Counting Bugs is Harder Than You Think

Paul E. Black
Software and Systems Division
U.S. National Institute of Standards and Technology
Gaithersburg, Maryland, USA
paul.black@nist.gov

***Abstract*—Software Assurance Metrics And Tool Evaluation (SAMATE) is a broad, inclusive project at the U.S. National Institute of Standards and Technology (NIST) with the goal of improving software assurance by developing materials, specifications, and methods to test tools and techniques and measure their effectiveness. We review some SAMATE sub-projects: web application security scanners, malware research protocol, electronic voting systems, the SAMATE Reference Dataset, a public repository of thousands of example programs with known weaknesses, and the Static Analysis Tool Exposition (SATE). Along the way we list over two dozen possible research questions, which are also collaboration opportunities.**

Software metrics are incomplete without metrics of what is variously called bugs, flaws, or faults. We detail numerous critical research problems related to such metrics. For instance, is a warning from a source code scanner a real bug, a false positive, or something else? If a numeric overflow leads to buffer overflow, which leads to command injection, what is the error? How many bugs are there if two sources call two sinks: 1, 2, or 4? Where is a missing feature? We conclude with a list of concepts which may be a useful basis of bug metrics.

***Keywords*-software engineering; software tools; software metrics; software debugging**

I. Introduction

Software Assurance Metrics And Tool Evaluation (SAMATE) is a project at the National Institute of Standards and Technology (NIST), an agency of the United States Federal Government. NIST "employs about 2 900 scientists, engineers, technicians, and support and administrative personnel." Also, NIST hosts about 2 600 guest researchers, summer students, sabbatical researchers, postdoctoral research associates and facility users from academia, industry, and other government agencies [24]. Most work in Gaithersburg, Maryland, with a smaller group in Boulder, Colorado [25]. NIST researches measurement science, standards, and technology in areas from atomic clocks to quantum computers to bullet-proof vests, from microspheres to dental ceramics, from computer forensics to robotics.

SAMATE began in the fall of 2004, originally proposed to the U.S. Department of Homeland Security (DHS) as analogous to NIST's Computer Forensics Tool Testing project [14]: pick a class of tools, investigate the behavior of tools in the class, define the class' characteristics, and develop an acceptance test. The vision was that SAMATE would be a pipeline of sub-projects working on classes of software assurance tools. A tool specification and acceptance test would help enhance and develop such tools by enabling testing of tools and measuring their effectiveness.

The International Software Testing Qualification Board (ISTQB) gives terms about causes of software defects [19].

> A human being can make an error (mistake), which produces a defect (fault, bug) in the program code ... If a defect in code is executed, the system may fail to do what it should do (or do something it shouldn't), causing a failure. Defects in software, systems or documents may result in failures, but not all defects do so.
>
> ...
>
> Failures can be caused by environmental conditions as well.

In this paper, we use the following terminology [26]:

> A *vulnerability* is a property of a system['s] security requirements, design, implementation, or operation that could be accidentally triggered or intentionally exploited and [would] result in a security failure. A vulnerability is the result of one or more *weaknesses* in requirements, design, implementation, or operation. A *warning* is an issue (usually, a weakness) identified by a tool. A (tool) *report* is the output from a single run of a tool on a test case. A tool report consists of warnings.

Whether a weakness is a vulnerability may depend on how the software is used. To simplify discussion, we generally mean weakness (what ISTQB calls defect) when we say "bug."

Section II explains some sub-projects in SAMATE, with those of particular interest to those involved with Source Code Analysis and Manipulation (SCAM) toward the end. We wish especially to draw attention to Sect. II-H on the SAMATE Reference Dataset (SRD), a public repository of thousands of example programs with known weaknesses, and Sect. II-I on the Static Analysis Tool Exposition (SATE), an opportunity for tool developers to run tools on real-world open source programs and share experiences. Where relevant we note critical research problems. In Section III we move toward a science of software bugs, explaining nuances, open questions, and thoughts about what exactly constitutes a weakness, how they might be measured, and some concepts that may be useful as a basis of weakness metrics.

II. Some SAMATE Sub-Projects

We briefly explain sub-projects within SAMATE, pointing out research and collaboration opportunities. This section puts sub-projects that may not be as interesting to the Source Code Analysis and Manipulation (SCAM) community earlier and those likely to be more interesting later.

A. Web Application Security Scanners

The first SAMATE sub-project we touch on is understanding web application security scanners. A web application scanner is an automated (non-interactive) program that examines web applications for security vulnerabilities by sending requests and examining responses (after [17]). Essentially a web app scanner is a form of automated hacker: it tries various attacks on all the pages of a web site it can find. Scanners check what is returned while watching for anything indicating a possible way to attack. Since the number of attacks is unbounded, scanners typically do relatively shallow analysis across a large set of pages. Even this limited analysis can take hours and produce reams of false positives or the same code weakness manifest in thousands of warnings. We developed a specification, which the Web Application Security Consortium (WASC) used as a basis for their Web Application Security Scanner Evaluation Criteria [34].

