NAME

flawfinder − lexically find potential security flaws ("hits") in source code

SYNOPSIS

flawfinder [**−−help**|**−h**] [**−−version**] [**−−listrules**]
[**−−allowlink**] [**−−followdotdir**] [**−−nolink**]
[**−−patch=***filename*|**−P** *filename*]
[**−−inputs**|**−I**] [**−−minlevel=***X* | **−m** *X*] [**−−falsepositive**|**−F**]
[**−−neverignore**|**−n**]
[**−−regex=***PATTERN* | **−e** *PATTERN*]
[**−−context**|**−c**] [**−−columns**|**−C**] [**−−csv**] [**−−dataonly**|**−D**] [**−−html**|**−H**] [**−−immediate**|**-i**] [**−−singleline**|**−S**] [**−−omittime**] [**−−quiet**|**−Q**]
[**−−loadhitlist=***F*] [**−−savehitlist=***F*] [**−−diffhitlist=***F*]
[**−−**] *[source code file or source root directory]+*

DESCRIPTION

Flawfinder searches through C/C++ source code looking for potential security flaws. To run flawfinder, simply give flawfinder a list of directories or files. For each directory given, all files that have C/C++ filename extensions in that directory (and its subdirectories, recursively) will be examined. Thus, for most projects, simply give flawfinder the name of the source code’s topmost directory (use ‘‘.’’ for the current directory), and flawfinder will examine all of the project’s C/C++ source code. Flawfinder does *not* require that you be able to build your software, so it can be used even with incomplete source code. If you only want to have *changes* reviewed, save a unified diff of those changes (created by GNU "diff -u" or "svn diff" or "git diff") in a patch file and use the −−patch (−P) option.

Flawfinder will produce a list of ‘‘hits’’ (potential security flaws, also called findings), sorted by risk; the riskiest hits are shown first. The risk level is shown inside square brackets and varies from 0, very little risk, to 5, great risk. This risk level depends not only on the function, but on the values of the parameters of the function. For example, constant strings are often less risky than fully variable strings in many contexts, and in those contexts the hit will have a lower risk level. Flawfinder knows about gettext (a common library for internationalized programs) and will treat constant strings passed through gettext as though they were constant strings; this reduces the number of false hits in internationalized programs. Flawfinder will do the same sort of thing with _T() and _TEXT(), common Microsoft macros for handling internationalized programs. Flawfinder correctly ignores text inside comments and strings. Normally flawfinder shows all hits with a risk level of at least 1, but you can use the −−minlevel option to show only hits with higher risk levels if you wish. Hit descriptions also note the relevant Common Weakness Enumeration (CWE) identifier(s) in parentheses, as discussed below. Flawfinder is officially CWE-Compatible. Hit descriptions with "[MS-banned]" indicate functions that are in the banned list of functions released by Microsoft; see http://msdn.microsoft.com/en-us/library/bb288454.aspx for more information about banned functions.

Not every hit (aka finding) is actually a security vulnerability, and not every security vulnerability is necessarily found. Nevertheless, flawfinder can be an aid in finding and removing security vulnerabilities. A common way to use flawfinder is to first apply flawfinder to a set of source code and examine the highest-risk items. Then, use −−inputs to examine the input locations, and check to make sure that only legal and safe input values are accepted from untrusted users.

Once you’ve audited a program, you can mark source code lines that are actually fine but cause spurious warnings so that flawfinder will stop complaining about them. To mark a line so that these warnings are suppressed, put a specially-formatted comment either on the same line (after the source code) or all by itself in the previous line. The comment must have one of the two following formats:

- // Flawfinder: ignore
- /* Flawfinder: ignore */

For compatibility’s sake, you can replace "Flawfinder:" with "ITS4:" or "RATS:" in these specially-formatted comments. Since it’s possible that such lines are wrong, you can use the −−neverignore option, which causes flawfinder to never ignore any line no matter what the comment directives say (more confusingly, −−neverignore ignores the ignores).

Flawfinder uses an internal database called the ‘‘ruleset’’; the ruleset identifies functions that are common causes of security flaws. The standard ruleset includes a large number of different potential problems, including both general issues that can impact any C/C++ program, as well as a number of specific Unix-like and Windows functions that are especially problematic. The −−listrules option reports the list of current rules and their default risk levels. As noted above, every potential security flaw found in a given source code file (matching an entry in the ruleset) is called a ‘‘hit,’’ and the set of hits found during any particular run of the program is called the ‘‘hitlist.’’ Hitlists can be saved (using −−savehitlist), reloaded back for redisplay (using −−loadhitlist), and you can show only the hits that are different from another run (using −−diffhitlist).

