

# **A Practical Guide to Vulnerability Checkers**

## **Secologic Project**

**Written by:**

Martin Johns

University of Hamburg / Security in Distributed Systems

johns AT informatik.uni-hamburg.de

© 2006

This work was supported by the German Ministry of Economics (BMWi) as part of the project "secologic", [www.secologic.org](http://www.secologic.org).

## Table of contents:

|  |    |
|--|----|
| Introduction .....   | 3  |
| About this document .....                                    | 3  |
| Organisation of the document .....                           | 3  |
| Part One: Syntactic checkers .....                           | 4  |
| When to use syntactic checkers .....                         | 4  |
| Overview .....   | 4  |
| The Tools.....   | 5  |
| Flawfinder .....   | 5  |
| RATS - Rough Auditing Tool for Security .....                | 7  |
| PScan: A limited problem scanner for C source files .....    | 9  |
| ITS4 - Software Security Tool .....                          | 10 |
| Part Two: Tools that support or require code annotation..... | 12 |
| When to use annotation based checkers .....                  | 12 |
| Overview .....   | 12 |
| The Tools.....   | 13 |
| CQUAL - A tool for adding type qualifiers to C .....         | 13 |
| Splint - Secure Programming Lint / Specifications Lint.....  | 16 |
| Part Three: Specialized tools.....                           | 19 |
| Overview .....   | 19 |
| The Tools.....   | 20 |
| BOON: Buffer Overrun detectiON .....                         | 20 |
| MOPS - MOdelchecking Programs for Security properties.....   | 22 |
| Further Reading.....   | 25 |

# Introduction

## ***About this document***

One of the main causes for security problems are programming errors. These errors may be obvious (e.g. the usage of `gets()` which is prone to buffer overflows) or subtle (e.g. an integer overflow caused by C++'s `new[]` operator). Depending on the knowledge of the programmer many of these errors may be hard to spot. To aide the process of finding security problems in source code, various tools have been developed.

This document is meant to ease the first steps in using the described tools and hereby lowering the initial effort in adapting this technology. It does not contain detailed information about the internal mechanisms of the tools. Also no statements are made about the quality of the tools' analysis results.

Only tools which are freely obtainable and for which the source code is available are covered. These conditions ensure the possibility to customize the tools in order to integrate them into existing processes.

## ***Organisation of the document***

For every presented tool the following information are provided:

- Where to get the source code
- How to install the tool
- How to run it
- Types of detected security problems
- Requirements for input data
- Syntax of the tool's output
- Links to further information

## Part One: Syntactic checkers

The class of syntactic checkers consist of tools that work on an almost purely syntactic level. These tools examine the source code to detect the existence of programming constructs that are know to cause security problems. Every occurrence of these constructs is reported. The analysis of these tools is thorough but prone to “false positives” (the reporting of safe constructs).

Due to the limited functionality of this class of tools, sophisticated security problem cannot be detected. For example data flow analysis (which is required to reliably detect cross site scripting problems) is out of the scope of these tools.

### ***When to use syntactic checkers***

Due to the fact, that syntactic checkers ignore problems of a higher semantic level and report a large ratio of false positives, these tools are not well suited for big software projects and for auditing projects after their completion.

Syntactic checkers work well on a small level. They are suited for quick and reoccurring tests of single source code files during the development process. They should rather be used by the programmer of the code than by a third party.

### ***Overview***

For this section the following tools were selected:

- Flawfinder
- Rats
- Pscan
- ITS4

## ***The Tools***

### **Flawfinder**

Flawfinder was written by David Wheeler, the author of the “Secure Programming for Linux and Unix HOWTO“. Wheeler’s goal was to create a simple and easy to learn tool that requires no installation. These characteristics were supposed to encourage a wider adoption in the open source world.

**Author:**

David A. Wheeler, community (see ChangeLog).

**Homepage:**

<http://www.dwheeler.com/flawfinder/>

**License:**

GNU GPL v2

**Quick Facts:**

Platform: written in Python, so it should run on any platform supporting Python (UNIX, Windows, OS/2, Mac, Amiga, others)

First released Version: 0.12 (2001-05-21)

Current Version: 1.26 (2004-06-15)

**Supported Languages:**

C, C++

**Found Problems:**

- Buffer Overflows
- Format String Problems
- Shell Executions
- Insecure Tmpfiles
- Race Conditions
- Access Violations
- Weak Random/Crypto
- User Input

**Installation:**

As Flawfinder is written in Python, there is no need to compile it. The actual program consists of one Python-File, so no real installation is needed. One just needs to unpack the archive and run the program.

