Testing Static Analysis Tools using Exploitable Buffer Overflows from Open Source Code *

Misha Zitser
D. E. Shaw Group
New York, NY
zitserm@deshaw.com

Richard Lippmann
MIT Lincoln Laboratory
Lexington, MA
rpl@ll.mit.edu

Tim Leek
MIT Lincoln Laboratory
Lexington, MA
tleek@ll.mit.edu

ABSTRACT

Five modern static analysis tools (ARCHER, BOON, PolySpace C Verifier, Splint, and UNO) were evaluated using source code examples containing 14 exploitable buffer overflow vulnerabilities found in various versions of Sendmail, BIND, and WU-FTPD. Each code example included a "BAD" case with and a "OK" case without buffer overflows. Buffer overflows varied and included stack, heap, bss and data buffers; access above and below buffer bounds; access using pointers, indices, and functions; and scope differences between buffer creation and use. Detection rates for the "BAD" examples were low except for PolySpace and Splint which had average detection rates of 87% and 57%, respectively. However, average false alarm rates were high and roughly 50% for these two tools. On patched programs these two tools produce one warning for every 12 to 46 lines of source code and neither tool accurately distinguished between vulnerable and patched code.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: [Software/Program Verification]; D.2.5 [**Software Engineering**]: [Testing and Debugging]; K.4.4 [**Computers and Society**]: [Electronic Commerce]

General Terms

Measurement, Performance, Security, Verification

Keywords

Security, buffer overflow, static analysis, evaluation, exploit, test, detection, false alarm, source code

*This work was sponsored by the Advanced Research and Development Activity under Air Force Contract F19628-00-C-0002. Opinions, interpretations, conclusions, and recommendations are those of the authors and are not necessarily endorsed by the United States Government.

SIGSOFT'04/FSE-12, Oct. 31–Nov. 6, 2004, Newport Beach, CA, USA.

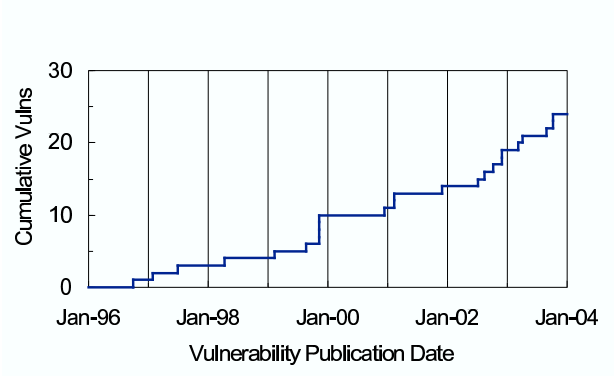

Figure 1: Cumulative buffer overflow vulnerabilities found in BIND, WU-FTPD, and Sendmail server software since 1996

1. INTRODUCTION

The Internet is constantly under attack as witnessed by recent Blaster and Slammer worms that infected more than 200,000 computers in a few hours [19, 25]. These, and many past worms and attacks exploit buffer overflow vulnerabilities in server software. The term buffer overflow is used in this paper to describe all types of out-of-bound buffer accesses including accessing above the upper limit or below the lower limit of a buffer.

Buffer overflow vulnerabilities often permit remote attackers to run arbitrary code on a victim server or to crash server software and perform a denial of service (DoS) attack. They account for roughly 1/3 of all the severe remotely exploitable vulnerabilities listed in the NIST ICAT vulnerability database [22]. The often-suggested approach of patching software as quickly as possible after buffer overflow vulnerabilities are announced is clearly not working given the effectiveness of recent worms. Figure 1 shows the dates that new remotely exploitable buffer overflow vulnerabilities were announced in three popular Internet server software applications (BIND, WU-FTP, and Sendmail) and the cumulative number of these vulnerabilities. For just these three servers, there have been from one to six remotely exploitable buffer-overflow vulnerabilities announced each year, no reduction in the rate of new vulnerabilities, and a total of 24 vulnerabilities published since 1996.

A detailed review of approaches that have been developed to counter buffer overflow exploits is available in [30].

These include static analysis to discover and eliminate buffer overflows during software development, dynamic analysis to discover buffer overflows during software testing, dynamic prevention to detect buffer overflows when they occur after software has been deployed, and the use of memory-safe languages. Static analysis is the only approach that eliminates both buffer overflows and their effects and that can be applied to the vast amounts of open-source legacy C code in widely-used open-source software. Dynamic testing is expensive and almost always cannot exercise all code paths. Dynamic prevention approaches such as Stackguard, CCured, and CRED [9, 11, 24, 23] detect some buffer overflows at run time, only to turn them into DoS attacks because a program halts in order to prevent a buffer overflow. Similarly, safe languages such as Java, LISP, and ML check buffer accesses at runtime, raising exceptions at any out-of-bounds attempts.

Many static analysis tools that detect buffer overflows in source code have been recently developed, but we are aware of no comprehensive evaluations. Most past evaluations were performed by tool developers, use few examples, and do not measure both detection and false alarm rates of tools [14, 15, 27, 29]. Although some studies apply tools to large amounts of source code and find many buffer overflows [29], the detection rate for in-the-wild exploitable buffer overflows is still not known and the false alarm rate is difficult to assess.

