STATIC ANALYZERS: SEAT BELTS FOR YOUR CODE

Paul E. Black
paul.black@nist.gov
National Institute of Standards and Technology

Abstract

Just as seat belt use is wide spread, we argue that the use of static analysis should be part of ethical software development. We explain some of the procedures of the four Static Analysis Tool Expositions (SATE) and some of the lessons we learned: sophisticated explanation aids and graphical code displays help programmers understand warnings, warnings have values more nuanced than just true or false including context-dependent or quality-related information, bugs are intermingled and diffuse rather than being distinct, and tools find real problems in code. Since security must be designed in, static analysis should be used early in software development to reduce vulnerabilities, or even better, to provide feedback to educate software developer and reinforce good practices, minimizing vulnerable constructs ever getting in the code. To understand static analyzers, we collected thousands of reference programs with documented weaknesses. These are publicly available. Even as industry migrates to languages safer than unconstrained C, which eliminates many possible weaknesses, static analysis can be even more useful to check annotations, guarantees, conditions, and specifications provided by developers.

Static Analysis Tool Exposition (SATE) Background

In his keynote address at the Static Analysis Summit II[1], November 2007, Bill Pugh called for large sets of results of different static analysis tools run on the same software. To provide such data sets, we began the Static Analysis Tool Exposition (SATE).[2] We based SATE on the NIST Text REtrieval Conference (TREC)[3], a very successful program which had been growing and adapting for some 17 years. We wanted to better understand the capabilities of static analysis tools, encourage research and improvement in them, and increase the use of such tools.

To start the process, 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. We selected those languages due to their wide use and the many static analysis tools that handle them. Participating toolmaker teams ran their tool on the test cases and returned the result to us for analysis. We planned to approximate ground truth by correlating warnings. We organized a workshop for June 2008 where everyone could share their experience and learn from each other. To allow developers of the programs that we chose as test cases time to address any urgent concerns which might arise, we waited six months to publish the reports and all the data, which we did in June 2009.[4] We caution that useful interpretation of this data is not simply a matter of, say, comparing the average number of warnings produced by each tool. Simplistic bug counting is not justified, and bug counting is far more complicated than we could have imagined, as we explain below.

Although we made many improvements in the details of the protocol, all SATEs have followed these same general steps of (i) organizers choosing test cases, (ii) teams running their tools, and (iii) organizers analyzing results, (iv) holding an experience workshop, and later (v) releasing reports and data.

Some improvements and changes to later SATEs

We were not prepared for the huge number and diversity of warnings. Checking all warnings manually was completely impractical for SATE. In subsequent SATEs we selected stratified random samples of warnings to analyze, and we commissioned expert security analysis of some of

the test cases and correlated warnings with the experts' findings. We were still not sure this constituted a good measure of program security.

Why not use sound static analyzers, for example those using abstract interpretation[5], to establish ground truth? For the classes of errors that they handle, they can reach the goal of proving complete absence of such errors. However there are many vulnerability classes they have not been programmed to handle. Even a proof that a program is a perfect implementation of a specification does not guarantee the absence of flaws in the specification. Finally the number of bugs does not necessarily correspond to the security of a program: a program with few but serious bugs is less secure than a program with many trivial bugs.

Paul Anderson proposed using publicly reported vulnerabilities to provide ground truth for analysis. In SATE 2010[2] we chose some of the test cases based on known software vulnerabilities listed in Common Vulnerability and Exposures (CVE)[6] reports. By sifting through CVE information and information for widely used programs, we identified programs that had many security-relevant vulnerabilities at some time. 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 the fix, to a related dataflow path, or to 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 old versions with fixed versions. In the SATE analysis phase, we searched for tool warnings that correlated with reported vulnerabilities.

Choosing programs with many CVEs was another reason for SATE 2010 and SATE IV to include far larger programs as test cases than in 2008 or 2009.[7]

Participating teams pointed out that it is time-consuming to set up the environment to compile large programs. This is especially burdensome as SATE asked the teams to handle several different programs. Toolmakers told us that it was easier to install their tool in an environment than to build a specific environment, since they are very familiar with the resources needed by their tool. To make it easier for teams, in SATE IV we created a virtual machine with an appropriate environment for each test case.

In SATE IV we added a track for programs in PHP. We also added about 60 000 small, engineered programs in C/C++ and Java as test cases. These are all of the so-called Juliet test suite, which is in the SAMATE Reference Dataset (SRD)[8], that compile in Linux.

Who needs graphical reporting?