How should web app scanners be measured for effectiveness? There is no one right way to scan a web site, no expected set of requests to watch for. Recognizing the requests that the scanner sent, classifying them into attack groups, then judging how effective a scanner would be is highly complex, especially considering the scanner must adapt its requests according to the responses it receives. As explained in [16], the simplest method to measure the effectiveness of web app scanners is to build web applications with known vulnerabilities and selectable levels of defense against exploits. For instance consider SQL injection, CWE-89 [13]. With defense turned off, user input goes directly to SQL queries without any filtering. A moderate level of defense removes some special characters. The highest defense level rejects all special characters, restricts the SQL user rights, and uses SQL prepared statements.

We developed one test application, a banking application, with selectable levels of defense. To use different sets of underlying technologies, we sketched two additional applications.

There are research opportunities to further classify levels of defense against exploits, recognize such defense mechanisms in source code, and design programming languages with these mechanisms built-in. There is also the opportunity to build more thorough test applications.

B. Malware Research Protocol

When we wanted to share the banking application, which we designed to have many vulnerabilities, there was a concern that someone might base a production web site on the NIST code and be exploited. We thought of several techniques to minimize the chance of the code being used accidentally [4]. For instance, the code could refuse to run if the system date is not 1976 or there is no file named `enableVulnerableWebApp`, analogous to the use of auxotrophs in biological research. Other possible safety mechanisms are to display a red banner that the web site is vulnerable, use outgoing firewalls, or run the machine without any on-going connection to other machines, a so-called "air gap."

To get community agreement and guidance for malware researchers, we suggest there be a malware research protocol. Biology laboratories working with pathogens take well-defined precautions: more serious pathogens require more stringent precautions [9]. Similarly the malware research community should have a broad consensus that, say, if you work with spam worms, use outgoing firewalls.

There are research opportunities to define a taxonomy of malware (e.g., virus, trojan horse, root kit, replicating worm, polymorphic virus, and phishing attack), a catalog of precautions and defenses, suggestions as to what defenses are appropriate for what malware, and a cool symbol analogous to those for biohazard [1], [2] or radiation.

C. Software Labels

Another effort is software labels. Just as a nutrition facts label on a candy bar or material safety data sheets give useful information, a software label would help the consumer buy more secure software in two ways. First, the consumer could select software that is rated more secure. Second, and more important in the long term, a label allows software developers to differentiate their products and receive a market incentive by developing more secure software.

Different audiences and products need vastly different information, and hence labels. Some possible audiences are casual home users, small businesses, and software integrators. For other cautions, similar programs and related efforts, more on scope, and possible content, see [3].

Why is this immediately relevant to SAMATE and the Source Code Analysis and Manipulation (SCAM) community? Because results from assurance tools can have additional benefit if they are used, and a visible (and reliable) label would increase the incentive to use better software development techniques.

There are research opportunities to catalog possible content of a label (lines of code, cyclomatic complexity [23], libraries used, CWEs mitigated), define criteria for including or excluding a type of information, and determine which criteria correlate with security or other attributes.

D. Software Assurance Studies

Long term, significant progress is often based on proven results, not just customary practices or expert opinion. To help build a scientific base, we are working on a number of studies.

1) Capture/Recapture: How many weaknesses are there in a program? Biologists estimate the total size of a population by capturing a sample of creatures, marking and releasing them, then capturing another sample. The number of creatures recaptured compared to the sample size lets them estimate the total population size [8]. Perhaps we can apply similar models to estimate the total number of weaknesses [7], [29]. Data from the Static Analysis Tool Exposition, or SATE, (see Sect. II-I) shows that static source code analysis tools find different instances of weaknesses in the same program, and the overlap is typically limited. We are working on estimating the total number of weaknesses in a program from the overlap using biologists' statistical methods.

Open questions include

- Are the weaknesses found a random sample of all weaknesses?
- Are the weaknesses found by one tool statistically independent of the weaknesses found by other tools?
- Can we estimate the number of weaknesses in one class from the number of weaknesses in other classes?
- Can we estimate the number of vulnerabilities from the number of weaknesses?

2) Does Coincidence Correlate With Correctness?: The National Security Agency's Center for Assured Software (CAS) found [35] that static analysis tools differed significantly in precision and recall, and their precision and recall ordering varied for different weaknesses. They conclude that the sophisticated use of multiple tools increases the rate of finding weaknesses and decreases the false positive rate.

There is an opportunity to explore the three years of SATE data to establish that warning coincidence does, indeed, imply warning correctness.