**Usage:**

```
flawfinder *.c  
or  
flawfinder source/
```

**Input:**

plain C/C++ source code

**Automation:**

Flawfinder can be integrated into VIM/Emacs by letting the editor parse the output. Flawfinder can compare the results to previous runs.

**Output:**

Flawfinder prints potential security flaws, sorted by risk. It prints the filename and line-number of the potential problems, the risk value, and the category of the problem and the name of the identified function. It then explains why this function often is a problem.

Flawfinder also displays a summary over the complete run, printing the total number of hits, lines analyzed, SLOC (Source Lines Of Code) found, hits sorted by risk, and hits per SLOC.

**Example:**

```
Flawfinder version 1.26, (C) 2001-2004 David A. Wheeler.
Number of dangerous functions in C/C++ ruleset: 158
Examining example7.c
example7.c:36: [4] (format) printf:
    If format strings can be influenced by an attacker, they can be
    exploited. Use a constant for the format specification.
example7.c:17: [2] (buffer) char:
    Statically-sized arrays can be overflowed. Perform bounds checking,
    use functions that limit length, or ensure that the size is larger than
    the maximum possible length.
example7.c:24: [1] (buffer) strncpy:
    Easily used incorrectly; doesn't always \0-terminate or check for
    invalid pointers. Risk is low because the source is a constant string.
example7.c:26: [1] (buffer) strncpy:
    Easily used incorrectly; doesn't always \0-terminate or check for
    invalid pointers.

Hits = 4
Lines analyzed = 43 in 0.54 seconds (1210 lines/second)
Physical Source Lines of Code (SLOC) = 25
Hits@level = [0]  0 [1]  2 [2]  1 [3]  0 [4]  1 [5]  0
Hits@level+ = [0+]  4 [1+]  4 [2+]  2 [3+]  1 [4+]  1 [5+]  0
Hits/KSLOC@level+ = [0+] 160 [1+] 160 [2+]  80 [3+]  40 [4+]  40 [5+]  0
Minimum risk level = 1
Not every hit is necessarily a security vulnerability.
There may be other security vulnerabilities; review your code!
```

**Quick Benchmark:**

```
scanning wu-ftp-2.6.2 (~20000 lines): 3 seconds
scanning sendmail-8.13.5 (~80000 lines): 8.56 seconds
scanning firefox-1.0.7 (~2400000 lines): 200 seconds
```

## **RATS - Rough Auditing Tool for Security**

RATS' functionality is very close to Flawfinder's. Both tools were developed simultaneously. When the respective developers discovered the other tool, it was decided to release the first version of each tool at exact the same day. RATS is one of the few tools that support the programming languages Perl, PHP and Python.

### **Author:**

Secure Software, Inc.

### **Homepage:**

[http://www.securesoftware.com/resources/download\\_rats.html](http://www.securesoftware.com/resources/download_rats.html)

### **License:**

GNU GPL v2

### **Quick Facts:**

Platform: Written in C, uses Flex and Expat. It runs on UNIX- and Windows-Commandline.

First released Version: 0.9 (2001-05-21)

Current Version: 2.1 (2002-09-25)

### **Supported Languages:**

C, C++, Perl, PHP, Python

### **Found Problems:**

- Buffer Overflows
- Format String Problems
- Shell Executions
- Insecure Tmpfiles
- Race Conditions
- Access Violations
- Weak Random
- User Input

### **Installation:**

- Make sure you have a C-Compiler, flex and expat.
- Then unpack the source package, change into its base-directory and run
- `./configure`
- Then run "make" and "make install"
- RATS gets installed to `/usr/local/bin`, the datafiles to `/usr/local/share` and
- the manual page to `/usr/local/man/man1`.

### **Usage:**

```
rats *.c
```

or

```
rats source/
```

**Input:**

Plain C/C++/Perl/PHP/Python source code. The files need to be named appropriate (.c, .cc, .pl, .php, .py) to allow the usage of the correct rule set (e.g. the Perl rule set). Alternatively the tool's language option must be set, to enforce the usage of a specified rule set.

**Automation:**

RATS can output its report in XML-Format. Automation is therefore easily possible.

