Secure Programming Group

Technical Report

Using Splint to Detect Buffer Overflow Vulnerabilities in Sendmail

David Friedman

5/2002

Splint's logo is Jefferson's serpentine walls on the grounds of the University of Virginia

> "*The walls are one brick thick, but because of their design are both strong and aesthetic. Like a secure program, secure walls depend on sturdy bricks, solid construction, and elegant and principled design."*
>
> *From the Splint website (http://www.splint.org/faq)*

Abstract

Despite the intense growth of computer science research in recent decades, a simple but significant problem in security remains: buffer overflow vulnerabilities. In this paper we explore using static analysis to mitigate the problem.

Buffer overflow attacks exploit unsafe languages like C which do not check the bounds on array accesses. Evidence shows that buffer overflow vulnerabilities remain a significant problem despite the development of safer languages.

The tool we studied (Splint) attempts to detect buffer overflow vulnerabilities in source code before deployment. However, it does not attempt to formally prove that an implementation matches a specification. Instead we take a more lightweight approach that allows our tool to run as fast as a compiler and still catch many vulnerabilities.

Although the case study did not reveal any buffer overflow vulnerabilities in Sendmail it did discover some important bugs in Splint.

Outline

Chapter 1 and 2 give background on the history of computer security, buffer overflow vulnerabilities, formal methods, static analysis, and Splint.

Chapter 3 Describes basic Splint operation.

Chapter 4 Describes the results of the case study.

Chapter 5 Offers some suggestions for further research

Table of Contents

1 Introduction 1
1.1 Context of the Buffer Overflow Problem 1
1.2 The Buffer Overflow Problem 2
1.3 Rationale and Scope 6
2 Splint and Software Development 7
2.1 Splint's Development 7
2.2 Other Security Auditing Tools using Static Analysis 10
3 Method 11
3.1 Overview 11
3.2 Splint Warnings 11
3.3 Annotations 11
4 Sendmail Analysis 14
4.1 Previous Sendmail Analyses 14
4.2 Chronological Record of Analysis 14
4.3 Bugs 15
5 Further Research 18
5.1 Summary 18
5.2 Further Research 18
6 Appendix A: Method Details 22
6.1 Sendmail Source Code 22
6.2 Directory Structure 23
7 Appendix B: Analysis Log 25

Figures and Tables

Figures

Figure 1: Design Space of Checkers (from David Evans's website) 9
Figure 2: Sendmail Analysis Directory Structure.. 23

Tables

Table 1: Buffer Overflow Shopping List Analogy 3
Table 2: Record of Analysis 14

1 Introduction

This report contributes toward the goal of eliminating buffer overflow vulnerabilities. In the 1990's with the explosive growth of the Internet and the emergence of E-commerce, computer security became more important and researchers began to focus on methods to prevent intruders from breaking into systems.

Malicious crackers often exploit a weakness in software called a buffer overflow vulnerability. A buffer overflow vulnerability is exploited by sending the computer more input than it expects. If the software does not correctly check the bounds of the input attackers can get the program to execute their code. Section 1.2 describes buffer overflow vulnerabilities in more detail.

This report focuses on a method to detect vulnerabilities using Splint, a software tool developed at the University of Virginia. The report describes a case study investigating the performance of the tool on Sendmail, a program that handles the details of the e-mail protocol.

1.1 Context of the Buffer Overflow Problem

During the 1940's and the 1950's no scientist researched computer security. Governments, large corporations, and major universities owned all the large mainframe machines and system administrators did not need to worry about somebody tampering with a machine behind a locked door.

In 1969 UCLA established a network link with Stanford. The first attempted message *logon* failed to reach its destination, but networking changed computing forever [Gromov 2002]. No longer did an attacker need to get through a locked door to break into a computer.

Few researchers predicted the growth of the Internet, and they designed protocols with little consideration for security. Researchers made decisions in the 1970's and 1980's which cause security vulnerabilities today.