In order to automatically compare warnings, we came up with a simple XML format for teams to submit their reports. The format comprised identifiers, one or more traces of locations (line number and file path), weakness type, severity and likelihood, human evaluation, and an optional container for whatever constituted the original tool output. The weakness type was a name and optionally a Common Weakness Enumeration (CWE)[9] number.

During analysis of warnings we found that using only location and weakness type was often very difficult to verify or refute the presence of a weakness and its severity. Many tools produce rich and varied sets of information, such as variable and function names, execution path as a code slice, and conditions, and display the information graphically. With the generous help of toolmakers, we were sometimes able to graphically consult this information to understand warnings.

We learned that a graphical user interface (GUI) and the rich information presented in it is often very helpful to efficiently and correctly understand warnings and what the effect might be.

What We Learned From SATE Events

We now know much more about how to understand static analysis tool capabilities, the nature of bugs, and how better to use these tools.

When we began SATE, we thought either a warning indicates an actual bug or the tool made a mistake: a simple true or false. That was far too simplistic, especially in a largely context-free situation like SATE. Some weaknesses, like Leftover Debug Code CWE-489[10], are just not discernible without application knowledge. That is, one cannot inspect

the program and know for sure in every case whether a block of code is left over from debugging work or provides needed functionality in the application. Outgoing Hard-coded Password CWE-259 might be discerned if the tool knows 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 OS Command Injection CWE-78 be reported if it is in clearly dead code (which might be activated some day by maintenance changes) or is in a function that is always called with safe parameters? Some warnings, like "strncpy() may not null-terminate" or Use of Inherently Dangerous Function CWE-242, are absolutely correct as worded, but may be utterly insignificant.

We learned that there are nuances to warnings, not just "true" or "false." In 2010[2] we used five categories: true-security, true-quality, true-insignificant, unknown, and not a weakness. "True-quality" means the warning is true, but relates to code quality, not security. In contrast to SATE's attempt to determine absolute truth, tools are used during development to decide what code, if any, should be changed, not decide absolutely if an attack is possible.

User should "tune" their tool by filtering outputs to remove low value warnings, turning off certain checks, or turning on other checks. For instance, a repeated (true!) caution that "strncpy() may not null-terminate" might be more distracting that helpful. Turning off or filtering out classes of warnings that have little value for one development style or application area allows developers to get more useful warnings in general.

Many tools allow users to write their own checks, which may help depending on the coding style and policies.

SATE results and analyses show that a program's security can be dramatically improved when developers keep it in mind. OpenNMS had a very large number of security issues in 2008. In fact, it had so much higher a density of issues than other programs in SATE that we wondered if our methods were flawed. We learned that it was not written with concern for security. (OpenNMS has since had a major security overhaul.)

Programming style may make it easier or harder for tools to find weaknesses. "For example in Nagios, 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."[11] Checking for NULL at use requires non-local analysis, which makes it harder for a tool or a human to be sure the function return is handled properly. The memory management in Dovecot produced a similar situation in 2010.

Most bugs are not distinct

We want to match the same warnings from different tools to get an idea of the overlap between them. Simplistic matching assumes each bug is distinct, which they are not. Consider the following pieces of code from CVE-2010-1773. The first is wrong; the second is fixed.

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

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

The variable numberShadow should be decremented instead of subtracting 1 from the result of the modulo (%) operation. An Off-by-one error, which is Common Weakness Enumeration (CWE)[9] 193, led to 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 while another tool reports CWE-200.

Confusion may also arise because weaknesses form hierarchies. For example, Cross Site Scripting (XSS) CWE-79 is a subclass of Improper Input Validation CWE-20. So one tool might warn of CWE-79 while another tool designates it CWE-20. But these are straightforward classification issues.

Precisely attributing the location of vulnerabilities or even counting the total number is much more

```
   1462    for (temp_event=event_list_low;temp_event;temp_event=temp_event->next){
               ...
           }
(a)        ...
           remove_event(temp_event,&event_list_low);
           free(temp_event);
           ...
           reschedule_event(new_event,&event_list_low);


   2603    for (temp_event=event_list_low;temp_event;temp_event=temp_event->next){
               ...
           }
(b)        ...
           remove_event(temp_event,&event_list_low);
           free(temp_event);
           ...
           reschedule_event(new_event,&event_list_low);


      add_event(...,timed_event **event_list){
           first_event = *event_list;
(c)        ...
    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
```

Figure 1. Nagios code shows intermingled dataflows. (a) Event is found, line 1462, which is the beginning or source of one dataflow. It is removed from the list, freed, then a new event is rescheduled, which calls add_event(). (b) Same pattern is repeated, line 2603, which is the second dataflow source. (c) Dataflows end at line 808, one sink, or line 819, a second sink. There are four paths: one from each source to each sink.

difficult when vulnerabilities share statements or execution paths. A good example comes from Nagios. Two different functions, in Figure 1a and 1b, remove an event from event_list_low, free the event, then call reschedule_event() to schedule a new event. 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(), Figure 1c. There the new event is added either at an appropriate place in the 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, which comes from lighttpd, one function was called from dozens of places, yielding some 70 warnings. However a fix could be made at just the function. It does not seem correct to count every path as a separate weakness.