**Output:**

RATS prints potential problems sorted by severity, then by function name, then by file, then by line number. For each function name rats prints an explanation of the problem (if available in the vulnerability database). Finally Rats prints the number of lines analyzed and the time used.

**Example:**

Analyzing example7.c

```
example7.c:17: High: fixed size local buffer
Extra care should be taken to ensure that character arrays that are
allocated on the stack are used safely. They are prime targets for buffer
overflow attacks.
```

```
example7.c:36: High: printf
Check to be sure that the non-constant format string passed as argument 1
to this function call does not come from an untrusted source that could
have added formatting characters that the code is not prepared to handle.
```

```
Total lines analyzed: 44
Total time 0.000395 seconds
111392 lines per second
```

**Quick Benchmark:**

```
scanning wu-ftp-2.6.2 (~20000 lines): 0.07 seconds
scanning sendmail-8.13.5 (~80000 lines): 0.3 seconds
scanning firefox-1.0.7 (~2400000 lines): 2.8 seconds
```

## **PScan: A limited problem scanner for C source files**

### **Author:**

Alan DeKok

### **Homepage:**

<http://www.striker.ottawa.on.ca/~aland/pscan/>

### **License:**

GNU GPL v2

### **Paper:**

<http://cert.uni-stuttgart.de/archive/bugtraq/2000/07/msg00256.htmlb>  
(Paper from Pascal Bouchareine on Format String Bugs in general)

### **Quick Facts:**

Platform: UNIX command-line program, written in C and flex.

First released Version: 2000-07-07

Current Version: 1.2 (2000-08-10)

### **Supported Languages:**

C

### **Found Problems:**

Format String Problems in printf-style C-Functions

### **Installation:**

Unpack the archive, run "make", then you can execute the resulting binary.

### **Usage:**

```
pscan [-vw] [-p problem_file] <files ...>
```

### **Options:**

- v            Verbose mode.
- w            Show warnings when a variable is used as the format argument.

### **Input:**

Plain C source code

### **Automation:**

Pscan can be called automatically.

### **Output:**

Pscan just prints the filename and line-number of the found potential format string problems.

### **Quick Benchmark:**

scanning wu-ftpd-2.6.2 (~20000 lines): 0.2 seconds

scanning sendmail-8.13.5 (~80000 lines): 0.2 seconds

scanning firefox-1.0.7 (~2400000 lines): 4 seconds

## ITS4 - Software Security Tool

### Author:

John Viega (Cigital, Inc.)

### Homepage:

<http://www.cigital.com/its4/>

### License:

Open Source, commercial use restricted

### Paper:

"ITS4: A Static Vulnerability Scanner for C and C++ Code"

<http://www.cigital.com/papers/download/its4.pdf>

### Quick Facts:

Platform: Written in C, runs on UNIX and Windows, provides minimal integration for Emacs and Microsoft Visual Studio

First released Version: probably 1.0 (2000-02-21)

Current Version: 1.1.1 (2000-10-02)

### Supported Languages:

C, C++

### Found Problems:

- Buffer Overflows
- Format String Problems
- Shell Executions
- TOCTOU
- Usage of weak random number generation
- User Input

### Installation:

You need a C Compiler

- Unpack the source package, change into its base-directory and run
- `./configure`
- Then run "make" and "make install" (as root).
- ITS4 gets installed to `/usr/local/` by default.

You have to include the script `emacs/its4.el` from the source package into your `~/.emacs` file, if you want Emacs integration.

### Usage:

ITS4 can be used on the commandline and has some integration for Emacs and Microsoft Visual Studio.

The Emacs integration is limited to scanning a bunch of files and linking the problems found by ITS4 to the files. Type "M-x its4-scan" and enter the file(s) which should be analyzed. ITS4 then analyzes the file and displays the results. You can then jump to the corresponding source line by moving the cursor on the error and pressing "enter", or by a middle click with the mouse on the error.

When used from the commandline, you type e.g.

```
its4 *.c
```

ITS4 scans the files and displays the found problems.

### **Input:**

plain C or C++ source code

### **Automation:**

The paper on ITS4 mentions an Emacs-integration scanning the code as the user types it, using syntax highlighting. This Emacs-integration could sadly not be found. As the output format is a quasi-standard, ITS4 could probably be integrated easily within other editors or into build processes.