Language designers did not consider security a high priority, either. Brian W. Kernighan, Dennis M. Ritchie, and other researchers at Bell Labs developed the C programming language in the early 1970's for development of the Unix operating system. Their design emphasized portability, flexibility, and efficiency, but not safety.

Today programmers write software to control everything from medical devices to web servers in C. Linus Torvalds (and others) wrote the Linux kernel in C. Programmers at Microsoft wrote parts of the Internet Information Server and Windows XP in C. Programmers use C because it gives them the freedom to write fast code. However, that freedom makes it easier for programmers to "shoot themselves in the foot".

1.2 The Buffer Overflow Problem

Thou shalt check the array bounds of all strings (indeed, all arrays), for surely where thou typest "foo" someone someday shall type "supercalifragilisticexpialidocious".

—Fifth Commandment of the *Ten Commandments for C programmers*

C makes it easy for programmers to write code with buffer overflow vulnerabilities. For example, in C a programmer can declare an array large enough to hold five characters, but because C does not perform bounds checking the user entering ten characters will overwrite the next sequential locations in memory. By choosing those additional characters carefully, attackers can coax a program into executing their code.

To understand buffer overflow vulnerabilities an analogy may help [ZDNet 2000]. Suppose somebody could fill out their tax form in a special way. Instead of just filling out the fields and sending it to the IRS they could write past the fields, and change the text on the form itself. They could change the equations on the form computing how much they owe.

Another analogy may help clarify the details. Suppose a program asks the user what to buy at the store. Its memory looks like:

Memory Location	Contents
1	`Ask user what he wants a store.`
2	`Put user input into memory starting at location 3 and then jump to the instruction at memory location 105`
3	`<allocated for user input> "0"`
4	`<allocated for user input> "0"`
...	...
104	`<allocated for user input> "0"`
105	`Go to store and buy the item at memory location 3. Keep buying the next sequential item in memory if it is not "0".`
106	`Return items to user.`

Table 1: Buffer Overflow Shopping List Analogy

Normal users would enter something like: `"bread", "milk", "cheese",` and the program will work correctly. But a malicious user could enter: `"spam", "spam", "spam", "spam", "spam", ... "spam",` (102 times), and then `"Give user my wallet".` The instruction `"Give user my wallet"` overwrites the old instruction at memory location 105 and the computer will execute it instead of the original instruction. The program should have checked that the user did not enter more than 102 items.

1.2.1 Seriousness of the Buffer Overflow Problem

Buffer overflow vulnerabilities jeopardize computer systems. Experts from academia, government, and industry have identified the importance of the problem. Gary McGraw of Cigital referred to buffer overflow vulnerabilities as the single biggest threat to software security and Crispin Cowan of Oregon Graduate Institute for Technology called them the "vulnerability of the decade" [McGraw 2000, Cowan 2000].

Statistics show the prevalence of the problem. Both Computer Emergency Response Team (CERT) advisories from 2002 at the time of this report and 18 of the 37 advisories from 2001 involve buffer overflow vulnerabilities [CERT 2002]. David Wagner and others analyzed three different vulnerability databases and found that buffer overflows accounted for 23%, 27%, and 29% of entries [Wagner 2000]. Evans found that buffer overflow vulnerabilites account for 19% of entires in Mitre's Common Vulnerabilities and Exposure List [Evans 2002].

Recent incidents have demonstrated the severity of the problem. In December of 2001 researchers at eEye digital security found a buffer overflow vulnerability in Microsoft's new operating system Windows XP. In late January 2002 researchers found a buffer overflow vulnerability in the popular instant messenger software American Online ICQ [CERT 2002]. The Code Red worm, which CNN reported caused $2 billion worth of damage, exploited a buffer overflow vulnerability in Microsoft's Internet Information Server [CNN 2001].

Buffer overflow vulnerabilities are a serious problem worthy of attention and resources.