We are aware of only three evaluations of tools that were not performed by tool developers. A qualitative survey of lexical analysis tools that detect use of functions often associated with buffer overflows is available in [20]. A single tool for detecting buffer overflows is evaluated in [21], described as "a tool created by David Wagner based upon the BANE toolkit." Presumably, this is BOON [27]. While the authors comment about excessive false positives and false negatives, they do not attempt to quantify them. The study described in [16] is more objective. It compares Flawfinder, ITS4, RATS, Splint, and BOON on a testbed of 44 function invocations, both safe and unsafe. They carefully count true positives and false positives for examples of "20 vulnerable functions chosen from ITS4's vulnerability database ... Secure programming for Linux and UNIX_HOWTO, and the whole `[fvsn]printf` family". These examples contain no complex control structures, instances of inter-procedural scope, or direct buffer accesses outside of string functions, and therefore cannot represent complex buffer access patterns found in Internet servers. However, this study is diagnostic. It exposes weaknesses in particular implementations (e.g. BOON cannot discriminate between a good and a bad strcpy even in its simplest form). High detection/false alarm rates are reported for the three purely lexical tools, Flawfinder, ITS4, and RATS, and lower detection/false alarm rates for the more sophisticated Splint and BOON. They also do not report the conditional probability of no false alarm in a corrected program given a detection in the vulnerable version. This conditional probability is important because it measures the ability of a tool to discriminate between safe and unsafe versions of the same code.

The purpose of the research described in this paper was to perform an unbiased evaluation of modern static analysis tools that can detect buffer overflows. This evaluation measures detection and false alarm rates using a retrospective collection of 14 remotely-exploitable buffer overflows selected from open-source server software. A secondary goal of this work was to characterize these in-the-wild buffer overflows in terms of type (e.g. stack, heap, static data) and cause (e.g. improper signed/unsigned conversions, off-by-one bounds check error, use of unsafe string function). A final goal was to provide a common collection of realistic examples that can be used to aid in the development of improved static analysis.

Tools	Analysis Strategy
ARCHER [29]	Symbolic, interprocedural, flow-sensitive analysis.
BOON [27]	Symbolic, interprocedural flow-insensitive analysis only strings.
PolySpace C Verifier [1]	Abstract interpretation, interprocedural, flow-sensitive.
SPLINT [14]	Lightweight static analysis, intraprocedural.
UNO [15]	Model checking, interprocedural, flow-sensitive.

Table 1: Static Analysis tools used in the evaluation

2. STATIC ANALYSIS TOOLS

Table 1 provides a summary of the five static analysis tools used in this evaluation. Four are open-source tools (ARCHER, BOON, SPLINT, UNO) and one is a commercial tool (PolySpace C Verifier). All perform in-depth symbolic or abstract analysis of source code and all detect buffer overflows. Simpler lexical analysis tools such as RATS and ITS4 [5, 26] were excluded from this study because they have high false alarm rates and limited scope.

ARCHER (ARray CHeckER) is a recently developed static analysis tool that has found many memory access violations in LINUX kernel and other source code [29]. It uses a bottom-up inter-procedural analysis. After parsing the source code into abstract syntax trees, an approximate call graph is created to determine an order for examining functions. Starting at the bottom of the call graph, symbolic triggers are calculated to determine ranges for function parameters that result in memory access violations. These triggers are used to deduce new triggers for the callers, and so on. Once the top-most caller is reached, if any of its triggers are satisfied, a memory violation flag is raised. Error detection is conservative with overflows reported only with strong evidence, and in some kind of rank order. Archer employs "several heuristics to minimize the number of false positives", which likely hamper its ability to detect some classes of overflows accurately. The analysis is further limited because function pointers are not modeled, heuristics are used to analyze loops, and only simple range constraints are considered. ARCHER was used to analyze 2.6 million lines of open-source code and generated 215 warnings. Of these, 160 were true security violations and 55 were false alarms [29].

BOON (Buffer Overrun detectiON) models manipulation of string buffers, through direct access as well as a subset of standard library functions [27]. Every string is modeled by a pair of integers - the number of bytes allocated for the

storage buffer and the actual number of bytes used. For each use of a string function, an integer range constraint is generated. Constraints are collected across a program, ignoring control flow and the order of statements, and used to detect accesses outside string boundaries. This analysis is limited because it only considers strings and is flow insensitive. BOON was applied to source code from Sendmail 8.9.3 and generated 44 warnings [27]. Only 4 of these were actual buffer overflows.

PolySpace C Verifier is a commercial tool designed to detect run-time errors in embedded software [1]. Few details of the algorithm are available other than the fact that it uses "abstract interpretation", although company represenatives have informed us that the algorithms are based upon the research of Patrick Cousot and Alain Deutsch [10, 12, 13]. In a white paper, PolySpace describes its tool in this way:

> Abstract Interpretation had to wait for the implementation of very efficient and non-exponential algorithms, and for the availability of increased processing power on modestly equipped computers. When applied to runtime error detection, Abstract Interpretation performs an exhaustive analysis of all risky operations and automatically provides a list of runtime errors contained in a program before it is run, tested or shipped. [1]

The only evaluation of the Polyspace C Verifier tool that we are aware of is described in [8]. This tool was applied to NASA software used in the Mars Exploration Rover. Software had to be manually broken up into 20-40k lines-of-code blocks because the analysis couldn't scale to larger code segments. An upper bound on the false alarm rate derived from the number and types of alerts generated is one false alarm for every 30 to 60 lines of code. False alarms, however, were not verified and the miss rates for different alert types (e.g. uninitialized variables, buffer overflows, zero divides) were not measured.

SPLINT (Secure Programming Lint) extends LCLINT to detect buffer overflows and other security violations [14]. It uses several lightweight static analysis techniques. SPLINT requires source annotations to perform inter-procedural analysis, but even without annotations, it monitors the creation of and accesses to buffers and detects bounds violations. SPLINT uses heuristics to model control flow and common loop constructs. The developers used SPLINT to analyze WU-FTP source code without annotations and generated 166 warnings [14]. Of these, 25 were real and 141 were false alarms.