### **Output:**

```
$ its4 fingerd.c
fingerd.c:112:(Urgent) fprintf
Non-constant format strings can often be attacked.
Use a constant format string.
-----
fingerd.c:91:(Risky) execv
Many potential problems.
Close all fds, clean the environment, set the umask to something good, and
reset uids before calling.
-----
fingerd.c:97:(Risky) fdopen
Can be involved in a race condition if you open things after a poor check.
For example, don't check to see if something is not a symbolic link before
Opening it. Open it, then check bt querying the resulting object. Don't
run tests on symbolic file names...
Perform all checks AFTER the open, and based on the returned object, not a
symbolic name.
-----
fingerd.c:99:(Some risk) getc
Be careful not to introduce a buffer overflow when using in a loop.
Make sure to check your buffer boundries.
-----
```

This output from the commandline interface prints the found problems with filename, linenumbr and found function-name. The results are first sorted for severity, then for function-names. A short description of the problem and possible solutions are displayed. The sort order can be changed with commandline parameters to ITS4. The vulnerability database contains the most often misused C functions.

### **Quick Benchmark:**

only .c and .h files are scanned:

scanning wu-ftpd-2.6.2 (~25000 lines): 0.2 seconds

scanning sendmail-8.13.5 (~128000 lines): 0.4 seconds

scanning firefox-1.0.7 (~1600000 lines): 6.7 seconds

## Part Two: Tools that support or require code annotation

There is one fact that will always be true concerning static analysis of source code: The programmer *knows* the semantics of a certain code fragment, while a static analysis tool has to *conclude* or even to *guess* (based on evidence) them. Code annotations are additional syntax constructs that specify semantic properties of the source code. Tools that support such annotations enable the programmer to aide the process of vulnerability discovery.

Even though the presented programs are not general purpose tools (i.e. they are not able to identify every vulnerability class) they are still very versatile.

### ***When to use annotation based checkers***

Annotations obviously cause additional work for the programmer. But if they are continuously applied to the source code during the whole development lifecycle, they can be a valuable asset to ensure the software's quality. It is therefore recommendable to decide early in favour or against the usage of such a tool, because the annotation of source code after its completion is time-consuming and error prone.

### ***Overview***

For this section the following tools were selected:

- CQUAL
- Splint

## ***The Tools***

### **CQUAL - A tool for adding type qualifiers to C**

CQUAL extends the programming language's type system with additional type qualifiers. These qualifiers specify a value's semantic characteristics. For example, every value obtained from an untrusted source can be qualified as "tainted". Only special filter functions are able to change the value's qualifier to "untainted" (i.e. trusted). During the compilation process, type inference is used to identify illegal usage of "mistyped" values: All execution paths that would lead to the usage of an untrusted "tainted" value in a security sensitive function (e.g. as part of a format string), can be identified.

#### **Author:**

Jeff Foster, David Wagner, Rob Johnson, Alex Aiken, and others.

#### **Homepage:**

<http://www.cs.umd.edu/~jfoster/cqual/>

#### **License:**

GNU GPL v2

#### **Papers:**

- "A Theory of Type Qualifiers"  
<http://www.cs.umd.edu/~jfoster/papers/pldi99.pdf>
- "Detecting Format-String Vulnerabilities with Type Qualifiers"  
<http://www.cs.umd.edu/~jfoster/papers/usenix01.pdf>
- "Flow-Sensitive Type Qualifiers"  
<http://www.cs.umd.edu/~jfoster/papers/pldi02.pdf>
- "Type Qualifiers: Lightweight Specifications to Improve Software Quality"  
<http://www.cs.umd.edu/~jfoster/papers/thesis.pdf>
- "Checking and Inferring Local Non-Aliasing"  
<http://www.cs.umd.edu/~jfoster/papers/pldi03.pdf>
- "Finding User/Kernel Pointer Bugs With Type Inference"  
<http://www.cs.berkeley.edu/~rtjohnso/papers/cquk.ps>

#### **Quick Facts:**

Platform: Written in C, supports Emacs integration.

First released Version: 0.9 (2001-09-13)

Current Version: 0.991 (2004-03-28)

#### **History:**

CQUAL has evolved from CARILLON, an early version of the type qualifier system used, written in SML/NJ. CQUAL parses C programs with the frontend of the region compiler from David Gay, which is based on GCC. The results can be displayed and browsed in Emacs with the help of the Program Analysis Mode (PAM).