3) Tool Effectiveness: Using a static analyzer and fixing weaknesses reported should improve the quality of the code, right? Dawson Engler points out [15] that code changes may add weaknesses, that static analyzers only find certain kind of weaknesses, and that time spent on sifting through reports and fixing trivial weaknesses might be better spent on, say, code reviews. We studied the reported error rates in open source code before and after they started using static analyzers [28]. We did not find conclusive evidence, but did uncover many confounding factors.

There are many research opportunities. Even in operation, that is after deployment, how can we assign a measure to "the security of a program"? Number of reported vulnerabilities? Number of breaches? Breaches per hour of operation? Number of subsequent code changes that can be traced to vulnerability fixes? Expert assessment? Even given that the community decides on some surrogate measure of security like the preceding, what correlates with security beforehand? That is, even if we designate some context or environment, how can we measure the security of software?

E. Software Laboratory

Many times we have come up with hypotheses or experiments about weaknesses in source code, but then dismissed them because it would be far too much work to investigate. For instance, static analyzers warn about buffer overflows, but is that all the buffer overflows? If we could count all sites of buffer accesses, we might be able to make some predictions. In another case, we posit that some metrics apply to implementations of an algorithm and some metrics apply to the algorithm itself, or to every (correct) implementation of the algorithm. For instance, "lines of code" clearly measures a particular implementation, whereas computational complexity (big "O" notation) measures the algorithm. We could experimentally determine the object of a metric by applying many behavior- and time-preserving transforms to a program. If we get the same measure for all the transformed programs, we have confidence that the metric applies to the algorithm. Otherwise it applies to the implementation.

A software laboratory with tools and in-house expertise to transform, slice, count, and generally investigate programs in many languages may help with these.

F. Source Code Security Scanners

Our core effort has come to be source code security scanners and work related to those. Why look at the end product of software development when we know the most important aspect is designing the system and writing code correctly in the beginning? We have two reasons. First, source code is "the only precise description of the behavior of the system" to quote SCAM's call for papers. A more subtle point is that the more precisely and accurately software properties can be measured, the better and more objectively we can judge the utility of new processes or proposed methodologies. The problem is undecidable in theory [22] and very hard in practice, but there has been great progress in the last five years.

We developed a specification for source code security scanners [5] and are finishing conformance tests. In developing the specification, many users wanted us to include metrics for false positive rates and similar operational components. While delving into an objective foundation for such metrics, we found many problems with the fundamental definitions upon which such metrics might be based. Do we count proven vulnerabilities or mere weaknesses? What is a single bug or weakness? Do two identical messages constitute two warnings or one? Do two messages about the same weakness instance constitute two warnings or one? Is the message `strncpy() does not null terminate` a good warning or a false positive? If an uncaught exception (CWE-248 [13]) or an SQL injection (CWE-89) is in dead code, is it a valid warning or a false positive? We elaborate on these and other open questions in Sect. III.

G. Tools to Analyze Voting Systems

Electronic voting systems are a big concern recently. The U.S. Congress passed the Help America Vote Act of 2002 (HAVA) leading to Voluntary Voting System Guidelines (VVSG), two volumes for electronic voting systems dealing with everything from accessibility to physical tamper resistance to auditing. Volume 1 [33] has "workmanship" requirements for source code. Currently laboratories mostly do manual review of voting system source code to assess compliance with requirements. For each of the 49 requirements in the current version, VVSG 1.0 (2005), SAMATE determined how much checking could be automated and wrote rules where feasible. We divided requirements into six classes based on how much checking can be automated:

1) Expressible as a general automated check. Example: case statements have a default explicitly defined.
2) May be implemented many ways, so check is style-dependent. Examples: Modules shall contain headers. In modules of 10 lines or less, the header may be limited to the identification of unit and revision number.
3) The tool used could not check, but another tool may.
4) Semantic content that a program can point to, but a human must judge conformance. Example: operator intervention shall not re-direct program control.
5) Human understanding required. Example: names shall enhance program readability and intelligibility.
6) Not relevant to the programming language. Example: self-modifying code is prohibited.

The study found "that automation could partially or fully verify 60 % of the requirements for the Java programming language and that automation could partially or fully verify 70 % of the requirements for the C language." [20] The research challenge is doing similar work for VVSG 1.1 and VVSG 2.0, which replace most of the requirements related to coding style with requirements more directly indicating code quality.

H. SAMATE Reference Dataset (SRD)

When we began to consider source code scanners, we realized we needed example code, that is, code with known weaknesses. We could not find any comprehensive collections. We therefore created the SAMATE Reference Dataset (SRD) [30] to be a web-accessible repository of code with known weaknesses. It is intended to be an open (community) repository. That is, anyone may add to it, even if not directly applicable to SAMATE. We plan for the SRD to also serve as a central library from which researchers can draw a spectrum of material and which preserves test cases developed by students who end their work and move on.