UNO is named for the three software defects it was designed to detect: the use of **U**ninitialized variables, dereferencing **N**il-pointers, and **O**ut-of-bound array indexing [15]. UNO uses a public-domain compiler extension named ctree to generate a parse tree for each procedure in a program. Parse trees are turned into control flow graphs that are analyzed using a model checker to find array indexing errors. UNO does not check array indices that involve complicated expressions or function calls. It only performs checks when a bound on the index can be determined or the index is a constant. Ranges for variables are deduced from assignments and conditions and combined in a conservative fashion. The analysis is not inter-procedural. UNO was applied to two open-source software applications (Sendmail and unravel) but detected no array indexing errors [15]. Overall, it produced 58 warnings for variables that were declared but not used or initialized. Only 5 of these were false alarms.

3. OPEN SOURCE TEST CASES

Three widely-used open-source programs were chosen to test the effectiveness of these five static analysis tools: BIND, WU-FTPD, and Sendmail. BIND [4] is the most popular DNS server, WU-FTPD [7] is a popular FTP daemon, and Sendmail [6] is the dominant mail transfer agent. The fourteen most recent severe buffer overflow vulnerabilities for these servers were selected for a retrospective analysis. Eleven of these allow a remote attacker to gain full control of the system running the vulnerable software and to execute arbitrary code. The goal of this retrospective analysis was to determine if any static analysis tool could have detected these vulnerabilities and been able to prevent their exploitation if employed during development.

As a first step, we tried to gauge how easy it is to use these tools on the sorts of programs we care about using a vulnerable version of Sendmail (8.12.4), weighing in at more than 145 thousand lines of code. Splint issued many parse errors regarding type definitions like `u_char` and `u_long`, even though all of the types in question were defined either in Sendmail include files or standard C include files. We were ultimately unable to convince Splint to analyze all of Sendmail [1]. ARCHER was able to parse the source, but it terminated with a `Division_by_zero` exception during analysis. PolySpace's C Verifier was similarly uncooperative. After considerable consultation with PolySpace Support, the analysis ran for four days before raising some fatal internal error [2].

This initial experience was disappointing; it suggested we would not be able to run the tools on large and complex programs like Sendmail. As an alternative we crafted self-contained *model* programs by extracting just as much code as was required to reproduce the buffer overflow vulnerability. These model programs were small enough to permit successful analysis with all of the tools, and ranged in size from 90 to 800 lines [3]. Every attempt was made to preserve the general structure and complexity of the vulnerable code when creating these models. If a buffer was declared in one function and overflowed in another, or if it was accessed via some complicated loops and conditionals, then the model did so as well. It was especially difficult to extract code when the vulnerability involved multiple procedure calls. On average, five to seven hours were required to construct each model program. In addition, we arranged for inputs to each model program that demonstrated a buffer overflow.

For each of the fourteen analyzed vulnerabilities, two model programs were constructed: a BAD version and an OK version. The BAD version contained one or more buffer overflow vulnerabilities modeling those seen in the real program. These vulnerabilities were corrected in the OK version of the model program via whatever strategy was indicated by the patch file distributed by the code maintainers. We had no analysis that could verify the completeness of the patches so there is the possibility that vulnerabilities still remain in the OK versions of the programs. However, for each OK model,

[1]David Evans supplied a page-long list of definitions necessary for processing sendmail

[2]The machine was a 2.66GHz Xeon with 2GB RAM

[3]Program size reported by sloccount [28]

we did verify that the input that revealed the overflow in the BAD model did not provoke an overflow in the OK model.

The following three sections describe vulnerabilities in BIND, Sendmail, and WU-FTP used to create model programs. Further details on these vulnerabilities and model programs, including descriptions, extracted source code, and vulnerable version numbers, are available in [30].

3.1 Bind

Four serious buffer overflow vulnerabilities in BIND shown in Table 2 were used to create model programs. These were discovered between 1999 and 2001 and affected BIND Version 4 up to 4.9.8 and BIND Version 8 up to 8.2. In the table, vulnerabilities are listed by a simple name (e.g. BIND-1), a common name (e.g. NXT record), a CVE (or CAN) number when available [3] or a CERT Advisory number for older vulnerabilities when no CVE number is available [2], a code that indicates the type of vulnerability, and a short description of the reason for the overflow. The code RC stands for Remote Compromise, the code RD stands for Remote DoS, and the code LC stands for Local Compromise. These codes indicate the location of the attacker and the result of the exploit. An attack on a server can either be issued from a remote machine or locally from the server and the attacker either achieves a high level of privilege (usually root or the privilege of the server) and can execute arbitrary code or the attacker disables the server. The RC code indicates the most critical vulnerabilities. The BIND-1 RC vulnerability (BIND-1) was responsible for the widespread Lion Internet worm [18].

3.2 Sendmail

As noted above, Sendmail is currently the most widely used mail transfer agent. The seven serious Sendmail buffer overflow vulnerabilities shown in Table 2 were used to create model programs. They were discovered between 1996 and 2003 and affect Sendmail versions up to 8.12. These include five RC vulnerabilities that permit a remote attacker to execute arbitrary code and two LC vulnerabilities that allow a local user to obtain root level privileges. Reasons for these buffer overflows are complex and include many logic errors, incorrect assumptions about the validity of input data, and typographic errors where one variable name was mistakenly used for another.

3.3 Wu-ftpd

The WU-FTPD FTP server is installed and enabled by default on many Linux operating systems including RedHat and Slackware. Three buffer overflow vulnerabilities in WU-FTPD shown in Table 2 were selected for this study. They were discovered between 1999 and 2003 and affected WU-FTP versions up to 2.6.2. They were caused by missing checks on array bounds for the strcpy() function and incorrect logic. All three are RC vulnerabilities that were again used to create model programs.

