version=\"1.0\"?>");
435     push(@lines, "<pixy_output>");
436     push(@lines, "<stats>");
437     push(@lines, "<vuln_count>".scalar(@$ref)."</vuln_count>");
438 # TODO: do we want / need more stats? time, vulnerabilities per category?
439     push(@lines, "</stats>");
440 
441 # TODO: output the vulnerabilities... sorted?
442     foreach my $vuln_ref (@$ref) {
443         my %vuln = %$vuln_ref;
444         if (exists($vuln{"file"}) and exists($vuln{"line"}) and
445                 exists($vuln{"graph"}) and exists($vuln{"type"}) and
446                 DESC->{$vuln{"type"}} and
447                 (exists($vuln{"cond"}) or $vuln{"type"} =~ m/^File$/)) {
448             push(@lines, "<vulnerability>");
449             push(@lines, "\t<type>".$vuln{"type"}."</type>");
450             if (exists($vuln{"cond"})) {
451                 push(@lines, "\t<condition>".$vuln{"cond"}."</condition>");
452             }
453             push(@lines, "\t<file>");
454             push(@lines, "\t\t<name>".$vuln{"file"}."</name>");
455             push(@lines, "\t\t<line>".$vuln{"line"}."</line>");
456             push(@lines, "\t</file>");
457             push(@lines, "\t<message>");
458             push(@lines, "\t\t".DESC->{$vuln{"type"}});
459             push(@lines, "\t</message>");
460             push(@lines, "\t<graph>".$vuln{"graph"}."</graph>");
461             if ($OS->value("context")) {
462                 my $html = get_context($vuln{"file"}, $vuln{"line"});
463                 encode_entities($html);
464                 push(@lines, "\t<context>");
465                 push(@lines, "\t\t$html");
466                 push(@lines, "\t</context>");
467             }
468             push(@lines, "</vulnerability>");
469         } else {
470             croak ("Unexpected data in list of vulnerabilities.");
471         }
472     }
473 
474     push(@lines, "</pixy_output>");
```

```
475     return \@lines;
476 }
477
478 #
479 # Print the contents in a html structure
480 # Parameters:
481 #    - the contents to output
482 #    - boolean flag: make the links relative, for the creation of an html file
483 #    in the target repository (in order to make it easier to zip the output into
484 #    an independant archive)
485 #
486
487 # TODO:
488 #    - $dir is often . - use something more meaningful
489
490 sub print_html ($$) {
491     my ($ref, $make_links_rel) = @_;
492     my (@lines, $longline);
493     unless(ref($ref) eq "ARRAY") {
494         croak ("Not an array reference!");
495     }
496
497     @lines = ();
498
499     push(@lines, "<\!DOCTYPE HTML  PUBLIC \"-//W3C//DTD HTML 4.01 Transitional//EN\"");
500     push(@lines, " \"http://www.w3.org/TR/html4/loose.dtd\">");
501
502     push(@lines, "<html lang=\"en\">");
503
504     push(@lines, "<head>");
505     push(@lines, "\t<meta http-equiv=\"Content-Type\" content=\"text/html; "
506         ."charset=utf-8\">");
507     push(@lines, "\t<title>Pixy Report for $dir</title>");
508
509 #TODO: Improve style...