In order to be an archive, source code will never be changed once it is added to the SRD. A test case might be marked deprecated to warn users that a significant flaw was found and that the test case should not be in new work. For instance test cases 1322 and 1323 concatenated material to a source string, not the work buffer. This was a simple mistake by the programmer, not an intentional weakness. Both cases were deprecated and replaced by 2083. However the source code for 1322 and 1323 will be accessible in the future. Such preservation allows later researchers to more closely repeat studies.

The SRD can handle any language or graphical notation with a digital representation. It is also intended to archive binaries and higher-level designs, in addition to source code in typical languages such as C, Java, or Erlang. Test cases can be small or large, synthetic (purpose written) or live (taken from production applications), pristine (absolutely no other errors) or typical (other weaknesses, stylistic problems, and problematic constructs.)

To understand the strengths of static analysis, weaknesses must be embedded in complicated code constructs, in addition to the most basic form. To illustrate, consider the following example of Resource Locking Problems CWE-411, from test case 2109.

```
#define logStr(s) if(writeLog){fputs(s,...

int main(int argc, char *argv[])
{
    bool writeLog = true;

    logFile = fopen(logFileName, "a");
    if (!logFile)
        writeLog = false;

    logStr("Action 1");
    logStr("Action 2");

    if (writeLog)fclose(logFile);
    return 0;
}
```

The weakness is that program should use some kind of interlock for the log file. In this basic case the open, write, and close are all in the same function. To form a more precise model of the analytical capabilities of a static analysis tool, the open could be in a called function and the corresponding close in another, peer, called function. In that case, analysis must be across procedure calls. Other code complexities are that the open and close could be inside loops; the file pointer could be saved in a data structure and accessed through various amounts of indirection, and so forth.

For ease of reference, the SRD allows test suites to be defined which include any number of test cases. A user can view or download all the test cases in a test suite. This allows a researcher to add a set of test cases to the SRD, then define a single test suite allowing users to examine or retrieve that exact set of test cases.

As of 1 June 2011 the SRD contains over 1 800 example programs or test cases in C, Java, C++, and PHP. About 1 100 are synthetic test cases in C used for Kratkiewicz's thesis to test static and dynamic buffer overflow tools [21]. The SRD includes 28 test cases from MIT Lincoln Labs, which were extracted from popular applications with known buffer overflows [37]. The SRD also has tests from the Comprehensive, Lightweight Application Security Process (CLASP) [32], Fortify Software Inc., Defence Research and Development Canada (DRDC), and Klocwork Inc. SAMATE wrote several hundred test cases for the Source Code Security Scanner specification [5]. We are adding an extensive test set, designated Juliet, which comprises almost 60 000 small test cases in C and Java covering over 100 CWEs.

There are research opportunities in determining what set of code complexities are likely to give an adequate model of the analytical capabilities of tools, determining what set of test cases can be used to predict a tool's behavior on production code, and collecting or contributing test cases in other languages and test cases consisting of binaries or high-level designs.

I. Static Analysis Tool Exposition (SATE)

Catalyzed by a suggestion from Bill Pugh, in 2008 we organized the Static Analysis Tool Exposition, or SATE, whose goal is to advance research in, and improvement of, static analysis tools that find security-relevant defects in source code. The three SATEs have had the following general steps. We choose a set of open source programs in C and Java as test cases. Tool makers, both academic and commercial, run their tools on the test cases and send back the results. Researchers led by SAMATE team members analyze the tool reports. We share experiences in a workshop, and reports and all tool results are released for public use.

The first SATE was in 2008, and the second in 2009. The final reports, our analysis, and all results are available for those. SATE 2008 found that tools can help find weaknesses in most of the SANS/CWE Top 25 [31] weakness categories [27]. SATE 2009 showed that "tools find weaknesses in many important weakness categories and can quickly identify and describe in detail many weakness instances." [26] Like spell checkers, these tools cannot catch everything. A spell checker will not turn your impromptu tweet into Shakespeare, but spell checkers are helpful enough that they are ubiquitous. We are finishing the final report for the third SATE, held in 2010. An organizing meeting for SATE IV was held in March 2011 in McLean, VA.

For SATE 2009 we changed the procedure to randomly select subsets of tool warnings for analysis and to select tool warnings based on human analysis of the test cases. The SATE 2009 data suggested that while tools often look for different types of weaknesses and the number of warnings varies widely by tool, there is significant agreement among tools for well-known weakness categories, such as buffer errors. The data also provided evidence that, while human analysis is best suited for identifying some types of weaknesses, for instance, improper access control, tools find a significant portion of weaknesses considered important by experts [26].