4. OVERFLOW CHARACTERISTICS

Buffer overflows in the fourteen model programs were characterized to obtain statistics on the types of buffer overflows that occur in real programs and are exploitable. It was found that buffer overflows within each individual model program were often similar and that they were sometimes repeated many times. For example, for the SM-1 model program, there were 28 buffer overflows of the same buffer that were identical with regard to the features used in Table 3. It is likely that an actual static analysis tool would detect none or all of these similar buffer overflows and that a programmer would also correct all or none. Results in Table 3 reflect this assumption and do not count identical buffer overflows in one model program individually. Instead, the relative frequencies of the observed values in Table 3 were first calculated separately for each model program weighting each buffer overflow uniformly when computing relative frequencies. Following this, overall relative frequencies were calculated by weighting relative frequencies uniformly for all model programs. The results, giving each model program a weight of one, appear in Table 3 and indicate that there is considerable variety in real buffer overflows. Most out-of-bound accesses exceed the upper bound, but one is below the lower bound. Most involve character arrays, but many involve `u_char` arrays. The buffer is on the stack for roughly 3/4 of the overflows but on the heap, bss, or data segments roughly 1/4 of the time. The difference in scope between where the buffer is declared and where it is accessed is inter-procedural roughly 40% of the time, intra-procedural half the time, and otherwise global. Most buffers are not inside a container, but a small percentage (7%) are in unions. Most (67%) of array accesses use a string manipulation function that includes a limit (e.g. `strncpy`) or access the array directly with an index (e.g. `array[i]`). For these, the index or limit is a variable most of the time, but can also be a linear expression or the contents of an integer array. Many (56%) of the buffer overflows are caused by incorrect use of a string manipulation function (e.g. strcpy, memcpy), and the rest are caused by direct accesses using pointers or an index. Buffers are accessed directly for only 1/3 of the overflows while 2/3 of the overflows use indirec-

Characteristic	Observed Values
Bound	93 % upper, 7% lower
Type	64% char, 36% u_char
Location	73% stack, 16% bss, 7% heap, 4% data
Scope	43% inter-procedural, 52% same function, 5% global buffer
Container	93% none, 7% union
Index or limit	64% none, 22% variable, 7% linear exp, 7% contents of buffer
Access	56% C function, 26% pointer, 11% index, 7% double de-reference
Buffer alias	52% alias, 34% no alias, 14% alias of an alias
Control flow	29% none, 49% if-statement, 22% switch
Surrounding loops	46% none, 42% while, 5% for, 7% nested
Input taint	64% packet, 22% dir functions, 7% file 7% argc/argv

Table 3: Characteristics of buffer overflows

Simple name	Common name	Ref	Type	Reason
BIND-1	NXT record	CA-1999-14	RC	Size arg of `memcpy` not checked.
BIND-2	SIG record	CA-1999-14	RD	negative arg to `memcpy` underflows to large positive int
BIND-3	iquery	CVE-1999-0009	RC	Size arg of `memcpy` not checked
BIND-4	nslookupComplain	(CVE-2001-0013)	RC	Use of sprintf() without proper bounds checking.
SM-1	crackaddr	CA-2003-07	RC	Upper bound increment for a `>` char but not decrement for `<`
SM-2	gecos	CVE-1999-0131	LC	gecos field copied into fixed-size buffer without size check
SM-3	8.8.0/8.8.1 mime	CVE-1999-0206	RC	Pointer to buffer not reset to beginning after line read.
SM-4	8.8.3/8.8.4 mime	CVE-1999-0047	RC	Typo prevents a size check from being performed.
SM-5	prescan	CA-2003-12	RC	Input byte set to `0xff` cast to minus one error code.
SM-6	tTflag	CVE-2001-0653	LC	Negative index passes size check but causes underflow.
SM-7	TXT record	CVE-2002-0906	RC	Size for `strncpy` read from packet header but not checked.
FTP-1	mapped chdir	CVE-1999-0878	RC	Several `strcpy` calls without bounds checks.
FTP-2	off-by-one	CAN-2003-0466	RC	Wrong size check inside `if >` should really be `>=`
FTP-3	realpath	CVE-1999-0368	RC	Several unchecked `strcpy` and `strcat` calls.

Table 2: Vulnerabilities in bind, sendmail, and wu-ftpd

```
int main(int argc, char *argv[]) {
  ...
  /* name is tainted and can be very long */
  char *name;
  name = argv[1];
  call_realpath(name);
}

void call_realpath(char *name){
  ...
  char path[MAXPATHLEN + 1];
  ...
  my_realpath(name,path,chroot_path);
}

char *my_realpath (const char *pname, char *result,
                   char* chroot_path) {
  char curpath[MAXPATHLEN];
  ...
  /*BAD*/
  strcpy(curpath, pname);
  ...
}
```

Figure 2: Source fragment extracted from model of FTP-3 containing one buffer overflow.

tion caused by aliases. The local surrounding control flow includes an if statement or a switch statement for roughly 70% of the overflows and a surrounding loop for roughly half of the overflows. Finally, tainted input from users that can cause the buffer overflow to occur comes from Internet packets for roughly 2/3 of the overflows but also from directory functions (e.g. `getcwd` and `pwd`), from file inputs, and from command line arguments.

