

# The Secflow source code security checker: Current problems and solution alternatives

Submitted as a work-in-progress to the  
23<sup>rd</sup> Annual Computer Security Conference, Dec. 10-14, 2007, Miami, USA

Dr. Holger Peine and Stefan Mandel

*first.last@iese.fraunhofer.de*

Fraunhofer Institute for Experimental Software Engineering, Kaiserslautern, Germany

At least since Microsoft has declared security a primary goal for its future software in 2002, the fact has started to move into the focus of research and practice that programming errors underlie many, if not most of the security vulnerabilities our IT systems suffer from. Therefore, programming errors that would result in a vulnerability of the fielded software should be recognized as early as possibly during software development. While the growing area of secure software engineering has already produced various manual techniques to consider security in early development stages, such as threat analysis (aka threat modelling) or security design patterns, automatic source code security checker tools are the only option when dealing with the 98% of software that has not been (and still is not) developed using secure software engineering techniques.

Research on such tools has expanded considerably over the last few years, building on older research in static analysis of programs, and commercial tools have appeared from companies like Fortify Software and Ounce Labs. However, static source code analysis for security is a complex problem, the design space of technologies that can be applied to its various aspects is large, and concrete solutions must make many trade-offs.

At Fraunhofer IESE, we are developing a source code security checker named "Secflow" to detect security properties derived from relationships such as invocation ("who calls whom", even across polymorphic method invocations), alias relationships ("does  $x$  really refer to the same object as  $y$ ?"), and dataflow relationships ("can the value of  $x$  influence the value of  $y$  in any way?"). A simple example for a security-relevant property building on top of such source-level relationships is information flow control: Say that module  $S$  should keep its data secret from module  $P$  (e.g. because  $P$  interfaces with the public Internet, or because  $S$  and  $P$  process data from mutually mistrusting subjects, e.g. competing customers of the same service provision company). A source-level formulation of this property could be "look for a variable  $v$  that is written in  $S$ , another variable  $w$  that is read in  $P$ , and a dataflow from  $v$  to  $w$ " (or even "a dataflow without proper 'cleaning' of the data, by which we mean  $z$ ").

The main target of the Secflow tool, however, will be vulnerabilities resulting from processing data from untrusted sources (such as the Internet) without proper validation. This large class of vulnerabilities includes e.g. popular problems like SQL injection and cross-site scripting. Our tool recognizes such vulnerabilities by tracking the program-internal dataflow from "untrusted sources" (such as network input operations) to "vulnerable sinks" (such as database accesses) and by checking for any code along such a "critical path" that matches one of a number of code patterns for known validation operations.

The Secflow tool's workflow starts with several front-ends specifically developed for each source language to be analyzed; a front-end for the Java language has been completed, and

another front-end for the .NET languages (like C# and Visual Basic.NET) is underway. The front-ends compile the source language into a common intermediate code (for a virtual register machine) which is then transformed to static-single-assignment (SSA) form to ease the subsequent global data flow analysis. The dataflow analysis derives all critical paths of the program in the above sense, and the subsequent validation analysis matches the validation patterns against each path and aggregates the results from the individual patterns into an estimation of whether a given path constitutes indeed should be reported as a vulnerability. Once the Secflow tool reaches a state that is usable by external users, we will publish it as open source software, inviting others to contribute further validation patterns or even further language front-ends.

In addition to the Java language front-end, a first version of the dataflow analysis engine has been completed, and it appears to deliver all critical paths correctly; however, its performance is still unsatisfactory. One dimension of the design space here is how closely our analysis should simulate actual program execution (e.g. what type of points-to / alias analysis to use, how to deal with reflection, exceptions, persistent storage, and many other issues). Another dimension is how to represent external code such as libraries that is called by the analyzed program, but shall not be included in the analysis.

Regarding code patterns for data validation, we have identified a few general, language-independent patterns, like `if (x>MAX) return ERR_OUT_OF_RANGE;` or `if (!isValid(x)) x = makeValid(x);`. These have to be complemented by language- and platform-specific patterns; we are still in the process of collecting these from real-world programs. We are still working on finding the best level of abstraction to represent validation patterns, which could be phrased in terms of intermediate code, or in terms of the later dataflow analysis.