Are the above examples just freakish exceptions? Let us call a weakness *simple* if it is associated with only one CWE (is not a part of a chain or hierarchy), is attributed to only one statement, and is distinct from other weaknesses (no intermingled flows). We estimate that only between 1/8 and 1/3 of all weaknesses are simple weaknesses.[12] So the Nagios and lighttpd examples are typical.

Even location is not always clear. Some CWEs are associated with whole regions of code. For instance Dead Code CWE-561 and Leftover Debug Code CWE-489 may relate to large pieces of code. OS Command Injection CWE-78 may occur when there is no appropriate filtering function anywhere in the path between the tainted data source and the sink where it leaves the program. Encryption of sensitive data may be entirely missing (CWE-311). In all these examples, no group of statements is wrong, so to which location should we ascribe it?

Tools find real problems

In SATE 2008, 2009, and 2010 combined, we received a total of 128 043 warnings from 17 different teams who analyzed test cases consisting of 12 468 329 lines of code in C, C++, and Java.

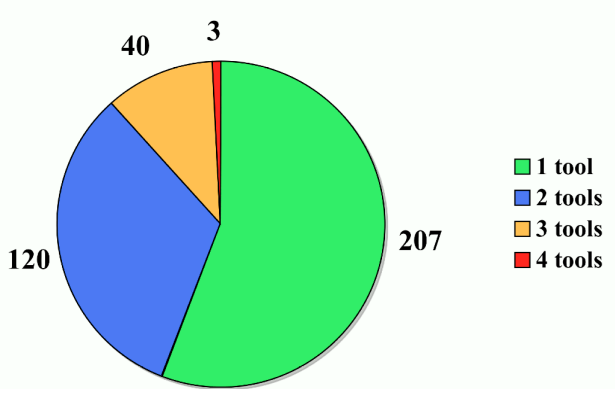

Figure 2. Of all warnings reviewed in SATE 2009, excluding false warnings, over half were only found by one tool each. Only 3 of 370 warnings were found by four tools.

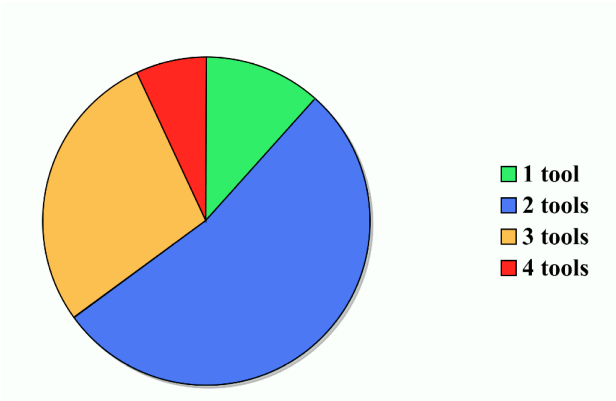

Figure 3. Overlap between sets of warnings is much higher for buffer errors. Over half of the not-false warnings we reviewed were reported by at least two tools.

Simplistic examination of the results may lead one 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 SANS/CWE Top 25 Weaknesses 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. "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."[13]

So, which tool should I use?

Just as the choice of biking, driving, taking a bus or walking depends on one's goals, resources, situation, and preferences, the choice of a static analysis tool has many facets. Some tools are designed to check general code quality, whereas others are security-oriented. Different users need different balances of the tensions between timely processing and precise analysis and between few low-priority warnings and not missing possible flaws. The number of warnings per thousand lines of code (k LoC) varies significantly. Over all three SATE events, tools, and test programs, the minimum is 0.031 warnings/k LoC, the maximum is 83, and the median is 4.55. The two test cases that had over 1 million LoC had a tighter range: minimum is 0.034, maximum is 1.33, and median is 0.33 warnings/k LoC.

Tools generally find different sets of weaknesses. Figure 2 shows that for all warning classes, the reports of different tools have little overlap. For example, over half of the warnings that we reviewed in SATE2009 were only found by one tool, excluding warnings determined to be false. Less than 1% of all not-false warnings were found by four tools. For common weaknesses, such as buffer errors, tools report the same error more often, as shown in Figure 3. As with Figure 2, Figure 3 excludes warnings that we found to be incorrect. Including or excluding true-security, true-quality, or true-insignificant warnings do not materially change these observations.