Figures 2 and 3 contain fragments of model source to illustrate their complexity. Figure 2 contains a code fragment from FTP-3 in which a command-line argument is read in, passed through two functions, and eventually copied into a fixed size buffer with no length check. The comment `/* BAD */` has been inserted immediately before the line

```
ADDRESS *recipient(...) {
  ...
  else {
  /* buffer created */
    char nbuf[MAXNAME + 1];
    buildfname(pw->pw_gecos,
               pw->pw_name, nbuf);
     ...
  }
}

void  buildfname(gecos, login, buf)
  register char *gecos;
  char *login;
  char *buf; {
  ...
  register char *bp = buf;
  /* fill in buffer */
  for (p = gecos; *p != '\0' &&
                  *p != ',' &&
                  *p != ';' &&
                  *p != '%'; p++) {
    if (*p == '&') {
      /* BAD */
      (void) strcpy(bp, login);
      *bp = toupper(*bp);
      while (*bp != '\0')
       bp++;
    }
    else
      /* BAD */
      *bp++ = *p;
  }
  /* BAD */
  *bp = '\0';
}
```

Figure 3: Source fragment extracted from model of SM-2 containing three buffer overflows.

with the buffer overflow. This example illustrates how a local user can cause a buffer overflow. Using features from Table 3, this buffer overflow is classified as: exceeds upper bound, char variable, on stack, buffer declaration and use in same scope, no container, no index computation, string function, no alias, no local control flow, no loop, and tainted input from the command line. This characterization, however, inadequately reflects the difficulty of analyzing the code. First, a taint analysis must understand that the string pointed to by name can be any length. Then, an interprocedural analysis must follow this pointer through two functions to where it is used to copy the name string into a fixed-length buffer. Our characterization does not measure the complexity of following the tainted string through the program or of identifying tainted input as it is read in.

Figure 3 contains a code fragment from SM-2. It contains three lines with potential buffer overflows all preceded by the comment line `/* BAD */`. The bottom two buffer overflows occur when the real name from the gecos field in the passwd file is copied into a fixed length buffer with no length check. Using features from Table 3, these are both classified as: exceeds upper bound, char variable, on stack, inter-procedural scope, no container, no index computation, pointer access, alias, in if statement, in for loop, and tainted input from a file. Both of these buffer overflows can be forced to occur by a local user because it is relatively easy to change the real name field in the password file to be a long string. The first buffer overflow copies another field in the password file that may be too long into a fixed length buffer. Characteristics of this buffer overflow are identical to those of the second two, except access to the buffer is through a string function instead of through a pointer. Detecting these buffer overflows requires understanding that two fields of the password structure (`pw_gecos`, `pw_name`) can point to long buffers, following pointers to these fields through multiple functions and aliases, and analyzing the loop and local control flow where these pointers are used to copy their contents into a fixed-length buffer with no bounds checks.

These two examples demonstrate the need for static analysis approaches that perform in-depth analyses of source code. A simple approach that is neither interprocedural nor flow sensitive will miss over half of vulnerabilities.

5. TEST PROCEDURES

Details of the test procedures are provided in [30] including command line settings for tools and scripts. No annotations were added to source code for any of the tools. The only modifications made were for PolySpace because buffer overflows were detected in library routines such as strcpy and not mapped into the main program to the point where the library routine was called. We corrected for this by adding to the model program as many copies (e.g. strcpy1, strcpy2) of a library function as there were calls to that function. This allowed us to map buffer overflow detections in these functions to call sites. Documentation, and often advice from tool developers, was used to determine appropriate flags and the environment for each tool.

The five tools were run on the fourteen pairs of BAD and OK model programs. Each BAD program had one or more lines in the code labeled BAD corresponding to the lines that could overflow a buffer for some input. The OK program employed the developers' patch. This sometimes resulted in a different number of BAD and OK lines.

System	$P(d)$	$P(f)$	$P(\neg f\|d)$
PolySpace	0.87	0.5	0.37
Splint	0.57	0.43	0.30
Boon	0.05	0.05	-
Archer	0.01	0	-
Uno	0	0	-

Table 4: Detection and flase alarm rates for all systems

Some tools provided a source code line number for each warning and we used this to count detections and false alarms. Only warnings for lines labeled BAD or OK in the model source code counted as detections or false alarms. For tools that did not provide a line number (e.g. BOON), we used the name and other buffer information to confirm that the correct buffer overflow or buffer access was detected.

6. RESULTS

Three performance measures were computed for each tool. For each run of a static analysis tool on a model program, we counted the number of times a line labeled "BAD" was correctly identified by inspecting the output of the tool. We called this the number of *detections* for that tool on that program, $C(d)$. Similarly, we counted the number of times a line labeled "OK" was incorrectly identified and called this the number of *false alarms* for the tool on the program, $C(f)$. Finally, we counted the number of times a detection was paired with a false alarm for a given BAD/OK pair of programs and called this the number of *confusions* for the tool on the program, $C(df)$. In Table 4 these counts are used to estimate probabilties of detection, $P(d)$, false alarm, $P(f)$, and *discrimination* (no confusion given a detection), $P(\neg f|d)$, according to the following formulae:

$$P(d) = C(d)/T(d) \quad (1)$$

$$P(f) = C(f)/T(f) \quad (2)$$

$$P(\neg f|d) = 1 - C(df)/C(d), \quad (3)$$

where $T(d)$ is the total number of detections possible for a model program, and $T(f)$ the total number of possible false alarms (note that $T(d) \neq T(f)$ is possible since correcting a vulnerability can change the number of buffer accesses).

Table 4 shows overall detection and false alarm rates for all systems. PolySpace and Splint detected a substantial fraction of buffer overflows while the other three tools generated almost no warnings for any model program. Boon had two confusions (detections combined with false alarms), one on each of SM-6 and FTP-1. Archer had one detection on SM-4 and no false alarms. UNO generated no warnings concerning buffer overflows. $P(d)$ for PolySpace and Splint are quite high at 0.87 and 0.57. False alarm probabilities are also high for these tools: both are near 0.5.

