Static Analyzers: Seat Belts for Your Code

Paul E. Black | US National Institute of Standards and Technology

Just as seat belt use is widespread, static analysis should be part of ethical software development. Because security must be designed in, static analysis should occur early in software development to reduce vulnerabilities.

Nobody plans to crash while driving, but we still wear seat belts. Similarly, nobody wants to put bugs in code, but static analyzers can save code from some catastrophes. Because tools can check millions of LOC quickly and repeatably, it seems perfectly reasonable that an ethical software development process should include automated analysis tools.

Here, I look at how the Static Analysis Tool Exposition (SATE) is trying to create a better understanding of static-analysis tools, encourage their improvement, and increase their use.

SATE

In 2007, William Pugh called for large sets of results of different static-analysis tools running on the same software.[1] In response, the Software Assurance Metrics And Tool Evaluation project (SAMATE; http://samate.nist.gov) began SATE.[2] My SAMATE colleagues and I based SATE on the Text Retrieval Conference[3]—a successful program that had been growing and adapting for some 17 years.

To being SATE, we chose test cases consisting of open source production programs of moderate size that had security-relevant aspects. We limited the programming languages to C and Java, owing to their wide use and the number of static-analysis tools that handle them. Participating toolmaker teams ran their tools on the test cases and returned the results to us for analysis. Our plan was to approximate ground truth by correlating warnings. We organized a workshop at which everyone could share their experience and learn from each other. We published the data more than six months later to give developers time to address any urgent concerns that might arise.[4]

Useful interpretation of this data isn't simply a matter of, say, comparing the average number of warnings each tool produces. Simplistic bug counting isn't justified—in fact, bug counting is far more complicated than we could have imagined, as I explain later.

Although we made many improvements in the protocol's details, all SATEs have followed the same general steps. The organizers choose test cases, the teams run their tools, the organizers analyze the results and hold an experience workshop, and the organizers release the reports and data.

Improvements and Changes to SATE

We weren't prepared for the huge number and diversity of warnings that came from the submitted results. Checking all warnings was completely impractical. In

 Copublished by the IEEE Computer and Reliability Societies

subsequent SATEs, we selected stratified random samples of warnings to analyze, commissioned expert security analysis of some test cases, and correlated warnings with the experts' findings. However, we still weren't sure this constituted a good measure of program security.

Why not use sound static analyzers—for example, those using abstract interpretation[5]—to establish the ground truth? For the classes of errors that static analyzers handle, they can prove the complete absence of such errors. However, there are many vulnerability classes they aren't programmed to handle. Even a proof that a program is a perfect implementation of a specification doesn't guarantee the absence of flaws in the specification. Typically, sound static analyzers can't process huge programs. Also, the number of bugs doesn't necessarily correspond with a program's security: a program with a few serious bugs is less secure than a program with many trivial bugs.

In SATE 2010, we chose some test cases based on known software vulnerabilities listed in CVE (the Common Vulnerabilities and Exposures list; http://cve.mitre.org). By sifting through information for CVE and widely used programs, we identified programs that had many security-relevant vulnerabilities at some time. We chose programs that had many more vulnerabilities than our previous test cases from 2008 and 2009 had.[6] For each program, we looked for an older version that had many of the vulnerabilities and a newer version in which they were fixed. To attribute locations to a fix, a related data-flow path, or a *sink* (that is, where the data leaves the program), we examined descriptions and references in CVE, bug-tracking and version control logs, and vulnerability notices. We also compared the source code of older versions with that of the fixed versions. During analysis, we searched for tool warnings that correlated with the CVE vulnerabilities.

Participating teams pointed out that setting up the environment to compile large programs is time-consuming. This is especially burdensome because SATE asked the teams to handle several different programs. Toolmakers told us that because they're familiar with the resources their tools need, installing their tools in an environment was easier than building a specific environment for the test cases. To make it easier for teams, in SATE IV (2012), we created a virtual machine with an appropriate environment for each test case.

In SATE IV, we added a track for PHP programs. We also added approximately 60,000 small, engineered C/C++ and Java programs as test cases. These programs, which compile in Linux, constitute the Juliet test suite, part of the SAMATE Reference Dataset (see the sidebar).