The SATE data can help users select tools that most closely match their needs. A user can download the SATE data, select warnings from the highest priority weakness classes, and determine which tool or tools are more effective at findings those weaknesses. This information, combined with languages and frameworks supported, resources required, presentation style, and customization options, may help users understand tool options.

FUTURE

We plan to use SATE results to develop reference sets of programs to serve as benchmarks for static analyzers. As one step, we collected over 61 000 programs with known weaknesses. They constitute

the SAMATE Reference Dataset (SRD)[8], which is publicly accessible and contains programs mostly in C, C++, and Java. Most of the programs are small, engineered programs, although a few are extractions from open source applications. Each program in the SRD has some amount of information, for example location and type of weakness. We will draw some of the benchmark programs from the SRD.

In future SATE events, we would like to add tracks for binary analysis, hybrid (combined static and dynamic) tools, and more languages.

With such data we can estimate the total number of weaknesses or vulnerabilities in a program by capture/recapture statistical analysis, investigate whether coincidence of warnings implies correctness, and determine how much assurance comes from using static analysis tools.

CONCLUSIONS

We learned that weaknesses are usually *not* localized and distinct, but overlap and are interrelated. We learned that there is no one right, simple answer to "what weaknesses does this program have?" We also learned that without additional machine-readable application specifications, tools cannot feasibly check for many important classes of weaknesses or higher level properties. We also learned that a graphical user interface (GUI) and the rich information presented in it are often helpful to efficiently and correctly understand warnings and what the effect might be.

Nobody plans to crash while driving, but everyone wears seat belts. Similarly nobody wants to put bugs in code and tools cannot catch all errors, but static analyzers can save code from some catastrophes. Since tools can check millions of lines of code quickly and repeatably, we see no reason an ethical software development process should not include automated analysis tools.

References

[1] Black, P. E., & Fong, E., "Proceedings of Static Analysis Summit II," *ADA Letters*, 28(1), April 2008, pp. 71-102. Available at http://hissa.nist.gov/~black/Papers/procSASII.html

[2] Okun, V., Delaitre, A., & Black, P. E., "The Third Static Analysis Tool Exposition (SATE 2010)," *NIST Special Publication 500-283*, September 2011, pp. 4-40. Available at http://samate.nist.gov/docs/NIST_Special_Publication_500-283.pdf

[3] *TREC: Experiment and Evaluation in Information Retrieval*, Voorhees, E. M., & Harman, D. K. (eds.), 2008, MIT Press. http://trec.nist.gov/

[4] Okun, V., Gaucher, R., & Black, P. E., "Static Analysis Tool Exposition (SATE) 2008," *NIST Special Publication 500-279*, June 2009, pp. 4-37. Available at http://samate.nist.gov/docs/NIST_Special_Publication_500-279.pdf

[5] "Code Verification and Run-Time Error Detection Through Abstract Interpretation," http://www.mathworks.com/tagteam/42825_white_paper_abstract_interpretation.pdf

[6] "Common Vulnerabilities and Exposures," http://cve.mitre.org/

[7] Okun, V., Delaitre, A., & Black, P. E., "The Second Static Analysis Tool Exposition (SATE) 2009," *NIST Special Publication 500-287*, June 2010. Available at http://samate.nist.gov/docs/NIST_Special_Publication_500-287.pdf

[8] "SAMATE Reference Dataset," http://samate.nist.gov/SRD

[9] "Common Weakness Enumeration," http://cwe.mitre.org/

[10] "CWE-489: Leftover Debug Code," http://cwe.mitre.org/data/definitions/489.html

[11] SATE 2008, *op. cit.*, p. 22.

[12] *ibid.*, p. 28.

[13] *ibid.*, p. 22.

Keywords: static analysis; software quality; software assurance; software engineering; SAMATE; SATE

Author Bio:

Dr. Black has been a Computer Scientist for the U.S. National Institute of Standards and Technology (NIST) for 14 years and leads the SAMATE project. He has nearly 20 years of industrial experience in areas such as developing software for integrated circuit design and verification, assuring software quality, and managing business data processing. Dr. Black has published in the areas of static analysis, software testing, software configuration control, networks and queuing analysis, formal methods, software verification, quantum computing, and computer forensics. He earned a Ph.D. at Brigham Young University in 1998 and is a member of ACM, IEEE, and IEEE Computer Society.

Mailing address:

Dr. Paul E. Black

100 Bureau Drive, Stop 8970

Gaithersburg, MD 20899-8970

USA

Phone: +1 301-975-4794

Fax: +1 301-975-6097

Email: paul.black@nist.gov