The information in Table 4 is also rendered graphically as a kind of ROC (Receiver Operating Characteristic) curve in Figure 4. Probability of detection and false alarm, $P(d)$ and $P(f)$, make up the vertical and horizontal axes in this plot. The diagonal line is the locus of points representing a naive system following the strategy of random guessing. By choosing different probabilities for labeling a line in a program as having a buffer overflow, this random system can

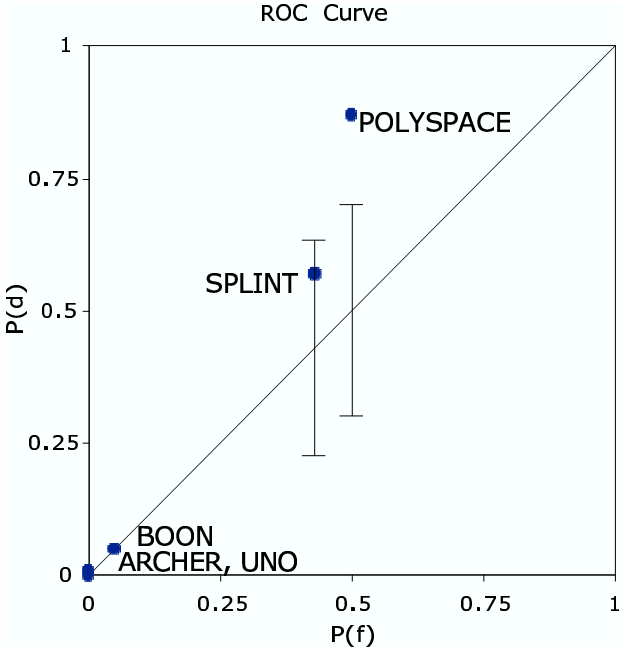

Figure 4: ROC-type plot for the five systems evaluated in this study. Only PolySpace has performance significantly better than the diagonal random guessing line.

achieve any performance for which $P(d) = P(f)$. A useful system must have an operating point that is substantially *above* this diagonal. Only PolySpace where $P(d) = 0.87$ and Splint where $P(d) = 0.57$ have points above the diagonal.

We further require that the vertical distance between an operating point and the diagonal be *statistically significant*. If a system randomly detects buffer overflows in the BAD/OK lines by flipping a biased coin then we would expect it to have an arbitrary $P(f)$, with a per-model-program variance given by $\sigma^2 = p(1-p)/N$ where $p = P(f)$ and N is the number of lines labeled BAD in the model program. The overall $P(d)$ is the average of the 14 per-model program average detection rates, and the variance of $P(d)$ is equal to the sum of the per-model program variances divided by $14^2 = 196$. The error bars in this figure are $\pm$ two standard deviations for random guessing systems with false alarm rates equal to those observed for Splint and PolySpace. From this we see that the detection rate of Splint is not outside the two standard deviation range, while that of PolySpace is substantially outside the range. Splint is thus not statistically significantly different at the 0.05 confidence level from a random guessing system that labels 43% of all lines BAD. The detection rate of PolySpace, however, is statistically greater than that of a random guessing system that labels 50% of all lines BAD.

The above analysis is incomplete, however, since $P(d)$ and $P(f)$ don't adequately capture how these tools might be used. We need to measure not only the ability of a tool to detect vulnerabilities, but also its ability to discriminate between the presence and the absence (patch) of vulnerabilities. If a system correctly detects every line of source code containing a buffer overflow, but is unable to notice that the overflow has been corrected, then a user of the system will not be able to determine whether a code modification designed to correct a problem is effective. Without the ability to validate patches that correct security weaknesses, a tool can only suggest potential problems, and it may not be used because its warnings are untrustworthy. We measured the probability of not false alarming on patched software as the conditional probability of not generating a false alarm on a corrected vulnerability, given a detection of the original vulnerability (see equation 3). These values have been calculated for Splint and PolySpace and are provided in Table 4 under the column labeled $P(\neg f|d)$. Note that an ideal system would have $P(\neg f|d) = 1.0$. For PolySpace and Splint, these conditional probabilities are 0.37 and 0.3, respectively. This means that more than half the time, these tools continue to signal a buffer overflow after it has been patched.

The above analyses focused only on source code lines in the model programs that were known to contain buffer overflows. Warnings caused by other lines of source code were ignored. Unfortunately, there were many such warnings. We counted the number of buffer-overflow related warnings generated by Splint and PolySpace for each of the 14 OK versions of the model programs. We used these counts to estimate the number of false alarms to expect from each tool per line of code. PolySpace produced one warning for every 12 lines of code and Splint produced one for every 46 lines of code. These are very high warning rates for *patched* programs, and they concur with our qualitative experience of the tools.

7. EXPLAINING FALSE ALARMS

This evaluation indicates that while state-of-the-art static analysis tools like Splint and PolySpace can find real buffer overflows with security implications, warning rates are unacceptably high. We can think of two ways to explain why a tool might signal a warning on a line labeled `OK`. First, the warning may really be a detection, meaning that the version of the model program created by applying the developers' patch is not perfectly guarded against the overflow for all inputs. In this case, static analysis is properly reporting that the vulnerability persists, and the reported *false alarm* rate is too high. Second, the warning may be a false alarm, and thus represents a blind spot for a particular static analysis tool.