In SATE 2010, based on Paul Anderson's proposal to consider real-life exploitable vulnerabilities, we selected test cases based on the Common Vulnerabilities and Exposures (CVE) database [12]. CVE-selected test cases are pairs of programs: an earlier version with vulnerabilities subsequently recorded in the CVE and a later version with those vulnerabilities fixed. For these test cases, our analysis targeted warnings that identify the CVEs. We selected test cases and prepared for analysis by finding where the source code was changed to fixed a CVE, which proved generally difficult. Identifying weakness paths and sinks, where a tool warning related to the CVE can appear, is even harder.

During the course of the three SATEs, 17 different teams ran their tools on a total of 15 C/C++ and Java test cases—open source programs from 30k (non-blank, non-comment) LoC to 3.9 M LoC. The test cases included web server, network management system, mail server, weblog server, parallel virtual machine, instant messenger, and others. In each SATE, there were tens of thousands of tool warnings, which are available for analysis. Because of resource limitations, we analyzed only a small portion, for example 4 % in SATE 2009, of the tool warnings.

There are research opportunities to analyze past SATE data, pose questions to be answered by SATE, recommend procedural changes to answer such questions, propose better criteria to select test cases, suggest more powerful statistical methods to select warnings to analyze, develop more useful methods and criteria to analyze warnings, and point out ways that SATE can lead to better software.

III. Toward a Science of Software Bugs

While trying to rigorously measure behavior during SATE analysis, we encountered many questions and complications. What exactly is a weakness? Do two warnings refer to the same weakness or two different weaknesses? How many weaknesses are there if two sources call two sinks: 1, 2, or 4? Where in the code should a tool report a missing feature? These complications are not rare. The SATE 2008 data suggests that only $1/8$ to $1/3$ of all weaknesses are simple weaknesses [27]: most weaknesses are complicated.

This section explains and discusses questions and complications that arose. At the end of this section we propose some concepts upon which useful, objective metrics might be based. Many parts of this section are taken from [27] Sect. 3.4, "On Differentiating Weakness Instances."

A. On Designating a Weakness

The first notion the reader must discard is that either something is a bug or it is not; that it exists or does not.

One example of a questionable weakness is theoretical integer overflow, after SRD [30] case 2083.

```
int main(int argc, char **argv){
    char buf[MAX];

    . . . copy a string into buf

    // theoretical integer overflow
    if (strlen(buf) + strlen(argv[2]) < MAX)
        strcat(buf, argv[2]);

    . . . print buf
}
```

If the command line argument is extremely long, the sum of the string lengths overflows and becomes negative, which is less than MAX. This allows the argument to be concatenated to the string in `buf` writing outside the memory allocated for `buf`. One could argue it is not a weakness because the environment would never pass a string 2^{31} characters (2 gigabytes) long. On the other hand, might this be a problem if it runs on a machine with small integers?

This kind of situation has led to failures in production binary search [6]. But is it worth coding around C's modulo arithmetic integers to be correct in all cases? Bugs from maintaining the awkward code may cause failures in more situations than the overflow.

Another example is language standard vs. implementation convention, after SRD case 201.

```
typedef struct
{
  int int_field;
  char buf[10];
} my_struct;

int main(int argc, char *argv[])
{
  my_struct s;

  /*  BAD  */
  s.buf[10] = 'A';

  return 0;
}
```

Most compilers allocate structures in multiples of one word, which is four characters. So `buf` usually has room for 12 characters. However, the C language standard does not require structures to be even one single byte longer than is specified. People differ on whether this is a weakness.

To try to capture the nuances of bughood, we expanded the number of correctness categories from two (true or false) in SATE 2008 to four in SATE 2009 and five in SATE 2010 [26, page 13]:

- True security weakness - a weakness relevant to security.
- True quality weakness - poor code quality, but may not be relevant to security. Examples: possible buffer overflow by local user and the program is not run with superuser permissions (SUID). Function has a weakness, but is always called with safe parameters. Weakness is in dead code.
- True but insignificant weakness. Examples: database tainted during configuration. Warning about properties of a standard library function without regard to its use.
- Status unknown - we did not determine correctness.
- Not a weakness - an invalid conclusion about the code.

The categories are ordered in the sense that a true security weakness is more important to security than a true quality weakness, which in turn is more important than a true, but insignificant weakness. We are not satisfied with these categories: the guidance for marking correctness and significance takes an entire page.

B. On Distinguishing Weaknesses

In addition to determining the correctness and significance of tool warnings, in SATE we need to associate or match up warnings that refer to the same weakness instance, regardless of whether the warnings come from one tool or from different tools. Originally, we thought that each weakness had a unique manifestation, that it could be cleanly distinguished from other weaknesses. However, we found that the notion of distinct weakness instances breaks down in many cases:

- The same mistake is repeated many times.
- Data or control flows are intermingled.
- Weaknesses are related as chains, composites, or hierarchies.

We discuss intermingled flows in the next section. Here let us define and examine chains, composites, and hierarchies.