510
511     push(@lines, "<STYLE type=\"text/css\">");
512     push(@lines, "body { background: #F1F1F1; color: #0C0C0C;}");
513     push(@lines, "#vuln_list {");
514 #     push(@lines, "\twidth: 640px;");
515     push(@lines, "}");
516     push(@lines, "#vuln_list ol {");
517     push(@lines, "\tlist-style: decimal inside;");
518     push(@lines, "\tfont-size: 15pt;");
519     push(@lines, "}");
520     push(@lines, "#vuln_list ol li {");
521     push(@lines, "\tpadding: 5px 5px 5px 5px;");
522     push(@lines, "\tmargin-left: -10px;");
523     push(@lines, "\tmargin-bottom: 10px;");
524 #     push(@lines, "\tbackground: #666;");
525     push(@lines, "}");
526     push(@lines, "#vuln_list ol li ul {");
527     push(@lines, "\tlist-style: disc;");
528     push(@lines, "\tfont-size: 12pt;");
529     push(@lines, "}");
530     push(@lines, "#vuln_list ol li ul li {");
531 #     push(@lines, "\tcolor: #669933;");
532     push(@lines, "\tpadding: 0 0 0 0;");
533     push(@lines, "\tmargin-left: 10px;");
534     push(@lines, "\tmargin-bottom: 0px;");
535     push(@lines, "}");
536     push(@lines, "</STYLE>");
537
538     push(@lines, "</head>");
539
540     push(@lines, "<body>");
541
542     push(@lines, "<h1>Pixy's reported vulnerabilities</h1>");
543     push(@lines, "<h3>Ran on $dir</h3>");
544     push(@lines, "\tGenerated ".LC::Util::timestamp(time()));
545
546     $longline = "<p>Found ".scalar(@$ref)." vulnerabilities in ";
547     $longline .= "total...</p>";
548 # TODO: Do we want more complete stats?
549     push(@lines, $longline);
550
551     push(@lines, "<div id=\"vuln_list\">");
552
553     push(@lines, "<ol>");
554     foreach my $vuln_ref (@$ref) {
555         unless (ref($vuln_ref) eq "HASH") {
556             croak ("Unexpected value in vulnerabilities array.");
557         }
558         push(@lines, "<li>");
```

```
559           my %vuln = %$vuln_ref;
560           if(exists($vuln{"type"}) and DESC->{$vuln{"type"}}) {
561                 $longline = "<b>";
562                 $longline .= DESC->{$vuln{"type"}};
563                 $longline .= "</b>";
564                 push(@lines, $longline);
565           }
566         push(@lines, "\t\t<ul>");
567           if(exists($vuln{"file"})) {
568                 $longline = "\t\t\t<li>File: ";
569                 $longline .= $vuln{"file"};
570             if(exists($vuln{"line"})) {
571                       $longline .= ", Line: ";
572                       $longline .= $vuln{"line"};
573                 }
574                 $longline .= "</li>";
575                 push(@lines, $longline);
576           }
577           if ($OS->value("context")) {
578                 my $html = get_context($vuln{"file"}, $vuln{"line"});
579                 encode_entities($html);
580                 push(@lines, "\t\t\t<li>Code snippet:");
581             push(@lines, "<pre><code class=\"php\">");
582                 push(@lines, $html);
583                 push(@lines, "</code></pre></li>");
584           }
585           if(exists($vuln{"cond"})) {
586                 $longline = "\t\t\t<li>";
587                 $longline .= $vuln{"cond"};
588                 $longline .= "</li>";
589                 push(@lines, $longline);
590           }
591           if(exists($vuln{"graph"})) {
592                 $longline = "\t\t\t<li>Graph file: ";
593                 my $graph = $vuln{"graph"};
594                 if ($make_links_rel) {
595                       $graph =~ s/$targetdir/./g;
596                 }
597                 $longline .= "<a href=\"$graph\">";
598                 $longline .= $graph."</a>";#
599                 $longline .= "</li>";
600                 push(@lines, $longline);
601           }
602           push(@lines, "\t\t</ul>");
603           push(@lines, "\t</li>");
604     }
605     push(@lines, "</ol>");
606
607     push(@lines, "</div>");
608
609     push(@lines, "</body>");
610     push(@lines, "</html>");
611
612     return \@lines;
613 }
614
615 #
616 # Print the contents in a plain text structure
617 #
618
619 sub print_text ($) {
620     my ($ref) = @_;
621     my (@lines);
622     unless(ref($ref) eq "ARRAY") {
623           croak ("Not an array reference!");