### Supported Languages:

- C
- C++ is supported by Oink (<http://www.cs.berkeley.edu/~dsw/oink.html>) which is based on CQUAL
- Java will be supported by JQUAL, in development in the CQUAL CVS

### Found Problems:

- Format String Problems (with taint qualifiers)
- Deadlocks (with qualifiers on locks)
- User-/Kernel-Space Trust Errors (with qualifiers on data copied between user space and kernel space)
- Y2K Bugs (with qualifiers on date strings)

CQUAL allows more, e.g. inferring const qualifiers, and can be extended by defining custom qualifiers and inference rules.

### Installation:

You need a C Compiler, Emacs and LaTeX (for the documentation).

- Unpack the source package, change into its base-directory
- `./configure`
- Then run "make" and "make install" (as root).
- CQUAL gets installed to `/usr/local/` by default.

You have to include the following lines into your `~/.emacs` file, in order to use CQUAL:

```
(setq load-path (cons "/usr/local/share/emacs/site-lisp/cqual"
load-path))
(load "cqual-pam")
```

### Usage:

CQUAL can be used on the commandline or via Emacs.

In Emacs, type "m-x cqual" and enter the file which should be analyzed. CQUAL then analyzes the file and displays the results with the help of "Program Analysis Mode (PAM)". PAM is a generic interface for marking up programs in Emacs and included in the CQUAL distribution. You can then interactively explore the analysis via hyperlinks. You can e.g. view the inferred types and the path leading to this type.

When used from the commandline, you type e.g.

```
cqual -config /usr/local/share/cqual/lattice examples/taint1.c
CQUAL analyzes the program and prints out possible type-errors and the paths leading to these type-errors.
```

### Input:

CQUAL needs C source code annotated with type qualifiers. As CQUAL can infer qualifiers, you only need to place type qualifiers at a handful of key places in the program.

**Automation:**

If the developer uses Emacs, the integration via PAM comes probably very handy. As CQUAL includes a commandline interface, it can also be integrated into other automated processes.

**Output:**

```
$ cqual -config /usr/local/share/cqual/lattice taint1.c
taint1.c:1 ``getenv'' used but not defined
taint1.c:2 ``printf'' used but not defined
taint1.c:7 incompatible types in assignment
*s: $tainted $untainted
taint1.c:1      $tainted <= *getenv_ret
taint1.c:7      <= *s
taint1.c:8      <= *t
taint1.c:9      <= *printf_arg1
taint1.c:2      <= $untainted
```

This output from the commandline interface prints used but not defined globals, and then lists type errors found. It shows the variable in which the type-error is detected (*s* in this case), and prints the inferred types (*\$tainted* and *\$untainted* in this case, which is an error). Then the output includes the path leading to the error, showing in which lines the type is defined or inferred.

## Splint - Secure Programming Lint / SPecifications Lint

Splint examines source code by trying checking code specifications. These specifications are added to the code in form of pre- or postconditions (for example: the precondition of a `strcpy` instruction could be, that the destination buffer has at least be as big as the source buffer). The conditions are specified in the source code using C comments, which adhere Splint's specification syntax. Basic C instructions don not need explicit annotations; Splint knows most of C's semantics. Splint already provides annotated libraries for e.g identifying Buffer Overflows.

### Authors:

David Evans, David Larochelle

### Homepage:

<http://www.splint.org/>

### License:

GNU GPL v2

### Papers:

- "LCLint: A Tool for Using Specifications to Check Code"  
<http://www.cs.virginia.edu/~evans/fse94.pdf>
- "Static Detection of Dynamic Memory Errors"  
<http://www.cs.virginia.edu/~evans/pldi96.pdf>
- "Statically Detecting Likely Buffer Overflow Vulnerabilities"  
<http://lclint.cs.virginia.edu/usenix01.pdf>
- "Improving Security Using Extensible Lightweight Static Analysis"  
<http://www.cs.virginia.edu/evans/pubs/ieeesoftware-revised.pdf>

### Quick Facts:

Platform: UNIX command-line program, written in C

First released Version: 3.0.1 (2002-01-07)

Current Version: 3.1.1 (2003-04-29)

### Supported Languages:

C

### Found Problems:

- Buffer Overrun problems by bounds checking with constraints
- Format String vulnerabilites with lexical analysis or taint analysis
- Memory Access violations, e.g. dereferencing null pointers, deallocation errors
- Memory leaks
- Undefined values/parameters, ignored return values, unused code
- and many more checks like public/private abstraction errors

**Installation:**

You can compile Splint from the source distribution or try the binary distribution from <http://www.splint.org/linux.html>.