1.2.2 Related Work

Researchers have reacted to the severity of the problem by taking different approaches. Some have looked at run-time solutions which work by inserting checks into the executable machine code. Because none of the source code must be modified, these solutions make the programmer's work easier. They have the disadvantage that a performance penalty negates somewhat the efficiency of using C [Larochelle 2001].

Our approach will use static analysis which works by examining lines of source code. Static analysis has the advantage of detecting errors before deployment, but the disadvantage of making the programmer do more work.

Some buffer overflow vulnerabilities may be found through safer software engineering practices including code inspections. Nevertheless, humans will always miss some subtle errors. A computer user found a buffer overflow vulnerability in a program even after the code had been specifically inspected for security errors [Bug 1998].

By using a safer language (e.g. Java or ML) programmers could avoid writing buffer overflows. However, some applications have strict performance requirements and developers must use a low-level language like C. Some programmers must also maintain C legacy code, so eliminating C can never be an encompassing solution [Cowan 2000].

1.3 Rationale and Scope

As discussed in section 1.2.1 (Seriousness of the Buffer Overflow Problem) buffer overflow vulnerabilities pose a real hazard to computers and the humans who use them.

No individuals other than the people who wrote Splint have performed a case study of the tool. This work provides insight into the research direction Splint will go in and more specifically the valuable improvements the Secure Programming Group can make.

2 Splint and Software Development

In this chapter we trace through the relationship between formal methods, static analysis, software development, and Splint. Examining Splint's history will help to relate to the tool more fully, and not just its operation.

2.1 Splint's Development

The original name of Splint was LCLint. David Evans completed the first implementation in June of 1994 for his Masters Thesis [Evans 1994]. Two of the people Evans worked with, John Guttag and Jim Horning, were investigating the effectiveness of a formal specification language (Larch C Language, LCL). The word LCLint comes from concatenating "LCL" and "lint." The "lint" parts comes from the program S.C. Johnson developed in 1978 named for the undesirable "bits of fluff" it helped to remove [Johnson 1978].

2.1.1 Formal Methods

Civilized debate in the computer science community has the capacity to become bickering. Issues like the best operating system (Windows vs. Unix), the best text editor (Emacs vs. VI), the use of `goto,` and the effectiveness of object-oriented programming, all have gone at times beyond the point of scholarly discussion.

Formal methods have occupied this same territory [Young 1991]. Tony Hoare, the inventor of quicksort and others advocated that everything about a program can be deduced statically [Hoare 1969]. In theory an analyst can verify formally that a piece of software does what it should do only by looking at the source code. A formal methods approach includes writing a mathematically precise specification and then running a checker which attempts to prove the source code satisfies the specification. Advocates of formal methods claim it decreases software errors and helps computer scientists reason more effectively about programs. Critics of formal methods point out that it takes substantial effort to write both the tool and the specification while both could have errors.

Guttag, Horning, and Evans conceived of LCLint as a tool for checking the consistency of C programs and LCL specifications.

2.1.2 Source Code Annotations

The conflict over formal methods centers on whether they actually apply to real software or just to academic toy programs.

Whatever the future may bring, today most software companies, if they use formal methods at all, use them only for small modules [Jalote 1991].

Source code annotations offer a compromise. Although most industry developers did not know, and were not willing to learn a formal specification language like LCL, programmers might be willing to learn something simpler: source code annotations. Programmers could add annotations to their code that Splint could use to understand their intentions [Evans 2000].[1]

[1] Section 2.1.3 describes the name change.

Source code annotations offered a "lightweight" approach in comparison to formal methods because they required less computing resources, and more importantly less human resources.