624     }
625
626     @lines = ();
627
628     foreach my $vuln_ref (@$ref) {
629           my %vuln = %$vuln_ref;
630           if (exists($vuln{"file"}) and exists($vuln{"line"}) and
631                       exists($vuln{"graph"}) and exists($vuln{"cond"}) and
632                       exists($vuln{"type"}) and DESC->{$vuln{"type"}}) {
633                 my $line = $vuln{"file"}.":".$vuln{"line"};
634                 push(@lines, $line);
635                 $line = "   ".DESC->{$vuln{"type"}};
636                 push(@lines, $line);
637                 $line = "   graph: ".$vuln{"graph"};
638                 push(@lines, $line);
639                 if ($OS->value("context")) {
640                       push(@lines, "==>".get_context($vuln{"file"}, $vuln{"line"}));
641                 }
642
```

```
643             } else {
644                 croak ("Unexpected data in list of vulnerabilities.");
645             }
646         }
647
648         return \@lines;
649 }
650
651
652 #
653 # Select the output function
654 #
655
656 sub print_output ($) {
657         my ($output_ref) = @_;
658         my ($contents, $lines_ref, @lines);
659         if ($OS->value("xml")) {
660             $lines_ref = print_xml($output_ref);
661         } elsif ($OS->value("html")) {
662             $lines_ref = print_html($output_ref, 0);
663         } else {
664             $lines_ref = print_text($output_ref);
665         }
666         unless (ref($lines_ref) eq "ARRAY") {
667             croak ("Not an array reference.");
668         }
669
670         @lines = @$lines_ref;
671
672         $contents = join("\n", @lines);
673
674         if ($outfile eq "") {
675             print "$contents\n";
676         } else {
677             LC::File::file_contents($outfile, $contents);
678         }
679 }
680
681 #
682 # Create an html report in the target dir.
683 #
684
685 sub html_to_target_dir($) {
686         my ($output_ref) = @_;
687         my ($report_file, $lines_ref, @lines, $contents);
688
689         $report_file = $targetdir."/index.html";
690
691         $lines_ref = print_html($output_ref, 1);
692
693         @lines = @$lines_ref;
694
695         $contents = join("\n", @lines);
696
697         LC::File::file_contents($report_file, $contents) or
698             croak ("Coulnd't write report.html in \"$targetdir\".");
699         return $report_file;
700 }
701
702
703 #
704 # Main
705 #
706
707 init();
708 find_php_files($dir);
709 if (scalar(@files) == 0) {
710         printf("No file matching *.(php[3-5]?|inc) found in \"%s\".\n\n", $dir);
711         print $OS->usage();
712         exit(1);
713 }
714 create_import_file();
715 my $output_ref = run_pixy($tmpfile);
716 my $clean_out_ref = parse_output($output_ref);
717 print_output($clean_out_ref);
718
719 my $report_html = html_to_target_dir($clean_out_ref);
720 if (!$OS->value("xml") && !$OS->value("html")) {
721     print "\nFull report in $report_html\n\n";
722 }
723
724 LC::File::destroy($tmpdir) or croak ("Couldn't erase temporary dir.");
725
726 __END__
```

```
727
728 =head1 NAME
729
730 pixy_wrapper - run Pixy on all php files found at the given location
731
732 =head1 SYNOPSIS
733
734 B<pixy_wrapper> [I<OPTIONS>] [DIRECTORY]
735
736 =head1 DESCRIPTION
737
738 Pixy is a static source code analysis tool for PHP. It specifically targets
739 risks of SQL injection and Cross-Site Sciprting vulnerabilities. Its default
740 distribution is limited to analysing single php files and needs to be run
741 individually on each file in your project. However, Pixy does scan all the files
742 required or included by the file it is given.
743 Taking advantage of this construction, pixy_wrapper will create a temporary php
744 file that include all of the php files in the directory given in argument.
745 Then it will run Pixy on that file, thus effectively scanning your whole
746 project.
747 As Pixy does crash on some files though, we also provide an option to ignore
748 given files. Parameters to the ignore option can be a list of comma separated
749 filenames and allow globbing.
750
751 Use pixy_wrapper -h to view the available options and man pixy_wrapper to display
752 this page.