If you want to compile Splint, you need gcc and make:

- Get the source from <http://www.splint.org/source.html> and unpack it
- Change into the source tree
- type `./configure`
- `make`
- The program can be run from the source tree.

If you compiled Splint or fetched the binary distribution and unpacked it, you need to setup your environment:

- Set `LARCH_PATH` to e.g. `$HOME/src/splint-3.1.1/lib/`,
- `LCLIMPORTDIR` to e.g. `$HOME/src/splint-3.1.1/imports/`
- include e.g. `$HOME/src/splint-3.1.1/bin/` in your `PATH`.

Example for bash:

```
export LARCH_PATH=$HOME/src/splint-3.1.1/lib/
export LCLIMPORTDIR=$HOME/src/splint-3.1.1/imports/
export PATH=$PATH:$HOME/src/splint-3.1.1/bin/
```

Then you can run the "splint" binary.

**Input:**

Splint needs plain C source code including needed headers.

**Usage:**

Call "splint" with the source files to be checked. Additional include paths can be given with the commandline parameter `-I`. What checks are performed can be controlled by giving commandline parameters, see the splint documentation for available checks/parameters.

**Output:**

Splint parses the given sourcefiles and performs the desired checks or tries to solve constraints on buffer bounds when desired. It then displays the results:

```
~$ splint +bounds buf.c
Splint 3.1.1 --- 28 Apr 2003

buf.c: (in function main)
buf.c:5:2: Likely out-of-bounds store:
  strcpy(dst, src)
Unable to resolve constraint:
requires 4 >= 9
  needed to satisfy precondition:
  requires maxSet(dst @ buf.c:5:9) >= maxRead(src @ buf.c:5:14)
  derived from strcpy precondition: requires maxSet(<parameter 1>) >=
  maxRead(<parameter 2>)
A memory write may write to an address beyond the allocated buffer.
(Use -likely-boundswrite to inhibit warning)

Finished checking --- 1 code warning
```

Here Splint has found a likely buffer overflow in function main in file buf.c at line 5 column 2. Splint was unable to resolve the constraint to satisfy the precondition for the C library function strcpy, i.e. the destination buffer is at least as big as the source buffer. The actual used buffers are noted in the precondition as "dst" at line 5 column 9 and "src" at column 14.

**Automation:**

Splint provides no integration into editors or build processes, but the output is possibly parseable by vim or Emacs for providing links to jump to the corresponding code.

**Quick Benchmark:**

Splint runs in the magnitude of seconds per 1000 lines of code checked, dependant on the checks it performs.

## **Part Three: Specialized tools**

In this section, tools that solely cover a certain area of security problems are presented. These tools are not applicable for general purpose source code analysis.

### ***Overview***

For this section the following tools were selected:

- BOON
- MOPS

## ***The Tools***

### **BOON: Buffer Overrun detectiON**

BOON is exclusively targeting “Buffer Overflow” vulnerabilities. To achieve this, BOON uses integer range constraints to track buffer sizes that might occur during program execution. It is therefore more capable to minimize the number of false alerts compared to the tools discussed in Part One.

#### **Author:**

David Wagner, in collaboration with Jeff Foster, Eric Brewer, Alex Aiken

#### **Homepage:**

<http://www.cs.berkeley.edu/~daw/boon/>

#### **License:**

Boon was released under a custom Open Source license.

#### **Paper:**

"A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities"

<http://www.cs.berkeley.edu/~daw/papers/overruns-ndss00.ps>

#### **Quick Facts:**

Platform: UNIX command-line program, BOON is written in SML/NJ.

First released Version: 1.0 (2002-07-03)

Current Version: 1.0 (2002-07-03)

#### **Supported Languages:**

C

#### **Found Problems:**

Buffer Overrun problems by solving integer range constraints.

#### **Installation:**

First you need SML/NJ installed. Probably the most easy way is to get the RPM for version 110.0.7-4 from <ftp://ftp.research.bell-labs.com/dist/smlnj/release/110.0.7/RPMS/> and if needed convert them to your distribution's format (e.g. with alien).

After installing SML/NJ:

- Unpack the BOON source-package,
- change in to the boon-directory
- type "make".
- Boon needs to run from its source-directory, so you have to add that directory to your \$PATH.