Flawfinder is a simple tool, leading to some fundamental pros and cons. Flawfinder works by doing simple lexical tokenization (skipping comments and correctly tokenizing strings), looking for token matches to the database (particularly to find function calls). Flawfinder is thus similar to RATS and ITS4, which also use simple lexical tokenization. Flawfinder then examines the text of the function parameters to estimate risk. Unlike tools such as splint, gcc’s warning flags, and clang, flawfinder does *not* use or have access to information about control flow, data flow, or data types when searching for potential vulnerabilities or estimating the level of risk. Thus, flawfinder will necessarily produce many false positives for vulnerabilities and fail to report many vulnerabilities. On the other hand, flawfinder can find vulnerabilities in programs that cannot be built or cannot be linked. It can often work with programs that cannot even be compiled (at least by the reviewer’s tools). Flawfinder also doesn’t get as confused by macro definitions and other oddities that more sophisticated tools have trouble with. Flawfinder can also be useful as a simple introduction to static analysis tools in general, since it is easy to start using and easy to understand.

Any filename given on the command line will be examined (even if it doesn’t have a usual C/C++ filename extension); thus you can force flawfinder to examine any specific files you desire. While searching directories recursively, flawfinder only opens and examines regular files that have C/C++ filename extensions. Flawfinder presumes that files are C/C++ files if they have the extensions ".c", ".h", ".ec", ".ecp", ".pgc", ".C", ".cpp", ".CPP", ".cxx", ".cc", ".CC", ".pcc", ".hpp", or ".H". The filename ‘‘−’’ means the standard input. To prevent security problems, special files (such as device special files and named pipes) are always skipped, and by default symbolic links are skipped (the −−allowlink option follows symbolic links).

After the list of hits is a brief summary of the results (use -D to remove this information). It will show the number of hits, lines analyzed (as reported by wc −l), and the physical source lines of code (SLOC) analyzed. A physical SLOC is a non-blank, non-comment line. It will then show the number of hits at each level; note that there will never be a hit at a level lower than minlevel (1 by default). Thus, "[0] 0 [1] 9" means that at level 0 there were 0 hits reported, and at level 1 there were 9 hits reported. It will next show the number of hits at a given level or larger (so level 3+ has the sum of the number of hits at level 3, 4, and 5). Thus, an entry of "[0+] 37" shows that at level 0 or higher there were 37 hits (the 0+ entry will always be the same as the "hits" number above). Hits per KSLOC is next shown; this is each of the "level or higher" values multiplied by 1000 and divided by the physical SLOC. If symlinks were skipped, the count of those is reported. If hits were suppressed (using the "ignore" directive in source code comments as described above), the number suppressed is reported. The minimum risk level to be included in the report is displayed; by default this is 1 (use −−minlevel to change this). The summary ends with important reminders: Not every hit is necessarily a security vulnerability, and there may be other security vulnerabilities not reported by the tool.

Flawfinder is released under the GNU GPL license version 2 or later (GPLv2+).

Flawfinder works similarly to another program, ITS4, which is not fully open source software (as defined in the Open Source Definition) nor free software (as defined by the Free Software Foundation). The author of Flawfinder has never seen ITS4’s source code. Flawfinder is similar in many ways to RATS, if you are familiar with RATS.

BRIEF TUTORIAL

Here’s a brief example of how flawfinder might be used. Imagine that you have the C/C++ source code for some program named xyzzy (which you may or may not have written), and you’re searching for security vulnerabilities (so you can fix them before customers encounter the vulnerabilities). For this tutorial, I’ll assume that you're using a Unix-like system, such as Linux, OpenBSD, or MacOS X.

If the source code is in a subdirectory named xyzzy, you would probably start by opening a text window and using flawfinder's default settings, to analyze the program and report a prioritized list of potential security vulnerabilities (the ''less'' just makes sure the results stay on the screen):

    flawfinder xyzzy | less

At this point, you will see a large number of entries. Each entry has a filename, a colon, a line number, a risk level in brackets (where 5 is the most risky), a category, the name of the function, and a description of why flawfinder thinks the line is a vulnerability. Flawfinder normally sorts by risk level, showing the riskiest items first; if you have limited time, it's probably best to start working on the riskiest items and continue until you run out of time. If you want to limit the display to risks with only a certain risk level or higher, use the −−minlevel option. If you're getting an extraordinary number of false positives because variable names look like dangerous function names, use the −F option to remove reports about them. If you don't understand the error message, please see documents such as the *Writing Secure Programs HOWTO* ⟨http://www.dwheeler.com/secure-programs⟩ at http://www.dwheeler.com/secure-programs which provides more information on writing secure programs.

Once you identify the problem and understand it, you can fix it. Occasionally you may want to re-do the analysis, both because the line numbers will change *and* to make sure that the new code doesn't introduce yet a different vulnerability.

If you've determined that some line isn't really a problem, and you're sure of it, you can insert just before or on the offending line a comment like

    /* Flawfinder: ignore */

to keep them from showing up in the output.

Once you've done that, you should go back and search for the program's inputs, to make sure that the program strongly filters any of its untrusted inputs. Flawfinder can identify many program inputs by using the −−inputs option, like this:

    flawfinder −−inputs xyzzy

Flawfinder can integrate well with text editors and integrated development environments; see the examples for more information.

Flawfinder includes many other options, including ones to create HTML versions of the output (useful for prettier displays). The next section describes those options in more detail.