753
754 =head1 AUTHOR
755
756 Thomas Hofer <Thomas.Hofer@cern.ch>
757
758 =head1 MAINTAINER
759
760 Sebastian Lopienski <Sebastian.Lopienski@cern.ch>
761
762 =head1 VERSION
763
764 $Id: pixy_wrapper, v. 1.0 2010/02/12 17:05 thofer Exp $
765
766 =cut
```

C.2 Script to Retrieve Wiki Pages

The documentation in Appendix A was created with DokuWiki (`http://www.dokuwiki.org`), because the Computer Security Team at CERN uses this software to keep versioned information for general purposes. One additional advantage was the ability to have the documentation reviewed and commented upon by the other people in the team. However, this could not be used directly to present the results publicly, as some of the information on the wiki is confidential and thus none of the wiki pages is world-readable, by decision of the team. Therefore, the simplest way to have an up-to-date version of the documentation, while retaining the dynamic aspect of the wiki, without making any of the wiki public, is to create a static copy of the wiki contents and update it as necessary. I have thus created a script that can be configured to retrieve any set of pages from the wiki, using the user supplied credentials to access the corresponding webpages over `HTTPS`. It then filters out the unnecessary headers and footers of the HTML document thus obtained, replacing it with user defined templates. It also removes all of the editing links and transforms the wiki internal `URL` anchors into relative paths, corresponding to the names of the files it creates. The configuration of the script is handled by internal constants, this way it only needs to be configured once for a given set of pages and whenever those pages have been updated the user only has to confirm the copying process by entering his password.

```
1 #!/usr/bin/perl -T
2 # Imports
3
4 use strict;
5 use warnings;
6 use English;
7 use LC::Secure qw(environment);
8 use LC::Exception;
9 use LC::File;
10 use LC::Option;
11 use LC::Process qw(execute);
12 use LC::Util qw($ProgramName);
13 use Term::ReadKey;
14
15 use CAF::FileWriter;
16
```

```
17 use Carp qw(croak);
18
19 # Constants
20 # TODO: Modify to fit your needs
21
22 use constant PROJECT_ROOT   => "projects:source_code_security_tools";
23 use constant HREF_BASE      => "/service-cert/wiki/doku.php?id=".PROJECT_ROOT;
24 use constant REMOTE_ROOT    => "https://service-cert.web.cern.ch".HREF_BASE;
25 use constant TARGET_DIR     => "target/";
26 use constant COOKIES_FILE   => "cookies";
27 use constant CSS_BASE_FILE  => "dokuwiki.css";
28 use constant CSS_FILE_NAME  => "wikipage.css";
29
30 # Global variables
31
32 our ($OS, $path, %pages, %refs, $user, $passwd, $title);
33
34 #
35 # Adds all the pages to the hashes,
36 # TODO: Modify to fit your needs
37 #
38
39 sub populate_pages () {
40     add_page("documentation", "documentation", "index.html");
41     add_page("c_tools", "documentation:c_tools", "c_tools.html");
42     add_page("c", "documentation:c", "c.html");
43     add_page("cpp", "documentation:cpp", "cpp.html");
44     add_page("java_tools", "documentation:java_tools", "java_tools.html");
45     add_page("php_tools", "documentation:php_tools", "php_tools.html");
46     add_page("perl_tools", "documentation:perl_tools", "perl_tools.html");
47     add_page("perl", "documentation:perl", "perl.html");
48     add_page("python_tools", "documentation:python_tools", "python_tools.html");
49     add_page("general", "documentation:general", "general.html");
50 }
51
52 #
53 # Initialize everything
54 #
55
56 sub init () {
57     $| = 1;
58     LC::Exception::Context->new()->will_report_all();
59     $OS = LC::Option::define("$ProgramName [OPTIONS] [--] [path ...]",
60         [ "help=h",             undef,  "show some help" ],
61         [ "manual=m",           undef,  "show the man page" ],
62         [ "inURL=i:path!",      undef,  "URL to copy from" ],
63         [ "outPath=o:string",   undef,  "path to the output file" ],
64         [ "update=u:boolean",   1,      "Deactivate to destroy and recreate" ]
65     );
66     LC::Option::parse_argv($OS);