Recall from the introduction that a vulnerability actually leads to a failure for certain inputs. A weakness may be a vulnerability (lead to a failure) in some systems. However in another system, a preceding component may block the inputs that cause failure, and thus the same weakness never leads to a failure and is not a vulnerability in that system.

A vulnerability may be the result of a chain of weaknesses or the composite effect of several weaknesses. "A 'Chain' is a sequence of two or more separate weaknesses that can be closely linked together within software." [10], [11] For instance, an Integer Overflow CWE-190 in calculating size may lead to allocating a buffer that is smaller than needed, which can allow a Buffer Overflow CWE-120. Thus two warnings, one labeled as CWE-190 and one as CWE-120, might refer to the same vulnerability.

```
    // integer overflow in size calculation
    size = k * n;
    // too little memory allocated
    dst = malloc(size);
    strcpy(dst, src);
```

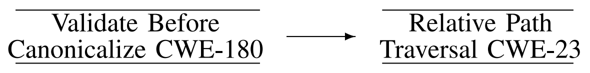
Validate Before Canonicalize CWE-180
Relative Path Traversal CWE-23

Figure 1. Chain: CWE-180 leads to CWE-23

Figure 4 in [36] is a semantic template with more details of the "patterns of relationship between software elements and faults", particularly CWE-190 and CWE-120.

Another example of a chain is the following Perl code from Chris Wysopal. Validate Before Canonicalize CWE-180 leads to Relative Path Traversal CWE-23. This flow is illustrated in Figure 1.

```
$home = "/home/www/l18n/";
$input = WebParam('lang');
if ($input =~ /\.\./) { die("Bad!"); }
$input = URLdecode($input);
$fname = $home + "$input" + ".txt";
ReadFileAndDumpToUser($fname);
```

If `lang` is `%2e./%2e./%2e./etc/password%00`, the string passes the check. The read converts `%2e` substrings into periods (.) and ascends the directory structure, possibly dumping the password file to the user.

"A 'Composite' is a combination of two or more weaknesses that can create a vulnerability, but only if they all occur at the same time." [10] For instance, Symlink Following CWE-61 requires Predictability CWE-340, Container Errors CWE-216, Race Conditions CWE 362, and Incorrect Permissions CWE-275.

Another problem in matching warnings is that some weakness classes are refinements of other classes. In other words, some classes form hierarchies of superclasses and subclasses. For instance, SATE 2008 warning 3388 reports an Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting') CWE-79 in OpenNMS. But CWE-79 is a child of the more general Improper Input Validation CWE-20. Two warnings labeled CWE-79 and CWE-20 may be the same vulnerability [27, Sect. 3.4.1].

C. *On Counting Weaknesses*

Many vulnerabilities involve multiple statements, such as a flow of tainted data or one statement that frees memory and a later statement that uses that memory. Because of shared routines, it is common for flows to be intermingled. An extreme case is SATE 2008 warning 17182 which associates some 70 different warnings of Buffer Overflow CWE-120 in lighttpd with string copy at line 638.

```
638   strcpy(path, dir->ptr);
```

The suspect statement is invoked indirectly from dozens of places. Careful analysis shows that buffer overflow is not possible from any of them. However if the code needed to be fixed, it might be done at that one statement, in a few intermediate functions, or change might be needed at many of the locations given in the warnings.

The following code from Nagios is a more subtle example. Two different functions find an event, remove the event's object from event list, free the object (line 1503 in one function and line 2644 in the other function), then reschedule a new event. Rescheduling involves adding a new object to the event list at the head of the list (line 808) or in its proper place (line 819).

```
      remove_event(temp_event,...);
1503  free(temp_event);
      ...
      reschedule_event(new_event,...);


      remove_event(temp_event,...);
2644  free(temp_event);
      ...
      reschedule_event(new_event,...);


 808  if(event->time < first_event->time){
          ...
      else{
          temp_event=*event_list;
          while (temp_event) {
 819          if(temp_event->next==NULL){
```

One tool reports four warnings, one for each of the paths. How many potential weakness instances should be counted? There are two initial statements, two uses, and four paths all together. It does not seem correct to count every path as a separate weakness in this case nor to consider it as just a single freakish two-headed weakness. On the other hand, there is no symmetrical way to tease the flows apart into just two distinct weaknesses.

Sometimes the same simple mistake is made repeatedly, analogous to a word misspelled the same way throughout a document. Although there is a weakness at every location where the mistake is made, there is in some sense a single syndrome, especially when all the errors can be corrected with a single syntactic replacement or addition.

D. *On Locating a Weakness*

Even when weaknesses can be clearly distinguished, many types of weaknesses are either the result of several statements interacting (as above), or may involve a region of code, or may not refer to any particular piece of code at all. For these reasons it is not misleading to insist that every weakness is at a single statement.