OPTIONS

Flawfinder has a number of options, which can be grouped into options that control its own documentation, select input data, select which hits to display, select the output format, and perform hitlist management. The commonly-used flawfinder options support the standard option syntax defined in the POSIX (Issue 7, 2013 Edition) section ''Utility Conventions''. Flawfinder also supports the GNU long options (double-dash options of form −−*option*) as defined in the *GNU C Library Reference Manual* ''Program Argument Syntax Conventions'' and *GNU Coding Standards* ''Standards for Command Line Interfaces''. Long option arguments can be provided as ''--name=value'' or ''-name value''. All options can be accessed using the more readable GNU long option conventions; some less commonly used options can *only* be accessed using long option conventions.

Documentation

−−help

−h Show usage (help) information.

−−version Shows (just) the version number and exits.

−−listrules List the terms (tokens) that trigger further examination, their default risk level, and the default warning (including the CWE identifier(s), if applicable), all tab-separated. The terms are primarily names of potentially-dangerous functions. Note that the reported risk level and warning for some specific code may be different than the default, depending on how the term is used. Combine with −D if you do not want the usual header. Flawfinder version 1.29 changed the separator from spaces to tabs, and added the default warning field.

Selecting Input Data

−−allowlink Allow the use of symbolic links; normally symbolic links are skipped. Don't use this option if you're analyzing code by others; attackers could do many things to cause problems for an analysis with this option enabled. For example, an attacker could insert symbolic links to files such as /etc/passwd (leaking information about the file) or create a circular loop, which would cause flawfinder to run "forever". Another problem with enabling this option is that if the same file is referenced multiple times using symbolic links, it will be analyzed multiple times (and thus reported multiple times). Note that flawfinder already includes some protection against symbolic links to special file types such as device file types (e.g., /dev/zero or C:\mystuff\com1). Note that for flawfinder version 1.01 and before, this was the default.

−−followdotdir
Enter directories whose names begin with ".". Normally such directories are ignored, since they normally include version control private data (such as .git/ or .svn/), build metadata (such as .makepp), configuration information, and so on.

−−nolink Ignored. Historically this disabled following symbolic links; this behavior is now the default.

−−patch=*patchfile*

−P *patchfile* Examine the selected files or directories, but only report hits in lines that are added or modified as described in the given patch file. The patch file must be in a recognized unified diff format (e.g., the output of GNU "diff -u old new", "svn diff", or "git diff [commit]"). Flawfinder assumes that the patch has already been applied to the files. The patch file can also include changes to irrelevant files (they will simply be ignored). The line numbers given in the patch file are used to determine which lines were changed, so if you have modified the files since the patch file was created, regenerate the patch file first. Beware that the file names of the new files given in the patch file must match exactly, including upper/lower case, path prefix, and directory separator (\ vs. /). Only unified diff format is accepted (GNU diff, svn diff, and git diff output is okay); if you have a different format, again regenerate it first. Only hits that occur on resultant changed lines, or immediately above and below them, are reported. This option implies −−neverignore. **Warning**: Do *not* pass a patch file without the **−P**, because flawfinder will then try to treat the file as a source file. This will often work, but the line numbers will be relative to the beginning of the patch file, not the positions in the source code. Note that you **must** also provide the actual files to analyze, and not just the patch file; when using **−P** files are only reported if they are both listed in the patch and also listed (directly or indirectly) in the list of files to analyze.

Selecting Hits to Display

−−inputs

−I Show only functions that obtain data from outside the program; this also sets minlevel to 0.

−−minlevel=*X*

-m *X* Set minimum risk level to X for inclusion in hitlist. This can be from 0 (“no risk”) to 5 (“maximum risk”); the default is 1.

−−falsepositive

−F Do not include hits that are likely to be false positives. Currently, this means that function names are ignored if they’re not followed by "(", and that declarations of character arrays aren’t noted. Thus, if you have use a variable named "access" everywhere, this will eliminate references to this ordinary variable. This isn’t the default, because this also increases the likelihood of missing important hits; in particular, function names in #define clauses and calls through function pointers will be missed.

−−neverignore

-n Never ignore security issues, even if they have an “ignore” directive in a comment.

−−regexp=*PATTERN*

-e *PATTERN*
Only report hits with text that matches the regular expression pattern PATTERN. For example, to only report hits containing the text "CWE-120", use “−−regex CWE-120”. These option flag names are the same as grep.

Selecting Output Format

−−columns

−C Show the column number (as well as the file name and line number) of each hit; this is shown after the line number by adding a colon and the column number in the line (the first character in a line is column number 1). This is useful for editors that can jump to specific columns, or for integrating with other tools (such as those to further filter out false positives).

−−context

−c Show context, i.e., the line having the "hit"/potential flaw. By default the line is shown immediately after the warning.

−−csv Generate output in comma-separated-value (CSV) format. This is the recommended format for sending to other tools for processing. It will always generate a header row, followed by 0 or more data rows (one data row for each hit). Selecting this option automatically enables −−quiet and −−dataonly. The headers are mostly self-explanatory. "File" is the filename, "Line" is the line number, "Column" is the column (starting from 1), "Level" is the risk level (0-5, 5 is riskiest), "Category" is the general flawfinder category, "Name" is the name of the triggering rule, "Warning" is text explaining why it is a hit (finding), "Suggestion" is text suggesting how it might be fixed, "Note" is other explanatory notes, "CWEs" is the list of one or more CWEs, "Context" is the source code line triggering the hit, and "Fingerprint" is the SHA-256 hash of the context once its leading and trailing whitespace have been removed (the fingerprint may help detect and eliminate later duplications). If you use Python3, the hash is of the context when encoded as UTF-8.