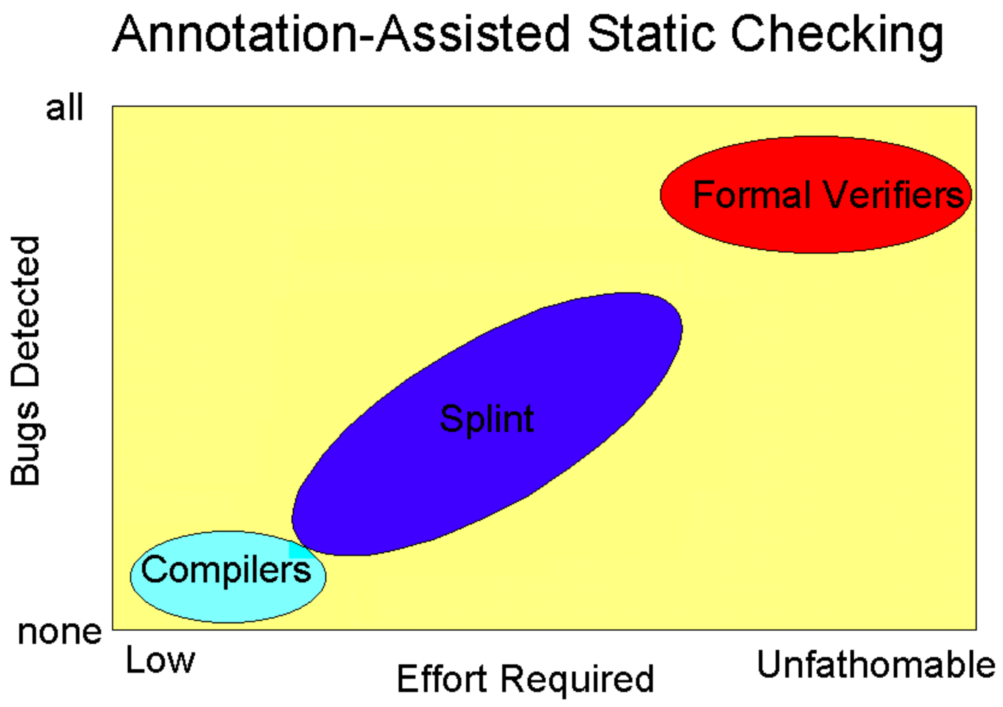

Figure 1: Design Space of Checkers (from David Evans's website)

The graph demonstrates how different tools require differing levels of effort.

Developers of a medical application would choose to use a formal verifier if the program must work correctly or people will die. On the other hand, video game developers would only use a compiler.

2.1.3 Splint and Memory Bounds Checking

Larochelle and Evans added memory bounds checking and extensible checks to Splint in 2001 [Larochelle 2001, Larochelle 2002a]. In January of 2002, Evans renamed the software Splint to reflect the improvements. The name has three different interpretations: Secure Programming Lint, to reflect the buffer overflow checking, Specifications Lint, to reflect the programmer added annotations, and the literal meaning of Splint as "first aid for programmers" [Evans 2002a].

2.2 Other Security Auditing Tools using Static Analysis

Several tools are available that use static analysis techniques to find security vulnerabilities. Flawfinder, RATS, and ITS4 are source code scanners that perform less sophisticated checking but take less effort to learn and run faster [LinuxJournal 2002]. Hence, they are nearer to compilers on the graph of the previous page.

3 Method

3.1 Overview

Checking source code works in an iterative fashion: run Splint, compare Splint's output with the source code, take appropriate action, and repeat. Keep checking while Splint still produces warnings. This chapter describes the process in more detail.

3.2 Splint Warnings

A typical buffer overflow warning looks like:

```
headers.c:1881:2: Possible out-of-bounds store:
    *tail   Unable to resolve constraint:
    requires: : maxSet((&ret @ headers.c:1871:24) ) >= (0)
     needed to satisfy precondition:
    requires: : maxSet((tail @ headers.c:1881:3) ) >= (0)
```

Splint raised a warning about the 1881st line in the file headers.c. An analyst would attempt to figure out why Splint raised the warning by examing the code in the file at that location. After determining the problem he would determine the appropriate action to take: add an annotation, add a control comment, or modify the code.