67     $OS->handle_help($ProgramName,  q$Revision: 0.2 $,
68                             q$Date: 2010/01/19 $ );
69     $OS->handle_manual();
70
71
72     if (!$OS->value("update")) {
73         LC::File::destroy(TARGET_DIR) or croak $!;
74     }
75
76     LC::File::makedir(TARGET_DIR, 0700);
77     LC::File::copy(CSS_BASE_FILE, TARGET_DIR.CSS_FILE_NAME);
78
79     %pages  = ();
80     %refs   = ();
81 }
82
83 #
84 # Add a page to be processed
85 #
86
87 sub add_page ($$$) {
88     my ($name, $remote, $localurl) = @_;
89     $pages{$name} = {
90         remote      => $remote,
91         localurl    => $localurl
92     };
93     $refs{$remote} = $localurl;
94 }
95
96 #
97 # Retrieve the user name to use for cURLing the data
98 #
99
100 sub user () {
```

```
101     if (!defined($user)) {
102         $user = getpwuid($EUID);
103     }
104     return $user;
105 }
106
107 #
108 # Ask the user for his password, to be used when cURLing
109 #
110
111 sub pass () {
112     if (!defined($passwd)) {
113         print "Enter host password for user '", user(), "': ";
114         ReadMode 'noecho';
115         $passwd = ReadLine 0;
116         chomp $passwd;
117         $passwd =~ m{^([[:print:]]*)$} or croak "Weird password...";
118         $passwd = $1;
119         ReadMode 'normal';
120         print "\n";
121     }
122     return $passwd;
123 }
124
125 #
126 # Retrieve page title and store it
127 #
128
129 sub get_title ($) {
130     my ($page) = @_;
131
132     if ($page =~ m/<h1>(.+)<\/h1>/) {
133         $title = $1;
134     } elsif ($page =~ m/<h2>(.+)<\/h2>/) {
135         $title = $1;
136     } else {
137         $title = "";
138     }
139
140     $title =~ s/<[^>]*>//g;
141 }
142
143 #
144 # Retrieve a page as an array of lines
145 # and look for a title...
146 #
147
148 sub curl_page ($) {
149     my ($page) = @_;
150     my ($pageurl, @cmd, $stdout, $stderr, @lines);
151     $stdout = $stderr = "";
152
153     $pageurl = &REMOTE_ROOT.":".$page;
154
155     @cmd = ("curl");
156     push(@cmd, "-u".user().":".pass());
157 #   Don't want to show the progress info...
158     push(@cmd, "-s");
159     push(@cmd, $pageurl);
160
161     execute(\@cmd, "stdout" => \$stdout);
162
163     get_title($stdout);
164
165     @lines = split(/\n/, $stdout);
166
167     return @lines;
168
169 }
170
171 #
172 # Process a page, remove wiki header and footer,
173 # Adapt the internal links...
174 #
175
176 sub extract_page (@) {
177     my (@filecontents) = @_;
178     my (@pagecontents, $write, $startpattern, $stoppattern, $editbutton);
179     @pagecontents = ();
180     $startpattern = "<\!-- wikipage start -->";
181     $stoppattern = "<\!-- wikipage stop -->";
182     $editbutton = "class=\"secedit\"";
183     $write = 0;
184     foreach my $line (@filecontents) {
```

```
185         $line =~ s/<acronym[^>]*>(.*?)<\/acronym>/$1/gi;
186         $line =~ s/title=\"\w*(:\w*)*\"//g;
187         if (!$write and $line =~ m/$startpattern/i) {
188             $write = 1;
189         } elsif ($write and $line =~ m/$stoppattern/i) {
190             return @pagecontents;
191         } elsif ($write and $line =~ m/class=\"wikilink[12]\"/) {
192             my $regex = qr/href=\"\Q${\(HREF_BASE)}\E:([^\"]*)\"/;
193             while ($line =~ $regex) {
194                 my ($href, $link, $anchor);
195                 $href = $1;
196                 $link = "";
197                 $anchor = "";
198                 if ($href =~ m/([^#]*)(#.*)/) {
199                     ($link, $anchor) = ($1, $2);
200                 } else {
201                     $link = $href
202                 }
203                 if (exists($refs{$link})) {
204                     $link = $refs{$link};
205                 } else {
206                     $link =~ s/^([^:]*:)*([^:]*)$/$2.html/;
207                 }
208                 $line =~ s/$regex/href=\"$link$anchor\"/;
209             }
210             push(@pagecontents, $line);
211         } elsif ($write and $line =~ /\w/ and $line !~ m/$editbutton/i) {
212             push(@pagecontents, $line);
213         }
214     }
215 }
216 
217 #
218 # Set the html header..