−−dataonly

−D Don't display the header and footer. Use this along with −−quiet to see just the data itself.

−−html

−H Format the output as HTML instead of as simple text.

−−immediate

-i Immediately display hits (don't just wait until the end).

−−singleline

-S Display as single line of text output for each hit. Useful for interacting with compilation tools.

−−omittime Omit timing information. This is useful for regression tests of flawfinder itself, so that the output doesn't vary depending on how long the analysis takes.

−−quiet

−Q Don't display status information (i.e., which files are being examined) while the analysis is going on.

Hitlist Management

−−savehitlist=*F*

Save all resulting hits (the "hitlist") to F.

−−loadhitlist=*F*

Load the hitlist from F instead of analyzing source programs. Warning: Do *not* load hitlists from untrusted sources (for security reasons). These are internally implemented using Python's "pickle" facility, which trusts the input. Note that stored hitlists often cannot be read when using an older version of Python, in particular, if savehitlist was used but flawfinder was run using Python 3, the hitlist can't be loaded by running flawfinder with Python 2.

−−diffhitlist=*F*

Show only hits (loaded or analyzed) not in F. F was presumably created previously using −−savehitlist. Warning: Do *not* diff hitlists from untrusted sources (for security reasons). If the −−loadhitlist option is not provided, this will show the hits in the analyzed source code files that were not previously stored in F. If used along with −−loadhitlist, this will show the hits in the loaded hitlist not in F. The difference algorithm is conservative; hits are only considered the ''same'' if they have the same filename, line number, column position, function name, and risk level.

Character Encoding

Flawfinder presumes that the character encoding your system uses is also the character encoding used by your source files. Even if this isn't correct, if you run flawfinder with Python 2 these non-conformities often do not impact processing in practice.

However, if you run flawfinder with Python 3, this can be a problem. Python 3 wants the world to always use encodings perfectly correctly, everywhere, even though the world often doesn't care what Python 3 wants. This is a problem even if the non-conforming text is in comments or strings (where it often doesn't matter). Python 3 fails to provide useful built-ins to deal with the messiness of the real world, so it's non-trivial to deal with this problem without depending on external libraries (which we're trying to avoid).

A symptom of this problem is if you run flawfinder and you see an error message like this:

UnicodeDecodeError: 'utf-8' codec can't decode byte ... in position ...: invalid continuation byte

If this happens to you, there are several options.

The first option is to convert the encoding of the files to be analyzed so that it's a single encoding (usually the system encoding). For example, the program "iconv" can be used to convert encodings. This works well if some files have one encoding, and some have another, but they are consistent within a single file. If the files have encoding errors, you'll have to fix them. I strongly recommend using the UTF-8 encoding for any source code; if you do that, many problems disappear.

The second option is to tell flawfinder what the encoding of the files is. E.G., you can set the LANG environment variable. You can set PYTHONIOENCODING to the encoding you want your output to be in, if that's different. This in theory would work well, but I haven't had much success with this.

The third option is to run flawfinder using Python 2 instead of Python 3. E.g., "python2 flawfinder ...".

EXAMPLES

Here are various examples of how to invoke flawfinder. The first examples show various simple command-line options. Flawfinder is designed to work well with text editors and integrated development environments, so the next sections show how to integrate flawfinder into vim and emacs.

Simple command-line options

flawfinder /usr/src/linux-3.16

Examine all the C/C++ files in the directory /usr/src/linux-3.16 and all its subdirectories (recursively), reporting on all hits found. By default flawfinder will skip symbolic links and directories with names that start with a period.

flawfinder −−minlevel=4 .

Examine all the C/C++ files in the current directory and its subdirectories (recursively); only report vulnerabilities level 4 and up (the two highest risk levels).

flawfinder −−inputs mydir

Examine all the C/C++ files in mydir and its subdirectories (recursively), and report functions that take inputs (so that you can ensure that they filter the inputs appropriately).

flawfinder −−neverignore mydir

Examine all the C/C++ files in the directory mydir and its subdirectories, including even the hits marked for ignoring in the code comments.

flawfinder −−csv .