3.3 Annotations

Splint's buffer overflow annotations fall into two forms: `requires` and `ensures` clauses. `Ensures` clauses state the function postconditions:

```
/* this function allocates a 24 byte buffer
   which is the size of an IP packet header */
void *alloc_iph()
/*@ensures maxSet(result) == 23@*/
{
  return malloc(24);
}
```

This `ensures` annotation tells Splint that the return value of this function is a pointer to a 24 byte buffer (in C arrays are zero indexed).

The `malloc` call returns the size requested in bytes or a NULL pointer if the request cannot be fulfilled.

`Requires` clauses state the function preconditions:

```
void fill_with_digits(char *ptr)
/*@requires maxSet(ptr) >= 9@*/
{
  int i;
  for(i = 0; i < 10; i++)
    ptr[i] = '0' + i;
}
```

This annotation states that the buffer pointer to by parameter `ptr` must have at least enough space to store ten bytes.

Annotations can both suppress warnings, and generate them. In the case of an `ensures` annotation Splint will generate a warning if it cannot verify that the function actually `ensures` the post-condition. An `ensures` annotation can also suppress a warning since it tells Splint that a condition will hold after the call. A `requires` annotation will generate a warning if Splint cannot verify that the condition holds before the call.

3.3.1 Adding a Control Comment

When there is no appropriate annotation to add the analyst can suppress the warning with a control comment:

```
/* dkf: don't think this is an error because controladdr is
allocated correctly (statically) */
/*@-boundswrite@*/
memset(&controladdr, '\0', sizeof controladdr);
/*@=boundswrite@*/
```

In this example we suppress checking for the `memset` call. Note that we also add a human comment describing the rationale.

This option is not as desirable as adding an annotation because it provides less information to Splint. Also the analyst could make a mistake. Nevertheless, control comments help to manage the complexity of checking thousands of lines of code.

3.3.2 Modify the Code

Analysts should not change the code lightly when checking well established code like Sendmail. If they do think there is a mistake they should verify the change with a colleague or the author. Analysts checking their own code or a project in the early development stages may change the code more freely.

For the following function:

```
char *strip(char *ptr)
{
  while(*ptr = ' ') ptr++;
  return ptr;
}
```

Splint reports a warning for the `while` loop because it uses an assignment as the test expression. In this case the programmer made the common mistake of confusing `=` and `==`, and should change the code.

4 Sendmail Analysis

4.1 Previous Sendmail Analyses

David Wagner found buffer overflow vulnerabilities in Sendmail version 8.7.5 using a static analysis tool. However, he did not find any exploitable vulnerabilities in the current version (8.9.3) at that time [Wagner 2000]. In 1995, 1996, and 1997 various people found vulnerabilities in the current version at that time [CERT]. We choose arbitrarily to analyze version 8.11.4.

4.2 Chronological Record of Analysis

The following table shows how the number of warnings decreased throughout the analysis as we added more annotations:

Date	Warnings	Annotations[2]
0 Tue 3/12	439	0
1 Tue 3/12	438	1
2 Tue 3/12	436	3
3 Thu 3/14	238	9
4 Thu 3/14	206	9[3]
5 Fri 3/15	160	28

Table 2: Record of Analysis

The analysis did not find any bugs in Sendmail. However, we did find a few significant bugs in Splint. We describe these bugs in more detail in section 4.3.

[2] These numbers were not obtained in a strict fashion, but are still representative of how the analysis went. Also see the log in appendix B.

[3] Modified the standard library annotation for memset

4.2.1 Analysis Process

Splint took only 1 minute to check all 70,000 lines of code on a 1200 MHz machine with 1 GB of RAM. The initial steps succeeded in addressing many spurious warnings by adding the appropriate control comments. For example, Splint reported a warning for every occurrence of the `newstr` macro:

```
#define newstr(s)\
     strcpy(malloc(strlen(s) + 1), s)
```

Splint could not determine that the calls to `newstr` cannot overflow a buffer since `malloc` allocates exactly the necessary space to hold `s`. Since this macro occurred in many places throughout the source code adding the correct control comment succeeded in addressing about 200 of the warnings.