The SAMATE Reference Dataset

We plan to use Static Analysis Tool Exposition results to develop reference sets of programs to serve as benchmarks for static analyzers. As one step, we created the SAMATE Reference Dataset (SRD), a collection of more than 61,000 programs with known weaknesses. The SRD, which is publicly accessible (http://samate.nist.gov/SRD), contains programs mostly in C, C++, and Java. Most of them are small, engineered programs, although a few are extractions from open source applications. Each has some amount of useful information such as location and type of weakness. We'll draw some of the benchmark programs from the SRD.

Who Needs Graphical Reporting?

To automatically compare warnings, we had teams submit their reports in a simple XML format comprising

- identifiers;
- one or more traces of locations, such as line number and file path;
- the weakness type, which consisted of a name and, optionally, a Common Weakness Enumeration (CWE; http://cwe.mitre.org) number;
- the severity and likelihood;
- human evaluation; and
- an optional container for whatever constituted the original tool output.

While analyzing warnings, we found that using only the location and weakness type often made it difficult to verify or refute the presence of a weakness and its severity. Many tools produce rich, varied sets of information—such as variable and function names, the execution path as a code slice, and conditions—and display them graphically. With the toolmakers' generous help, we could sometimes consult this information to understand warnings. We discovered that a GUI and the rich information it presents often help users efficiently and correctly understand warnings and potential effects.

What We Learned

When we began SATE, we thought a warning simply indicated that the tool either found an actual bug or the tool made a mistake; "true" or "false" would suffice. That was far too simplistic, especially in a largely context-free situation such as SATE.

Some weaknesses, such as leftover debug code (CWE-489; http://cwe.mitre.org/data/definitions/489.html), aren't discernible without application knowledge. That is, you can't inspect the program and know for sure in every case whether a block of code is leftover from debugging or provides essential application functionality. The tool might reveal outgoing hard-coded password weaknesses (CWE-259) if it knows

```
while ((numberShadow /= sequenceSize) > 0)
  letters[lettersSize - ++length] =
      sequence[numberShadow % sequenceSize - 1];

while ((numberShadow /= sequenceSize) > 0)
{
  --numberShadow;
  letters[lettersSize - ++length] =
      sequence[numberShadow % sequenceSize];
}
```

Figure 1. Pieces of code from CVE-2010-1773 showing a chain of weaknesses. The first piece subtracts one, which is off by one in some cases, which leads to an out-of-bounds read, which leads to information disclosure. The second correctly decrements `numberShadow`.

widely used libraries, frameworks, and applications—for example, SQL databases. But does a classic buffer overflow (CWE-120) really matter if it might occur only at start-up while reading the configuration file with a keyword over 10,000 characters long? Should a tool report an OS command injection (CWE-78) that's clearly in dead code (which maintenance changes might activate someday) or in a function that's always called with safe parameters? Some warnings (for example, "strncpy() may not null-terminate" or "use of inherently dangerous function CWE-242") are absolutely correct as worded but could be utterly insignificant.

So, warnings have nuances beyond what a simple "true" or "false" can express. In 2010, we used five categories: true-security, true-quality, true-insignificant, unknown, and not a weakness. For example, true-quality means the warning is true but relates to code quality, not security. Unlike SATE's attempt to determine absolute truth, developers use tools to decide what code, if any, to change—not to decide absolutely whether an attack is possible.

Users should tune their tools by filtering outputs to remove low-value warnings, turning off certain checks, or turning on others. For instance, a repeated caution that "strncpy() may not null-terminate" might be more distracting than helpful, even though it's true. Turning off or filtering out classes of warnings that have little value for one development style or application area generally gives developers more useful warnings. Many tools let users write their own checks, which might help, depending on the coding style and policies.

Programming style can make it easier or harder for tools to find weaknesses. Vadim Okun and his colleagues gave an example: "In Nagios [an application for host, service, and network monitoring], the return value of malloc, strdup, or other memory allocation functions is not checked for NULL immediately; instead, it is checked for NULL before each use."[4] Checking for null at use requires nonlocal analysis, which makes it more difficult for a tool or a human to be sure the function return is handled properly.