In our examination of the false alarms in the model programs, there were certainly many situations in which the false alarms seemed genuine. These false alarms overwhelmingly involved buffer overflows subject to some complicated control flow (loops, nested conditionals, etc) which in turn is subject to the contents of either an array or structure. The model programs are excerpted from server code, which often includes logic to parse input: a pointer walks through the contents of a buffer in a manner that is determined by the contents of the buffer itself and may also affect the way in which another buffer is simultaneously populated with values.

As mentioned above, the OK models were too complicated to warrant against out-of-bounds accesses by inspection. So we decided to create for closer study a few contrived examples of the sort of program that looked like it was a problem for most of the tools. One such example, "aia2", appears in Figure 5, in which an array, `x`, is populated via a `for` loop with the values `-1,0,+1`. The *contents* of this array are sub-

```
1: int main () {
2:   int i;
3:   int x[3], y[2];
4:   x[0] = 1;
5:   for (i=0; i<3; i++)
6:     x[i] = i-1;
7:   for (i=1; i<3; i++)
8:     y[x[i]] = i;
9: }
```

Figure 5: False alarm example "aia2", in which the lack of overflow depends upon contents of an array.

```
1: int main () {
2:   char buf[] = "The";
3:   char *bp;
4:   bp=buf;
5:   while(1)
6:     if (!(*bp++))
7:       break;
8:   printf ("strlen(buf)=%d\n", bp-buf);
9: }
```

Figure 6: False alarm example "inp" in which the lack of overflow depends upon existence/non-existence of a null terminator in a string..

sequently used, again in a `for` loop, to index into another array, `y`. Even though `x` takes on a value `-1`, that value is never used to index into `y` because of the limits chosen for the loop. Notice that if we change the lower loop limit in line 7 to `i=0` then there is an underflow.

Another contrived example, "inp", of a false alarm involving the contents of a buffer appears in Figure 6. This example revolves around an implementation of `strlen` which will read beyond the upper bound of any string that is not null terminated. However, here the string is guaranteed to be null terminated because it is initialized with a string literal [17]. Notice that this example can be turned into a read overflow if we insert the line `buf[3]='a'` between lines 3 and 4: the string is no longer guaranteed to be properly terminated.

We paired these two `OK` example programs with `BAD` ones created by making the two modifications described above. Then we tried the five static analysis tools evaluated in this study on these four tiny programs. As seen in Table 5, the subjective assessment that these static analysis tools generate false alarms on buffer accesses subject to the contents of an array seems to be well supported. Archer and Uno neither detect anything nor false alarm on anything. Boon registers both a detection and a false alarm on the "inp" example. Splint registers both but only on the "aia2" example. PolySpace registers both for both examples. In all cases, when a static analysis tool finds an error in the `BAD` example, it also false alarms on the `OK` one, meaning that $P(\neg f|d) = 0$ for Boon, Splint, and PolySpace. For these examples, discrimination could not be any worse. Unfortunately, this sort of code is not only common in server software, it is essential, and it is hard to imagine how to do without it.

System	aia2	inp
Archer		
Boon		df
PolySpace	df	df
Splint	df	
Uno		

Table 5: Performance of five evaluated tools on false alarm examples

8. DISCUSSION

The performance of five modern static analysis tools was analyzed using 14 model programs to determine both detection rates for known buffer overflows and false alarm rates for patched source code after these buffer overflows have been eliminated. The model programs were generated by analyzing 14 serious buffer overflow vulnerabilities in BIND, WU-FTP, and Sendmail and then hand extracting source code required to reproduce these vulnerabilities. It was necessary to excerpt in this way because the majority of the tools could not operate upon full programs.

These experiments are the first we are aware of that carefully measure *detection*, *false alarm*, and *discrimination* rates for in-the-wild buffer overflows and that analyze characteristics of such overflows. The results demonstrate that simple static analysis techniques do not detect buffer overflows that occur in Internet server software. The detection rates of three of the five systems tested were below 5% when tested on C source code modeled after those sections of open-source C WU-FTP, Sendmail, and BIND server software that contain known and exploitable buffer overflows. These poor detection rates may be due to the complexity of analyzing real buffer overflows. An analysis of the overflows in this server software indicates that they differ in many characteristics. Only roughly half of them involve string manipulation routines, two-thirds involve buffers on the stack, half involve inter-procedural scope difference between locations where buffers are created and used, and about half involve pointers that are aliases of the original buffer. Finally, one vulnerability was a buffer underflow, many were inside loops, and some buffers were in unions. These results suggest that static analysis tools must be designed to handle complex buffer accesses, since they occur in quantity in actual code. They should also determine when a buffer access is tainted and can be forced to occur by external inputs. All of the in-the-wild buffer overflows were tainted.

Even though two static analysis tools (Splint and PolySpace) had high detection rates of 87% and 57%, they are not without problems. These tools would have detected some in-the-wild buffer overflows, but warnings generated by them might have been ignored by developers annoyed by high false alarm rates. The false alarm rate measured just on the patched lines in the model programs was 43% and 50%. More concerning, perhaps, is the rate of false alarms per line of code, which for these tools is 1 in 12 and 1 in 46. Additionally, these tools do not appear to be able to discriminate between vulnerable source code and patched software that is safe, making them of little use in an iterative debugging loop. We estimate the discrimination rates, i.e. the probability of not warning about a buffer overflow in a properly patched line of code, for these two tools at

less than 40%, as compared with 100% for an ideal system and 50% for a trivial system that flips a biased coin. Finally, the tool with the best performance, PolySpace is slow enough to preclude common use; it takes days to analyze a medium-sized program of 100,000 lines.