Code omission is an example of a weakness that is difficult to attribute to particular statements. If a web service program creates credentials for a remote user's session and does not invalidate the credentials after some time of inactivity, an attacker might use those credentials. This is a Session Does Not Expire CWE-613 weakness, but no statement or even group of statements is wrong. In another example, a static analyzer with access to a specification may warn of Unimplemented or Unsupported Feature in UI CWE-447 and report some location in the user interface processing where the flow might be diverted, but no existing statement is wrong.

Even when particular statements are involved, it is not obvious what statement should be reported by the tool. Certainly a path may be helpful, but typically a particular statement is indicated for summary or initial report purposes. Should it be the source statement, where the data or state first appears? The statement where a fault can occur? The statement where it is best mitigated? Likely the answer is different for different weaknesses and possibly depends on many details of the code and context.

E. On Possibly Useful Concepts

It is possible that software and the notion of weaknesses are more similar to defining "species" [18], fractals, or philosophical discussions. Perhaps our attempts to come up with objective, consistent descriptions and clear categories are misguided. Whether that is the case or not, we conclude with concepts that may be useful in order to measure, understand, analyze, and ultimately eliminate bugs.

As we saw above, "number of weaknesses" is a subtle concept. Perhaps a better base concept is the number of code changes needed so the weakness is not present. If so, would that be the minimum number of changes or the number of changes which preserves a good code structure? For example, good maintenance may dictate correcting mistakes in many calls even though one kludge might be added to the called code to make the fix. It seems that the number of weaknesses is less than or equal to the minimum of the number of sources and the number of sinks, because the number of code fixes is (usually) less than the number of weaknesses.

Here are additional concepts which may be useful, particularly when precisely defined.

Weakness class - CWE is one enumeration of classes. Much work is still needed to precisely and accurately define classes, and concepts such as chains and composites.

Vulnerability - A property of a system's security requirements, design, implementation, or operation that could be accidentally triggered or intentionally exploited and would result in a security failure.

Attack - Deliberate events attempting to cause a system failure. "Exploit" is a similar concept.

Source, sink - A location where data enters (leaves) the system. Perhaps they form a bipartite graph.

Fault - The earliest time that the state of a program diverges from what would be intended (after IEEE).

Path, data or control - A set of statements in a sequential execution.

Error - A mistake by a person, for instance, forgetting to add code to close a session.

IV. Conclusion

We sketched many of the subprojects in the Software Metrics And Tool Evaluation (SAMATE) project to point out over two dozen open research questions, which are also possible collaborations. In particular we described the SAMATE Reference Dataset (SRD) so researchers can use the thousands of example programs, which have well-characterized weaknesses, to speed their work and contribute their examples to the SRD in order to increase public resources. We also described the Static Analysis Tool Exposition (SATE). The analysis done for SATE led to many of the notions discussed in the final section. The final section points out problems with simplistic notions such as bug, false positive, false or true positive rates based on bug counts, and location of a bug. We suggest some concepts which contribute to a more rigorous foundation for defining and enumerating weaknesses, bugs, or flaws, and a better understanding of software in general.

We are interested in collaborating on questions like those posed here to help improve the assurance of software.

Acknowledgments

We thank Vadim Okun for his voluminous suggestions and Dawn J. Lawrie for her guidance and comments. Thanks also to Liz Fong, Arpad Beszedes, and Kevin Mills.

References

[1] C. L. Baldwin and R. S. Runkle. Biohazards symbol: Development of a biological hazards warning signal. *Science*, 158(3798):264–265, October 1967.

[2] Biohazard symbol history. http://www.hms.harvard.edu/orsp/coms/biosafetyresources/history-of-biohazard-symbol.htm.

[3] P. E. Black. Software facts. http://swaconsortium.org/projects/softwareFacts/softwareFacts.html.

[4] P. E. Black. Software assurance with SAMATE reference dataset, tool standards, and studies. In *IEEE/AIAA 26th Digital Avionics Systems Conference (DASC'07)*, pages 6.C.1 1–6. IEEE, October 2007.

[5] P. E. Black, M. Kass, M. Koo, and E. Fong. Source code security analysis tool functional specification version 1.1. Special publication 500-268 v1.1, NIST, February 2011.

[6] J. Bloch. Extra, extra – read all about it: Nearly all binary searches and mergesorts are broken. http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html, June 2006.

[7] L. C. Briand, K. E. Emam, B. G. Freimut, and O. Laitenberger. A comprehensive evaluation of capture-recapture models for estimating software defect content. *IEEE Transactions on Software Engineering*, 26:518–540, 2000.

[8] A. Chao. An overview of closed capture-recapture models. *Journal of Agricultural, Biological, and Environmental Statistics*, 6(2):158–175, 2001.