Examine the current directory down (recursively), and report all hits in CSV format. This is the recommended form if you want to further process flawfinder output using other tools (such as data correlation tools).

flawfinder −QD mydir

Examine mydir and report only the actual results (removing the header and footer of the output). This form may be useful if the output will be piped into other tools for further analysis, though CSV format is probably the better choice in that case. The −C (−−columns) and −S (−−singleline) options can also be useful if you're piping the data into other tools.

flawfinder −QDSC mydir

Examine mydir, reporting only the actual results (no header or footer). Each hit is reported on one line, and column numbers are reported. This can be a useful command if you are feeding flawfinder output to other tools.

flawfinder −−quiet −−html −−context mydir > results.html

Examine all the C/C++ files in the directory mydir and its subdirectories, and produce an HTML formatted version of the results. Source code management systems (such as SourceForge and Savannah) might use a command like this.

flawfinder −−quiet −−savehitlist saved.hits *.[ch]

Examine all .c and .h files in the current directory. Don't report on the status of processing, and save the resulting hitlist (the set of all hits) in the file saved.hits.

flawfinder −−diffhitlist saved.hits *.[ch]

Examine all .c and .h files in the current directory, and show any hits that weren't already in the file saved.hits. This can be used to show only the ''new'' vulnerabilities in a modified program, if saved.hits was created from the older version of the program being analyzed.

flawfinder −−patch recent.patch .

Examine the current directory recursively, but only report lines that were changed or added in the already-applied patchfile named *recent.patch*.

flawfinder −−regex "CWE-120|CWE-126" src/

Examine directory *src* recursively, but only report hits where CWE-120 or CWE-126 apply.

Invoking from vim

The text editor vim includes a "quickfix" mechanism that works well with flawfinder, so that you can easily view the warning messages and jump to the relevant source code.

First, you need to invoke flawfinder to create a list of hits, and there are two ways to do this. The first way is to start flawfinder first, and then (using its output) invoke vim. The second way is to start (or continue to run) vim, and then invoke flawfinder (typically from inside vim).

For the first way, run flawfinder and store its output in some FLAWFILE (say "flawfile"), then invoke vim using its -q option, like this: "vim -q flawfile". The second way (starting flawfinder after starting vim) can be done a legion of ways. One is to invoke flawfinder using a shell command, ":!flawfinder-command > FLAWFILE", then follow that with the command ":cf FLAWFILE". Another way is to store the flawfinder command in your makefile (as, say, a pseudocommand like "flaw"), and then run ":make flaw".

In all these cases you need a command for flawfinder to run. A plausible command, which places each hit in its own line (-S) and removes headers and footers that would confuse it, is:

flawfinder −SQD .

You can now use various editing commands to view the results. The command ":cn" displays the next hit; ":cN" displays the previous hit, and ":cr" rewinds back to the first hit. ":copen" will open a window to show the current list of hits, called the "quickfix window"; ":cclose" will close the quickfix window. If the buffer in the used window has changed, and the error is in another file, jumping to the error will fail. You have to make sure the window contains a buffer which can be abandoned before trying to jump to a new file, say by saving the file; this prevents accidental data loss.

Invoking from emacs

The text editor / operating system emacs includes "grep mode" and "compile mode" mechanisms that work well with flawfinder, making it easy to view warning messages, jump to the relevant source code, and fix any problems you find.

First, you need to invoke flawfinder to create a list of warning messages. You can use "grep mode" or "compile mode" to create this list. Often "grep mode" is more convenient; it leaves compile mode untouched so you can easily recompile once you've changed something. However, if you want to jump to the exact column position of a hit, compile mode may be more convenient because emacs can use the column output of flawfinder to directly jump to the right location without any special configuration.

To use grep mode, enter the command "M-x grep" and then enter the needed flawfinder command. To use compile mode, enter the command "M-x compile" and enter the needed flawfinder command. This is a meta-key command, so you'll need to use the meta key for your keyboard (this is usually the ESC key). As with all emacs commands, you'll need to press RETURN after typing "grep" or "compile". So on many systems, the grep mode is invoked by typing ESC x g r e p RETURN.

You then need to enter a command, removing whatever was there before if necessary. A plausible command is:

flawfinder −SQDC .

This command makes every hit report a single line, which is much easier for tools to handle. The quiet and dataonly options remove the other status information not needed for use inside emacs. The trailing period means that the current directory and all descendents are searched for C/C++ code, and analyzed for flaws.

Once you've invoked flawfinder, you can use emacs to jump around in its results. The command C-x ` (Control-x backtick) visits the source code location for the next warning message. C-u C-x ` (control-u control-x backtick) restarts from the beginning. You can visit the source for any particular error message by moving to that hit message in the *compilation* buffer or *grep* buffer and typing the return key. (Technical note: in the compilation buffer, this invokes compile-goto-error.) You can also click the Mouse-2 button on the error message (you don't need to switch to the *compilation* buffer first).

If you want to use grep mode to jump to specific columns of a hit, you'll need to specially configure emacs to do this. To do this, modify the emacs variable "grep-regexp-alist". This variable tells Emacs how to parse output of a "grep" command, similar to the variable "compilation-error-regexp-alist" which lists various formats of compilation error messages.

Invoking from Integrated Development Environments (IDEs)

For (other) IDEs, consult your IDE's set of plug-ins.

COMMON WEAKNESS ENUMERATION (CWE)

The Common Weakness Enumeration (CWE) is ‘‘a formal list or dictionary of common software weaknesses that can occur in software's architecture, design, code or implementation that can lead to exploitable security vulnerabilities... created to serve as a common language for describing software security weaknesses’’ (http://cwe.mitre.org/about/faq.html). For more information on CWEs, see http://cwe.mitre.org.