Most Bugs Aren't Distinct

We wanted to match the same warnings from different tools to get an idea of their overlap. Simplistic matching assumes that each bug is distinct, but they're not. Consider the two segments of code from CVE-2010-1773 in Figure 1: the first is wrong; the other is fixed. The variable `numberShadow` should be decremented instead of subtracting one from the result of the modulo operation (denoted as %). An off-by-one error (CWE-193) led to an out-of-bounds read (CWE-125), which led to information disclosure (CWE-200). This bug is an example of a chain of several weaknesses. One tool might report CWE-125, whereas another reports CWE-200.

Confusion could also arise because weaknesses form hierarchies. For example, cross-site scripting (CWE-79) is a subclass of improper input validation (CWE-20). So, one tool might warn of CWE-79, whereas another designates the weakness as CWE-20. These are straightforward classification issues.

Precisely attributing the location of vulnerabilities or even counting the total number is much more difficult when vulnerabilities share statements or execution paths. A good example comes from Nagios. Two different functions (see Figures 2a and 2b) remove an event from `event_list_low`, free it, and then call `reschedule_event()` to reschedule it. Lines 1462 and 2603, where the list was touched, constitute two sources or beginnings of data flows. Eventually, `reschedule_event()` passes `event_list_low` to `add_event()` (see Figure 2c). There, the rescheduled event is added either at an appropriate place in `event_list` or as the head, if it comes first. The statements adding the event constitute two sinks or terminations of data flows. One tool reported a total of four use-after-free (CWE-416) warnings: one warning for each path between the two sources and the two sinks.

In another example, from lighttpd (an open source Web server), one function was called from dozens of places, yielding some 70 warnings. However, a fix could be made at just the function. Counting every path as a separate weakness doesn't seem fair.

Are these examples just freakish exceptions? Let's define a weakness as simple if it's

- associated with only one CWE (not a part of a chain or hierarchy),
- attributed to only one statement, and
- distinct from other weaknesses (no intermingled flows).

We estimate that only between one-eighth and one-third

of all weaknesses are simple.[4] So, the Nagios and lighttpd examples are typical.

Even location isn't always clear. Some CWEs are associated with whole regions of code. For instance, dead code (CWE-561) and leftover debug code (CWE-48910) can relate to larger pieces of code. OS command injection (CWE-78) might occur when no appropriate filtering function occurs anywhere in the path between the tainted data source and the sink. Encryption of sensitive data might be entirely missing (CWE-311). In all these examples, no group of statements is wrong, so to which location should we ascribe the bug?

Tools Find Real Problems

In SATE 2008, 2009, and 2010, we received a combined 128,043 warnings from 17 teams that analyzed test cases consisting of 12,468,329 LOC in C, C++, and Java. A simplistic examination of the results could lead to unwarranted conclusions. What have we learned about static-analysis tools?

Tools can find real problems from a wide range of weakness classes. In 2008, warnings reported by tools included at least 21 of the CWE/SANS (SysAdmin, Audit, Network Security) Top 25 Software Errors or related CWE IDs. In 2009, about half the manual findings from expert analysis were also reported by tools.

Human review and automated analysis by tools complement each other. As Okun and his colleagues said, "While human analysis is better for some types of weaknesses, such as design and authorization issues, tools find weaknesses in many important weakness categories and can quickly identify and describe in detail many weakness instances."[4]

So, Which Tool Should I Use?

Just as the choice of biking, driving, taking a bus, or walking to a location depends on the traveler's goals, resources, situation, and preferences, the choice of a static-analysis tool has many facets. Some tools aim to check general code quality; others are security oriented. Different users need different balances of the tensions between timely processing and precise analysis and between receiving a minimal number of low-priority warnings and not missing possible flaws.

The number of warnings per thousand lines of code (KLOC) varies significantly among applications and tools. Over all three SATE events, all tools, and all test programs, the minimum was 0.031 warnings per KLOC, the maximum was 83, and the median was 4.55. The two test cases that had over 1 million LOC had a tighter range: the minimum was 0.034 warnings, the maximum was 1.33, and the median was 0.33.