219 # TODO: Modify according to your needs...
220 #
221 
222 sub preamble() {
223     my (@header) = ();
224 
225     push(@header, "<\!DOCTYPE html  PUBLIC \"-//W3C//DTD XHTML 1.0 Transitional//EN\"");
226     push(@header, " \"http://www.w3c.org/TR/xhtml1/DTD/xhtml1-transitional.dtd\">");
227 
228     push(@header, "<html xmlns=\"http://www.w3c.org/1999/xhtml\" xml:lang"
229         ."=\"en\" lang=\"en\">");
230 
231     push(@header, "<head>");
232     push(@header, "\t<meta http-equiv=\"Content-Type\" content=\"text/html; "
233         ."charset=utf-8\" />");
234     push(@header, "\t<title>$title</title>");
235     push(@header, "\t<link rel=\"stylesheet\" media=\"all\" type=\"text/css\" "
236         ."href=\"".CSS_FILE_NAME."\" />");
237 
238     push(@header, "</head>");
239 
240     push(@header, "<body>");
241 
242     push(@header, "<div class=\"dokuwiki\">");
243     push(@header, "<div class=\"page\">");
244     push(@header, "");
245 
246     return @header;
247 }
248 
249 #
250 # Set the html footer...
251 # TODO: Modify according to your needs...
252 #
253 
254 sub postlude() {
255     my (@footer) = ();
256 
257     push(@footer, "");
258     push(@footer, "</div>");
259     push(@footer, "</div>");
260 
261     push(@footer, "</body>");
262     push(@footer, "</html>");
263 
264     return @footer;
265 }
266 
267 #
268 # Print the new page to a local file
```

```
269 #
270
271 sub output($@) {
272     my ($outfile, @lines) = @_;
273     my ($fh);
274
275 # Open/Create file and write to it...
276     $fh = CAF::FileWriter->open(TARGET_DIR.$outfile);
277     print $fh join("\n", preamble());
278     print $fh join("\n", @lines);
279     print $fh join("\n", postlude());
280     $fh->close();
281 }
282
283 # Main
284
285 init();
286 populate_pages();
287 foreach my $key (keys(%pages)) {
288     my (@lines, @contents);
289     @lines = curl_page($pages{$key}{remote});
290     @contents= extract_page(@lines);
291     output($pages{$key}{localurl}, @contents);
292 }
293
294 __END__
295
296 =head1 NAME
297
298 get_wiki - Copy wiki pages from dokuWiki to local html files
299
300 =head1 SYNOPSIS
301
302 B<get_wiki> [I<OPTIONS>]
303
304 =head1 DESCRIPTION
305
306 This program retrieves pages from a dokuwiki installation and creates local html
307 files. It uses curl over HTTPS to get the pages' content, which is why it
308 requests your password.
309
310 The internal dokuwiki links are converted to relative links, using the I<%refs>
311 hash, which is populated by the I<populate_pages> sub. Adapt its contents as
312 necessary. Whenever a link points to an item not defined in the hash, it will be
313 replaced by a link to an html file whose name is the last element of the
314 dokuwiki id (I<i.e.>, the element after the last semi-column.
315
316 This program also gets rid of heading and trailing dokuwiki-specific html and
317 replaces it with a custom header and footer, defined in I<preamble> and
318 I<postlude> respectively.
319
320 The I<constants> declared at the beginning of the program allow further
321 customization.
322
323 Use "get_wiki -h" to see the list of options and "get_wiki -m" to see this
324 documentation.
325
326 =head1 AUTHOR
327
328 Thomas Hofer C<mailto:thomas.hofer@cern.ch>, (C) CERN C<http://www.cern.ch>
329
330 =head1 VERSION
331
332 $Id: get_wiki,v 0.2 2010/01/19 16:51 thofer Exp $
333
334 =cut
```