#### **Input:**

BOON needs preprocessed C source code. It provides a script "preproc":

```
boon-1.0$ preproc examples/main.c
```

This call produces the file `examples/main.i`.

**Usage:**

Call "boon" from its source-directory with the preprocessed .i files:

```
boon-1.0$ boon examples/main.i
```

BOON then parses the file and generates and solves the constraints.

**Output:**

BOON displays the runtime for parsing the source and solving the constraints.

It then lists possible vulnerabilities:

```
POSSIBLE VULNERABILITIES:
Possibly a buffer overflow in `x@main()':
  10..10 bytes allocated, 0..19 bytes used.
  <- siz(x@main())
  <- len(x@main())
```

Here BOON has found a possible buffer overflow in the buffer "x". It has determined that there may be up to 19 bytes written to a buffer of size 10.

**Automation:**

BOON provides no integration into editors or build processes.

**Quick Benchmark:**

BOON seems to have problems with even moderate sized programs like wu-ftpd without changing some of the code. For a figure: BOON exits after generating constraints for about 10.000 lines of wu-ftpd and needs 30 seconds till then. It seems, therefore, recommendable to apply BOON during the development process to smaller program units.

## **MOPS - Modelchecking Programs for Security properties**

MOPS allows the identification of all security problems, that can be modelled by a finite state automation (FSA). MOPS is therefore especially suited for finding temporal problems, that may occur if different execution paths may result in different security conditions. While the construction of appropriate FSAs is non trivial, MOPS guarantees a reliable detection of the problems for which FSAs are available.

### **Author:**

Hao Chen, David Wagner

### **Homepage:**

- <http://www.cs.ucdavis.edu/~hchen/mops/>
- <http://www.cs.berkeley.edu/~daw/mops/> (outdated)

### **License:**

4-clause BSD, some parts GNU GPL/LGPL v2

### **Papers:**

- "MOPS: an infrastructure for examining security properties of software"  
<http://www.cs.ucdavis.edu/~hchen/paper/ccs02.pdf>
- "Model checking one million lines of C code"  
<http://www.cs.ucdavis.edu/~hchen/paper/ndss04.pdf>
- "Using Build-Integrated Static Checking to Preserve Correctness Invariants"  
<http://www.cs.ucdavis.edu/~hchen/paper/ccs04.pdf>
- "Model Checking An Entire Linux Distribution for Security Violations"  
<http://www.cs.ucdavis.edu/~hchen/paper/acsac05.pdf>

### **Quick Facts:**

Platform: UNIX program, written in Java and C

First released Version: 0.9 (2002-06-14)

Current Version: 0.9.2 (2005-02-19)

### **Supported Languages:**

C

### **Found Problems:**

- Everything that is a temporal safety property and can be modelled as a FSA.
- Provided examples:
  - Wrong chroot/chdir calls
  - Dangerous system calls in privileged context
  - Missing string termination in context of strncpy/strncat
  - Closed standard file descriptors (stdin, stdout, stderr)
  - Unsafe tempfile creation
  - Non-constant format strings (preliminary, many false positive)

**Installation:**

To compile MOPS, you need gcc, make and Java 1.4.

- Get the source from "http://www.cs.ucdavis.edu/~hchen/mops/"
- unpack it.
- Change into the source tree
- type "make".
- For testing the compiled program, type "cd test ; make".
- MOPS can be run from the source tree.

**Input:**

MOPS works on plain C source, but can also check source RPMs.

**Usage:**

For checking a single C source file, you can invoke MOPS like follows:

```
$MOPSDIR/bin/mops -m $PROPERTY -t $TMPDIR -o $OUTPUTDIR \\  
  -- "gcc -o foo foo.c"
```

where \$MOPSDIR is the pathname of the source tree, \$PROPERTY points to the meta-FSA which is to be checked and \$OUTPUTDIR is the directory to which MOPS should write its results. \$TMPDIR should be obvious.

**Output:**

MOPS provides a wrapper around gcc, so that it runs whenever you run gcc. It prints its results in two formats, text and HTML.

The HTML output allows for easy browsing of the results providing even highlighted source code.

The text output \$OUTPUTDIR/texttrace/ consists of an index file for each generated executable, which lists the checked properties and for each violation a filename of a trace leading to the violation is noted.