Tools generally find different sets of weaknesses. Figure 3 shows that for all warning classes, the reports of

```
1462    for (temp_event=event_list_low;temp_
        event;temp_event=temp_event->next){
        ...
        }
        ...
        remove_event(temp_event,&event_list_low);
        free(temp_event);
        ...
        reschedule_event(new_event,&event_list_low);
(a)
2603    for (temp_event=event_list_low;temp_
           event;temp_event=temp_event->next){
        ...
        }
        ...
        remove_event(temp_event,&event_list_low);
        free(temp_event);
        ...
        reschedule_event(new_event,&event_list_low);
(b)
        add_event(...,timed_event **event_list){
            first_event = *event_list;
            ...
808     else if(event->run_time < first_event->run_
            time){// 43523 43525
            ...
        else{
            temp_event = *event_list;
            while(temp_event!=NULL){
819             if(temp_event->next==NULL){// 43522
                   43524
(c)
```

Figure 2. Code for Nagios (an application for host, service, and network monitoring) shows intermingled data flows (a) The application finds an event (line 1462), which is the source (beginning) of one data flow. The application removes the event from the list, frees it, and reschedules it, calling `add_event()`. (b) This pattern repeats (line 2603), which is the second data-flow source. (c) The data flows end at one sink (line 808) or another (line 819). Four paths exist—one from each source to each sink.

publication 500-283, US Dept. of Commerce, Sept. 2011, pp. 4–40; http://samate.nist.gov/docs/NIST_Special_Publication_500-283.pdf.

3. E.M. Voorhees and D.K. Harman, eds., *TREC: Experiment and Evaluation in Information Retrieval*, MIT Press, 2008.
4. V. Okun, R. Gaucher, and P.E. Black, "Static Analysis Tool Exposition (SATE) 2008," NIST special publication 500-279, US Dept. of Commerce, June 2009, pp. 4–37; http://samate.nist.gov/docs/NIST_Special_Publication_500-279.pdf.

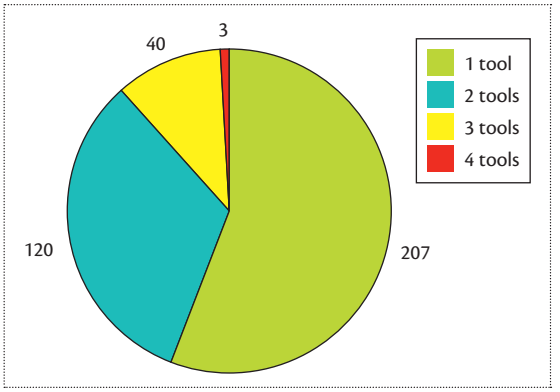

Figure 3. Of all the warnings reviewed in SATE 2009, excluding false warnings, over half were found by only one tool. Only three of 370 warnings were found by four tools.

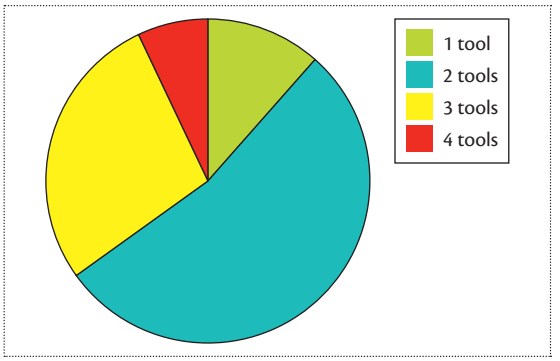

Figure 4. Overlap between sets of warnings is much higher for buffer errors. Over half of the warnings we reviewed, excluding false warnings, were reported by at least two tools.

5. "Code Verification and Run-Time Error Detection through Abstract Interpretation," white paper, MathWorks, 2007; www.mathworks.com/tagteam/42825_white_paper_abstract_interpretation.pdf.
6. V. Okun, A. Delaitre, and P.E. Black, eds., *The Second Static Analysis Tool Exposition (SATE) 2009*, NIST special publication 500-287, US Dept. of Commerce, June 2010; http://samate.nist.gov/docs/NIST_Special_Publication_500-287.pdf.

Paul E. Black is a computer scientist for the US National Institute of Standards and Technology and leads the SAMATE (Software Assurance Metrics And Tool Evaluation) project. His research interests include algorithms and data structures, formal methods, assuring software quality, and static analysis of programs. Black has a PhD from Brigham Young University's Computer Science Department. He's a member of IEEE, the IEEE Computer Society, and ACM. Contact him at paul.black@nist.gov.