We achieved the next significant decrease by removing the requires annotation from the `memset` prototype. Since `memset` copies a constant into a buffer an attacker cannot exploit it to inject his own code.

The final winnowing in warnings was achieved by adding annotations and by putting Splint in a different analysis mode. That mode is not fully debugged so Splint may have suppressed some non-spurious warnings.

The annotations usually worked as expected. When they did not we made a note of the problem so somebody could later fix it. The next section describes the bugs we found in Splint during the analysis.

4.3 Bugs

This section describes the most important bugs we found in Splint. The most serious problem is the erroneous loop checking.

4.3.1 Segmentation Violation

When Splint tried to parse the Sendmail source code it generated a segmentation violation for the file `stats.c`. After discussing it with a colleague we decided to make a note of the problem and continue the analysis without that file. Since it only contains 174 out of the 70,000 lines of code Splint could analyze the rest of the source without it.

4.3.2 Loop Heuristics

Splint has trouble analyzing loops correctly. It raises a spurious warning for the following piece of code:

```
void test()
{
  int i;
  double arr[10];
  for (i = 0; i < 10; i++)
    arr[i] = 0.0;

}
```

Splint should recognize the common `for(i=start; i < end; i++)` idiom.

More importantly Splint fails to find buffer overflow vulnerabilities in code with loops. Splint fails to generate a warning for this write beyond the end of the buffer:

```
int main()
{
  char* char_array[40];
  int i = 0;
  char* p = malloc(1);
  *p = 'c';

  for(i = 0; i < (int) sizeof(char_array); i++)
      char_array[i] = p;
 return 0;
}
```

Here the programmer wrote: `i < sizeof (char_array)` when he meant: `i < (sizeof (char_array)) / (sizeof (char_array[0]))`

`Sizeof (char_array)` is 160 bytes yet in each iteration the program is not going through the individual bytes but the pointers themselves, so the upper limit should be 40.

This problem appeared when we attempted to analyze a program with a known buffer overflow vulnerability. Splint could not find the vulnerability because it could not check loops correctly [Security 2001, Bugzilla 2001].

5 Further Research

5.1 Summary

In many cases Splint worked correctly. However, it still has some significant bugs in the loop heuristics. We are aware of these problems and will try to fix them.

5.2 Further Research

The following are possible ways to continue research:

- **Check to see if Splint can find known vulnerabilities** -- Find a known vulnerability on one of the full disclosure lists like BugTraq and see if Splint can find it (similar to the work described in section 4.3.2.).
- **Analyze a simpler program.**
- **Analyze Splint's source code using Splint.**
- **Fix some of the bugs.**

References

[Bug 1998] Chris Evans. Nasty Security Hole in `lprm`. Bugtraq mailing list. April 18, 1998. `http://www.securityfocus.com/`

[Bugzilla 2001] *Man 1.5h1-10 has an exploitable overflow.* May 13, 2001. Bugzilla Bug 40400. `https://bugzilla.redhat.com/bugzilla/`

[CERT 2002] See `http://www.cert.org/advisories`

[CERT] See `http://www.cert.org` and search for "Sendmail".

[CNN 2001] *Cost of 'Code Red' rising*. August 8, 2001 `http://www.cnn.com/2001/TECH/internet/08/08/code.red.II/`

[Cowan 2000] Crispin Cowan, Perry Waggle, Calton Pu, Steve Beattie, and Jonathan Walpole. *Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade.* DARPA Information Survivability Conference and Exposition. January 2000.

[Dictionary] Dictionary.com `http://dictionary.com`

[Evans 1994] David Evans. *Using Specifications to Check Source Cod*e, MIT/LCS/TR-628, MIT Laboratory for Computer Science, June 1994.

[Evans 2000] David Evans. *Annotation Assisted Lightweight Static Checking*. The First International Workshop on Automated Program Analysis, Testing and Verification (ICSE 2000). Feb 25, 2000.