Example (for file "foo.c" \$OUTPUTDIR=output and \$PROPERTY=chrootchdir.mfsa):

**Invocation:**

```
$ ../bin/mops -m chrootchdir.mfsa -t temp/ -o output/ -- "gcc -o foo foo.c"
```

```
INFO: mops: Running gcc -o foo foo.c  
Checking the program for the property:  
The program violates the property. Writing error trace 1  
Writing to /tmp/output/texttrace/chrootchdir/foo.1
```

```
/tmp/output/texttrace/chrootchdir/foo:  
mfsa: /home/christian/src/mops-0.9.2/doc/tutorial/chrootchdir.mfsa  
mfsalabel: Chroot vulnerability  
cfg: /tmp/temp/CFG/foo-G0C7.mcfg  
/tmp/output/texttrace/chrootchdir/foo.1
```

"foo-G0C7.mcfg" is the control flow graph which when model checked against the property-FSA leads to the errors stated in "foo.1"

```

/tmp/output/texttrace/chrootchdir/foo.1:
-- compilation --
error: chroot() not followed by chdir()
entry: main
/tmp/foo.c:4: "Init" 1
/tmp/foo.c:5: "Init" 1 5
/tmp/foo.c:6: "After chroot" 1 12
/tmp/foo.c:4: "Error" 1

```

The trace file "foo.1" consists of the error description of the found error, the entry-function leading to this error and then the trace from the entry to the error, stating the line-numbers and the corresponding state the FSA changed to in that line. This trace is written in the same format as a standard compilation log, thus it can be loaded e.g. into vim or Emacs.

The HTML output is located in the folder \$OUTPUTDIR/HTMLtrace/

### Automation:

MOPS provides an integration into build processes by providing a wrapper for gcc that does the checking while compiling the program. This is done by embedding the computed control flow graphs into the object files and checking them against the property FSA on generation of the executable by also wrapping the linker. The MOPS integration even succeeds on bigger software that is usually compiled with "./configure ; make". If some software does not work with the provided wrapper, you can of course run the components of MOPS manually or integrate it into other build processes. See the manual in the directory "doc" in the source tree for details.

### Quick Benchmark:

The runtime of MOPS is dependant on the complexity of the property FSA, but for a quick overview, some overall duration for a complete build process with checking for unsafe tempfile-handling is listed, along with build-time without MOPS:

|   |                      |
|---|----------------------|
| Building boa-0.94.14rc21 (~11000 LOC):  | 4 seconds            |
| Building and checking boa-0.94.14rc21:  | 21 seconds           |
| Building openssh-3.5p1 (~64400 LOC):    | 31 seconds           |
| Building and checking openssh-3.5p:     | 4 minutes 53 seconds |
| Building sendmail-8.13.5 (~128000 LOC): | 45 seconds           |
| Building and checking sendmail-8.13.5:  | 4 minutes 40 seconds |

Other benchmarks with different properties checked can be found in the paper "Model checking one million lines of C code".

## Further Reading

Yves Younan, Wouter Joosen and Frank Piessens. Code injection in C and C++: A Survey of Vulnerabilities and Countermeasures. Technical Report CW386, Departement Computerwetenschappen, Katholieke Universiteit Leuven, July 2004.  
<http://fort-knox.org/CW386.pdf>

J. Wilander, M. Kamkar. A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention. In Proceedings of the 10th Network and Distributed System Security Symposium, San Diego, CA, February 2003, pp. 149-162  
[http://www.ida.liu.se/~johwi/research\\_publications/paper\\_ndss2003\\_john\\_wilander.pdf](http://www.ida.liu.se/~johwi/research_publications/paper_ndss2003_john_wilander.pdf)

J. Wilander and M. Kamkar. A comparison of publicly available tools for static intrusion detection. In Proceedings of the Nordic Workshop on Secure IT Systems, pages 68--84, November 2002.  
[http://www.ida.liu.se/~johwi/research\\_publications/paper\\_nordsec2002\\_john\\_wilander.pdf](http://www.ida.liu.se/~johwi/research_publications/paper_nordsec2002_john_wilander.pdf)

Peter Silberman, Richard Johnson (iDefense). A Comparison of Buffer Overflow Prevention Implementations and Weaknesses. Black Hat USA 2004 Briefings & Training, 2004  
<http://www.blackhat.com/presentations/bh-usa-04/bh-us-04-silberman/bh-us-04-silberman-paper.pdf>