[9] L. C. Chosewood, MD and D. E. Wilson, DrPH, CBSP, editors. *Biosafety in Microbiological and Biomedical Laboratories (BMBL).* U.S. Department of Health and Human Services, Centers for Disease Control and Prevention, and National Institutes of Health, 5th edition, 2009. http://www.cdc.gov/biosafety/publications/bmbl5/.

[10] S. Christey. Chains and composites. The MITRE Corporation, 2008.

[11] S. Christey and C. Harris. Introduction to vulnerability theory. The MITRE Corporation, 2009. http://cwe.mitre.org/documents/vulnerability_theory/intro.html.

[12] Common vulnerabilities and exposures (CVE). The MITRE Corporation. http://cve.mitre.org/.

[13] Common weakness enumeration (CWE). The MITRE Corporation. http://cwe.mitre.org/.

[14] Computer forensics tool testing (CFTT) project web site. http://www.cftt.nist.gov/.

[15] D. Engler. Finding bugs with system-specific static analysis. Keynote, 2002 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering (PASTE'02), Nov 2002. http://www.stanford.edu/~engler/.

[16] E. Fong, R. Gaucher, V. Okun, P. E. Black, and E. Dalci. Building a test suite for web application scanners. In *Proc. 41th Annual Hawaii International Conference on System Sciences (HICSS'08)*, page 478. IEEE, January 2008. http://samate.nist.gov/docs/wa_paper2.pdf.

[17] E. Fong and V. Okun. Web application scanners: Definitions and functions. In *Proc. 40th Annual Hawaii International Conference on System Sciences (HICSS'07)*, page 280b. IEEE, January 2007. http://samate.nist.gov/docs/wa_paper.pdf.

[18] J. Hey. The mind of the species problem. *Trends in Ecology & Evolution*, 16(7):326–329, 2001.

[19] International Software Testing Qualification Board. Certified tester foundation level syllabus edition 2011. http://istqb.org/display/ISTQB/Foundation+Syllabus.

[20] M. Kass. Leveraging automation to verify voting system source code conformance to the VVSG 1.0 workmanship requirements. NIST, 2011. to be published.

[21] K. J. Kratkiewicz. Evaluating static analysis tools for detecting buffer overflows in c code. Master's thesis, Harvard University, 2005.

[22] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1:323–337, December 1992.

[23] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, December 1976.

[24] NIST general information. http://www.nist.gov/public_affairs/general_information.cfm.

[25] Office of workforce management. http://www.nist.gov/hrmd/.

[26] V. Okun, A. Delaitre, and P. E. Black. The second static analysis tool exposition (SATE) 2009. NIST Special Publication 500-287, NIST, 2010.

[27] V. Okun, R. Gaucher, and P. E. Black. Static analysis tool exposition (SATE) 2008. NIST Special Publication 500-279, NIST, June 2009.

[28] V. Okun, W. F. Guthrie, R. Gaucher, and P. E. Black. Effect of static analysis tools on software security: Preliminary investigation. In *Proc. 2007 ACM workshop on Quality of Protection (QoP'07)*, pages 1–5. ACM, 2007.

[29] H. Petersson, T. Thelin, P. Runeson, and C. Wohlin. Capture-recapture in software inspections after 10 years research–theory, evaluation and application. *Journal of Systems and Software*, pages 249–264, 2004.

[30] SAMATE reference dataset. NIST. http://samate.nist.gov/SRD.

[31] 2010 CWE/SANS top 25 most dangerous software errors. version 1.08. Steve Christey ed. http://cwe.mitre.org/top25/archive/2010/2010_cwe_sans_top25.pdf, March 2010.

[32] J. Viega. *CLASP Reference Guide*. Secure Software, Inc., volume 1.1 edition, 2005.

[33] Voluntary voting system guidelines version 1.0 (2005): Volume I voting system performance guidelines. Election Assistance Commission (EAC), 2005. http://www.eac.gov/testing_and_certification/voluntary_voting_system_guidelines.aspx.

[34] Web Application Security Consortium (WASC). Web application security scanner evaluation criteria. http://projects.webappsec.org/w/page/13246986/Web-Application-Security-Scanner-Evaluation-Criteria, 2011.

[35] C. Willis. CAS static analysis tool study overview. In *Proc. Eleventh Annual High Confidence Software and Systems Conference*, page 86. National Security Agency, 2011. http://hcss-cps.org/.

[36] Y. Wu, R. A. Gandhi, and H. Siy. Using semantic templates to study vulnerabilities recorded in large software repositories. In *Proc. 2010 ICSE Workshop on Software Engineering for Secure Systems*, SESS '10, pages 22–28, New York, NY, USA, 2010. ACM.

[37] M. Zitser, R. P. Lippmann, and T. Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *Proc. 12th Internt'l Symp. on Foundations of Software Engineering*, pages 97–106. ACM SIGSOFT, 2004.