NAME

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

SYNOPSIS

flawfinder [**−−help**] [**−−version**] [**−−allowlink**] [**−−inputs**|**−I**] [**−−minlevel=***X* | **-m** *X*] [**−−falsepositive**|**−F**] [**−−neverignore**|**−n**] [**−−context**|**−c**] [**−−columns**|**−C**] [**−−dataonly**|**−D**] [**−−html**] [**−−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 will produce a list of ‘‘hits’’ (potential security flaws), 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 most 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.

Not every hit 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 */

Note that, 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 comments say. Thus, responses that would otherwise be ignored would be included (or, more confusingly, −−neverignore ignores the ignores). This comment syntax is actually a more general syntax for special directives to flawfinder, but currently only ignoring lines is supported.

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. 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).

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,

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 intentionally 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.

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 a large number of entries; each entry begins with 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 for Linux and Unix HOWTO* 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 which hits to display, select the output format, and perform hitlist management.

Documentation

−−help Show usage (help) information.

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

Selecting Hits to Display

−−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.

−−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.

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.

−−dataonly

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

−−html 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.

−−diffhitlist=*F*

Show only hits (loaded or analyzed) not in F. F was presumably created previously using −−savehitlist. 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.

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-2.4.12

Examine all the C/C++ files in the directory /usr/src/linux-2.4.12 and all its subdirectories (recursively), reporting on all hits found.

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 -QD mydir

Examine mydir and report only the actual results (removing the header and footer of the output). This form is useful if the output will be piped into other tools for further analysis. The −C (−−columns) and −S (−−singleline) options can also be useful if you’re piping the data into 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.

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 (when using the mouse 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.

SECURITY

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 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 the analyzer cannot trust 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).

BUGS

Flawfinder is currently limited to C/C++. It's designed so that adding support for other languages should be easy.

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 because flawfinder is based on text pattern matching, which is part of its fundamental design and not easily changed. This isn't as much of a problem for C code, but it can be more of a problem for some C++ code which heavily uses classes and namespaces. On the positive side, flawfinder doesn't get confused by many complicated preprocessor sequences that other tools sometimes choke on. Also, having the same name as a common library routine name can indicate that the developer is simply rewriting a common library routine, say for portability's sake. Thus, there are reasonable odds that these rewritten routines will be vulnerable to the same kinds of misuse. The −−falsepositive option can help somewhat. 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.

The routine to detect statically defined character arrays uses simple text matching; some complicated expresions 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 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'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.

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 (failure to report on a security vulnerability). 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, especially those like flawfinder that use a simple pattern-based approach to identifying problems. 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 tool.

SEE ALSO

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

AUTHOR

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