Flawfinder supports the CWE and is officially CWE-Compatible. Hit descriptions typically include a relevant Common Weakness Enumeration (CWE) identifier in parentheses where there is known to be a relevant CWE. For example, many of the buffer-related hits mention CWE-120, the CWE identifier for ‘‘buffer copy without checking size of input’’ (aka ‘‘Classic Buffer Overflow’’). In a few cases more than one CWE identifier may be listed. The HTML report also includes hypertext links to the CWE definitions hosted at MITRE. In this way, flawfinder is designed to meet the CWE-Output requirement.

In some cases there are CWE mapping and usage challenges; here is how flawfinder handles them. If the same entry maps to multiple CWEs simultaneously, all the CWE mappings are listed as separated by commas. This often occurs with CWE-20, Improper Input Validation; thus the report "CWE-676, CWE-120" maps to two CWEs. In addition, flawfinder provides additional information for those who are are interested in the CWE/SANS top 25 list 2011 (http://cwe.mitre.org/top25/) when mappings are not directly to them.

Many people will want to search for specific CWEs in this top 25 list, such as CWE-120 (classic buffer overflow). The challenge is that some flawfinder hits map to a more general CWE that would include a top 25 item, while in some other cases hits map to a more specific vulnerability that is only a subset of a top 25 item. To resolve this, in some cases flawfinder will list a sequence of CWEs in the format "more-general/more-specific", where the CWE actually being mapped is followed by a "!". This is always done whenever a flaw is not mapped directly to a top 25 CWE, but the mapping is related to such a CWE. So "CWE-119!/CWE-120" means that the vulnerability is mapped to CWE-119 and that CWE-120 is a subset of CWE-119. In contrast, "CWE-362/CWE-367!" means that the hit is mapped to CWE-367, a subset of CWE-362. Note that this is a subtle syntax change from flawfinder version 1.31; in flawfinder version 1.31, the form "more-general:more-specific" meant what is now listed as "more-general!/more-specific", while "more-general/more-specific" meant "more-general/more-specific!". Tools can handle both the version 1.31 and the current format, if they wish, by noting that the older format did not use "!" at all (and thus this is easy to distinguish). These mapping mechanisms simplify searching for certain CWEs.

CWE version 2.7 (released June 23, 2014) was used for the mapping. The current CWE mappings select the most specific CWE the tool can determine. In theory, most CWE security elements (signatures/patterns that the tool searches for) could theoretically be mapped to CWE-676 (Use of Potentially Dangerous Function), but such a mapping would not be useful. Thus, more specific mappings were preferred where one could be found. Flawfinder is a lexical analysis tool; as a result, it is impractical for it to be more specific than the mappings currently implemented. This also means that it is unlikely to need much updating for map currency; it simply doesn't have enough information to refine to a detailed CWE level that CWE changes would typically affect. The list of CWE identifiers was generated automatically using "make show-cwes", so there is confidence that this list is correct. Please report CWE mapping problems as bugs if you find any.

Flawfinder may fail to find a vulnerability, even if flawfinder covers one of these CWE weaknesses. That said, flawfinder does find vulnerabilities listed by the CWEs it covers, and it will not report lines without those vulnerabilities in many cases. Thus, as required for any tool intending to be CWE compatible, flawfinder has a rate of false positives less than 100% and a rate of false negatives less than 100%. Flawfinder almost always reports whenever it finds a match to a CWE security element (a signature/pattern as defined in its database), though certain obscure constructs can cause it to fail (see BUGS below).

Flawfinder can report on the following CWEs (these are the CWEs that flawfinder covers; ‘‘*’’ marks those in the CWE/SANS top 25 list):

- CWE-20: Improper Input Validation
- CWE-22: Improper Limitation of a Pathname to a Restricted Directory (‘‘Path Traversal’’)
- CWE-78: Improper Neutralization of Special Elements used in an OS Command (‘‘OS Command Injection’’)*
- CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer (a parent of CWE-120*, so this is shown as CWE-119!/CWE-120)
- CWE-120: Buffer Copy without Checking Size of Input (‘‘Classic Buffer Overflow’’)*
- CWE-126: Buffer Over-read
- CWE-134: Uncontrolled Format String*
- CWE-190: Integer Overflow or Wraparound*
- CWE-250: Execution with Unnecessary Privileges
- CWE-327: Use of a Broken or Risky Cryptographic Algorithm*
- CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization (‘‘Race Condition’’)
- CWE-377: Insecure Temporary File
- CWE-676: Use of Potentially Dangerous Function*

- CWE-732: Incorrect Permission Assignment for Critical Resource*
- CWE-785: Use of Path Manipulation Function without Maximum-sized Buffer (child of CWE-120*, so this is shown as CWE-120/CWE-785)
- CWE-807: Reliance on Untrusted Inputs in a Security Decision*
- CWE-829: Inclusion of Functionality from Untrusted Control Sphere*

You can select a specific subset of CWEs to report by using the ‘‘−−regex’’ (-e) option. This option accepts a regular expression, so you can select multiple CWEs, e.g., ‘‘−−regex "CWE-120|CWE-126"’’. If you select multiple CWEs with ‘‘|’’ on a command line you will typically need to quote the parameters (since an unquoted ‘‘|’’ is the pipe symbol). Flawfinder is designed to meet the CWE-Searchable requirement.