The results are promising because some static analysis tools would have detected in-the-wild buffer overflows. They are disappointing because false alarm rates are high and discrimination is poor. These results suggest that further work developing static analysis tools to detect buffer overflows should include testing on complex buffer overflows found in actual software and the careful measurement of detection, false alarm, and discrimination rates. To this end, we plan to release the 14 model programs used in this study for use by developers and evaluators. [4] In addition, we are developing a library of much simpler test cases that explore buffer overflows differing along the dimensions used to create Table 3. When developed, such test cases can be used to better diagnose the capabilities and limitations of existing and new static analysis tools.

Our evaluation also suggests that static analysis tools should perform a taint analysis that tracks external inputs to a program from packets, files, command-line arguments, and system calls. Any buffer overflow that is affected by these inputs, especially inputs that can be affected by remote attackers, is more critical than others. Further, static analysis tools should be designed to accommodate large complex programs such as Sendmail, WU-FTP, and BIND without extensive tuning, modification, or changes to the build environment. None of the best tools could analyze a program as big and complicated as Sendmail. And only ARCHER was able to impersonate gcc in makefiles (it uses the front-end of gcc to generate its abstract syntax trees), requiring no changes in the build environment.

The above results and conclusions should be interpreted only in the context of these experiments. They are based on model programs that contain already discovered buffer overflows that occur in Internet server source code. The model programs extract only the part of the source essential to replicate and represent the out-of-bounds buffer accesses. The false alarm rates reported here cannot be conclusively verified, for they assume that developers creating patches to address specific vulnerabilities actually succeed in doing so completely. Both detection and false alarm rates, further, may be unrepresentative of that for the remainder of the server code that was excluded in extracting the model programs. In addition, this study focuses on BIND, Sendmail, and WU-FTP Internet server software. The results may not generalize to other types of server software (e.g. database servers, web servers) or to other types of software (e.g. operating system kernel, word processing, numerical simulation, or graphics).

9. ACKNOWLEDGEMENTS

We would like to thank Robert Cunnigham, Roger Khazan, Kendra Kratkiewicz, and Jesse Rabek for discussions on static analysis. We would also like to thank David Evans for his help with Splint, David Wagner for answering questions about BOON, Yichen Xie and Dawson Engler for their help with ARCHER, and Chris Hote and Vince Hopson for all their help on answering questions about C-Verifier.

[4]Please direct all inquiries to tleek@ll.mit.edu.

10. REFERENCES

[1] Abstract interpretation. http://www.polyspace.com/downloads.htm, September 2001.

[2] Cert coordination center. http://www.cert.org/advisories, October 2003.

[3] Common vulnerabilities and exposures. http://www.cve.mitre.org, October 2003.

[4] Internet software consortium – bind. http://www.isc.org/products/BIND, October 2003.

[5] Secure software, rough auditing tool for security (rats). http://www.securesoftware.com, October 2003.

[6] Sendmail consortium. http://www.sendmail.org, October 2003.

[7] Wu-ftp development group. http://www.wu-ftpd.org, October 2003.

[8] G. Brat and R. Klemm. Static analysis of the mars exploration rover flight software. In *Proceedings of the First International Space Mission Challenges for Information Technology*, pages 321–326, 2003.

[9] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer. Ccured in the real world. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 232–244. ACM Press, 2003.

[10] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976.

[11] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, January 1998.

[12] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 230–241, 1994.

[13] A. Deutsch. On the complexity of escape analysis. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 358–371, 1997.

[14] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Softw.*, 19(1):42–51, 2002.

[15] G. Holzmann. Static source code checking for user-defined properties. Pasadena, CA, USA, June 2002.

[16] M. K. J. Wilander. A comparison of publicly available tools for static intrusion prevention. In *Proceedings of the 7th Nordic Workshop of Secure IT Systems*, 2002.

[17] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, Murray Hill, NJ, 2nd edition, 1988.

[18] W. S. M. Fearnow. Sans institute – lion worm. http://www.sans.org/y2k/lion.htm, April 2001.

[19] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. The spread of the sapphire/slammer worm. Technical report, CAIDA, ICSI, Silicon Defense, UC Berkeley EECS and UC San Diego CSE, 2003.

[20] J. Nazario. Source code scanners for better code. *Linux Journal*, 2002.

[21] E. O. P. Broadwell. A comparison of static analysis and fault injection techniques for developing robust system services. Technical report, University of California, Berkeley, May 2002.

[22] T. G. P. Mell, V. Hu. Nist icat metabase. http://icat.nist.gov, October 2003.

[23] M. Rinard, C. Cadar, D. Dumitran, D. Roy, and W. Beebee. Enhancing availability and security through failure-oblivious computing. Technical Report 935, Massachusetts Institute of Technology, 2004.

[24] O. Ruwase and M. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS 2004)*, February 2003.

[25] J. Ullrich. Sans institute – blaster, power outage, sobig: Two weeks in august and the internet storm center. http://isc.incidents.org/presentations/sansne2003.pdf, 2003.

[26] J. Viega, J. T. Bloch, T. Kohno, and G. McGraw. ITS4: A static vulnerability scanner for C and C++ code. *ACM Transactions on Information and System Security*, 5(2), 2002.

[27] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, pages 3–17, San Diego, CA, February 2000.

[28] D. Wheeler. More than a gigabuck: Estimating gnu/linux's size. http://www.dwheeler.com/sloc, 2001.

[29] Y. Xie, A. Chou, and D. Engler. Archer: using symbolic, path-sensitive analysis to detect memory access errors. In *Proceedings of the 10th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 327–336. ACM Press, 2003.

[30] M. Zitser. Securing software: An evaluation of static source code analyzers. Master's thesis, Massachusetts Institute of Technology, 2003.