If your goal is to report a subset of CWEs that are listed in a file, that can be achieved on a Unix-like system using the ‘‘−−regex’’ aka ‘‘−e’’ option. The file must be in regular expression format. For example, ‘‘flawfinder -e $(cat file1)’’ would report only hits that matched the pattern in ‘‘file1’’. If file1 contained ‘‘CWE-120|CWE-126’’ it would only report hits matching those CWEs.

A list of all CWE security elements (the signatures/patterns that flawfinder looks for) can be found by using the ‘‘−−listrules’’ option. Each line lists the signature token (typically a function name) that may lead to a hit, the default risk level, and the default warning (which includes the default CWE identifier). For most purposes this is also enough if you want to see what CWE security elements map to which CWEs, or the reverse. For example, to see the most of the signatures (function names) that map to CWE-327, without seeing the default risk level or detailed warning text, run ‘‘flawfinder −−listrules | grep CWE-327 | cut -f1’’. You can also see the tokens without a CWE mapping this way by running ‘‘flawfinder -D --listrules | grep -v CWE-’’. However, while −−listrules lists all CWE security elements, it only lists the default mappings from CWE security elements to CWE identifiers. It does not include the refinements that flawfinder applies (e.g., by examining function parameters).

If you want a detailed and exact mapping between the CWE security elements and CWE identifiers, the flawfinder source code (included in the distribution) is the best place for that information. This detailed information is primarily of interest to those few people who are trying to refine the CWE mappings of flawfinder or refine CWE in general. The source code documents the mapping between the security elements to the respective CWE identifiers, and is a single Python file. The ‘‘c_rules’’ dataset defines most rules, with reference to a function that may make further refinements. You can search the dataset for function names to see what CWE it generates by default; if first parameter is not ‘‘normal’’ then that is the name of a refining Python method that may select different CWEs (depending on additional information). Conversely, you can search for ‘‘CWE-number’’ and find what security elements (signatures or patterns) refer to that CWE identifier. For most people, this is much more than they need; most people just want to scan their source code to quickly find problems.

SECURITY

The whole point of this tool is to help find vulnerabilities so they can be fixed. However, developers and reviewers must know how to develop secure software to use this tool, because otherwise, *a fool with a tool is still a fool*. My book at http://www.dwheeler.com/secure-programs may help.

This tool should be, at most, a small part of a larger software development process designed to eliminate or reduce the impact of vulnerabilities. Developers and reviewers need know how to develop secure software, and they need to apply this knowledge to reduce the risks of vulnerabilities in the first place.

Different vulnerability-finding tools tend to find different vulnerabilities. Thus, you are best off using human review and a variety of tools. This tool can help find some vulnerabilities, but by no means all.

You should always analyze a *copy* of the source program being analyzed, not a directory that can be modified by a developer while flawfinder is performing the analysis. This is *especially* true if you don’t necessily trust a developer of the program being analyzed. If an attacker has control over the files while you’re analyzing them, the attacker could move files around or change their contents to prevent the exposure of a security problem (or create the impression of a problem where there is none). If you’re worried about malicious programmers you should do this anyway, because after analysis you'll need to verify that the code eventually run is the code you analyzed. Also, do not use the −−allowlink option in such cases; attackers could create malicious symbolic links to files outside of their source code area (such as /etc/passwd).

Source code management systems (like GitHub, SourceForge, and Savannah) definitely fall into this category; if you're maintaining one of those systems, first copy or extract the files into a separate directory (that can't be controlled by attackers) before running flawfinder or any other code analysis tool.

Note that flawfinder only opens regular files, directories, and (if requested) symbolic links; it will never open other kinds of files, even if a symbolic link is made to them. This counters attackers who insert unusual file types into the source code. However, this only works if the filesystem being analyzed can't be modified by an attacker during the analysis, as recommended above. This protection also doesn't work on Cygwin platforms, unfortunately.

Cygwin systems (Unix emulation on top of Windows) have an additional problem if flawfinder is used to analyze programs that the analyst cannot trust. The problem is due to a design flaw in Windows (that it inherits from MS-DOS). On Windows and MS-DOS, certain filenames (e.g., "com1") are automatically treated by the operating system as the names of peripherals, and this is true even when a full pathname is given. Yes, Windows and MS-DOS really are designed this badly. Flawfinder deals with this by checking what a filesystem object is, and then only opening directories and regular files (and symlinks if enabled). Unfortunately, this doesn't work on Cygwin; on at least some versions of Cygwin on some versions of Windows, merely trying to determine if a file is a device type can cause the program to hang. A workaround is to delete or rename any filenames that are interpreted as device names before performing the analysis. These so-called "reserved names" are CON, PRN, AUX, CLOCK$, NUL, COM1-COM9, and LPT1-LPT9, optionally followed by an extension (e.g., "com1.txt"), in any directory, and in any case (Windows is case-insensitive).

Do *not* load or diff hitlists from untrusted sources. They are implemented using the Python pickle module, and the pickle module is not intended to be secure against erroneous or maliciously constructed data. Stored hitlists are intended for later use by the same user who created the hitlist; in that context this restriction is not a problem.

BUGS

Flawfinder is based on simple text pattern matching, which is part of its fundamental design and not easily changed. This design approach leads to a number of fundamental limitations, e.g., a higher false positive rate, and is the underlying cause of most of the bugs listed here. On the positive side, flawfinder doesn't get confused by many complicated preprocessor sequences that other tools sometimes choke on; flawfinder can often handle code that cannot link, and sometimes cannot even compile or build.

Flawfinder is currently limited to C/C++. In addition, when analyzing C++ it focuses primarily on the C subset of C++. For example, flawfinder does not report on expressions like cin >> charbuf, where charbuf is a char array. That is because flawfinder doesn't have type information, and ">>" is safe with many other types; reporting on all ">>" would lead to too many false positives. That said, it's designed so that adding support for other languages should be easy where its text-based approach can usefully apply.

Flawfinder can be fooled by user-defined functions or method names that happen to be the same as those defined as "hits" in its database, and will often trigger on definitions (as well as uses) of functions with the same name. This is typically not a problem for C code. In C code, a function with the same name as a common library routine name often indicates that the developer is simply rewriting a common library routine with the same interface, say for portability's sake. C programs tend to avoid reusing the same name for a different purpose (since in C function names are global by default). There are reasonable odds that these rewritten routines will be vulnerable to the same kinds of misuse, and thus, reusing these rules is a reasonable approach. However, this can be a much more serious problem in C++ code which heavily uses classes and namespaces, since the same method name may have many different meanings. The −−falsepositive option can help somewhat in this case. If this is a serious problem, feel free to modify the program, or process the flawfinder output through other tools to remove the false positives.

Preprocessor commands embedded in the middle of a parameter list of a call can cause problems in parsing, in particular, if a string is opened and then closed multiple times using an #ifdef .. #else construct, flawfinder gets confused. Such constructs are bad style, and will confuse many other tools too. If you must analyze such files, rewrite those lines. Thankfully, these are quite rare.

Some complex or unusual constructs can mislead flawfinder. In particular, if a parameter begins with gettext(" and ends with), flawfinder will presume that the parameter of gettext is a constant. This means it will get confused by patterns like gettext("hi") + function("bye"). In practice, this doesn't seem to be a problem; gettext() is usually wrapped around the entire parameter.

The routine to detect statically defined character arrays uses simple text matching; some complicated expressions can cause it to trigger or not trigger unexpectedly.

Flawfinder looks for specific patterns known to be common mistakes. Flawfinder (or any tool like it) is not a good tool for finding intentionally malicious code (e.g., Trojan horses); malicious programmers can easily insert code that would not be detected by this kind of tool.

Flawfinder looks for specific patterns known to be common mistakes in application code. Thus, it is likely to be less effective analyzing programs that aren't application-layer code (e.g., kernel code or self-hosting code). The techniques may still be useful; feel free to replace the database if your situation is significantly different from normal.

Flawfinder's default output format (filename:linenumber, followed optionally by a :columnnumber) can be misunderstood if any source files have very weird filenames. Filenames embedding a newline/linefeed character will cause odd breaks, and filenames including colon (:) are likely to be misunderstood. This is especially important if flawfinder's output is being used by other tools, such as filters or text editors. If you are using flawfinder's output in other tools, consider using its CSV format instead (which can handle this). If you're looking at new code, examine the files for such characters. It's incredibly unwise to have such filenames anyway; many tools can't handle such filenames at all. Newline and linefeed are often used as internal data delimeters. The colon is often used as special characters in filesystems: MacOS uses it as a directory separator, Windows/MS-DOS uses it to identify drive letters, Windows/MS-DOS inconsistently uses it to identify special devices like CON:, and applications on many platforms use the colon to identify URIs/URLs. Filenames including spaces and/or tabs don't cause problems for flawfinder, though note that other tools might have problems with them.

Flawfinder is not internationalized, so it currently does not support localization.

In general, flawfinder attempts to err on the side of caution; it tends to report hits, so that they can be examined further, instead of silently ignoring them. Thus, flawfinder prefers to have false positives (reports that turn out to not be problems) rather than false negatives (failures to report security vulnerabilities). But this is a generality; flawfinder uses simplistic heuristics and simply can't get everything "right".

Security vulnerabilities might not be identified as such by flawfinder, and conversely, some hits aren't really security vulnerabilities. This is true for all static security scanners, and is especially true for tools like flawfinder that use a simple lexical analysis and pattern analysis to identify potential vulnerabilities. Still, it can serve as a useful aid for humans, helping to identify useful places to examine further, and that's the point of this simple tool.

SEE ALSO

See the flawfinder website at http://www.dwheeler.com/flawfinder. You should also see the *Secure Programming HOWTO* at *http://www.dwheeler.com/secure-programs*.

AUTHOR

David A. Wheeler (dwheeler@dwheeler.com).