[Evans 2002] David Evans and David Larochelle. *Improving Security Using Extensible Lightweight Static Analysis.* IEEE Software. Jan/Feeb 2002.

[Evans 2002a] David Evans. *Splint FAQ*. Available at: `http://www.splint.org/faq.html`

[Gromov 2002] Gregory G. Gromov. *The Roads and Crossroads of Internet History.* `http://www.netvalley.com/intval.html`

[Hoare 1969] C. A. R. Hoare. *An axiomatic basis for computer programming*. Communications of the ACM, 12(3):335-355, 1969

[Jalote 1991] Pankaj Jalote. *An Integrated Approach to Software* Engineering. New York: Springer-Verlag New York, Inc., 1991

[Jargon] The Jargon File. `http://www.tuxedo.org/~esr/jargon/`

[Johnson 1978] S.C Johnson (1978). *Lint, a C Program Checker* Unix Programmer's Manual, AT&T Bell Laboratories: Murray Hill, NJ.

[Larochelle 2001] David Larochelle and David Evans. *Statically Detecting Likely Buffer Overflow Vulnerabilities.* In Proceedings of the 2001 USENIX Security Symposium, Washington D.C., August 13-17, 2001.

[Larochelle 2002a] David Larochelle and David Evans. *Improving Security Using Extensible Lightweight Static Analysis* 2001. IEEE Software, Jan/Feb 2002.

[Larochelle 2002b] David Larochelle and David Evans. *Splint Users Manual*. Available at: `http://www.splint.org/manual`

[LinuxJournal 2002] Linux Journal. *Source Code Scanners for Better Code.* January 26, 2002.

[McGraw 2000] Gary McGraw, John Viega. *Get reacquainted with the single biggest threat to software security.* March, 2000 `http://www-106.ibm.com/developerworks/library/overflows/`

[NASA 1999] NASA. Basic COCOMO Web Calculator available at: `http://www.jsc.nasa.gov/bu2/COCOMO.html`

[Paulson 2002] Linda Dailey Paulson. *Wanted: More Network-Security Graduates and Research.* IEEE Computer. February, 2002. Available at: `http://www.cs.virginia.edu/misc/wulfnews.pdf`

[Security 2001] Security Focus Vulnerability Database. June 12, 2001. *Linux Man Page Source Buffer Overflow Vulnerability*. BugTraq ID: 2872. `http://online.securityfocus.com/`

[Splint] The Splint website. `http://www.splint.org`

[Wagner 2000] David Wagner, Jeffrey S. Foster, Eric A. Brewer and Alexander Aiken. *A First Step Towards Automated Detection of Buffer Overflow Vulnerabilities.* Network and Distributed Systems Security Symposium. February 2000.

[Webopedia] Webopedia. `http://webopedia.internet.com/`

[Young 1991] William D. Young. *Formal Methods versus Software Engineering: Is there a Conflict?* Proceedings of the Symposium on Software Testing and Analysis. October 1991.

[ZiffDavis 2000] ZDNet. *Tech View: How 'buffer overflow' attacks work*. July 20, 2000 `http://techupdate.zdnet.com/techupdate/stories/main/0,14179,26505669,00.html`

6 Appendix A: Implementation Details

This appendix describes the procedure we used to analyze Sendmail. This method includes a scheme to keep track of the various stages of the analysis.

We worked in a Unix (specifically RedHat Linux 7.2) environment. The Sendmail consortium does not produce a Windows version of Sendmail.

There exist many ways to use Splint to analyze source code, and programmers may want to adapt this method for their task or may decide on a different structure. A single person conducted this analysis relying only on simple scripts and Unix utilities. A team of developers may want to use a more sophisticated version control system that supports multiple analysts working simultaneously (CVS for example).

6.1 Sendmail Source Code

First we downloaded and examined the Sendmail source code (available at http://www.sendmail.org). For RedHat Linux 7.2 the Sendmail source consists of eighty files totaling about seventy-thousand lines of code. Because the project faced limited time and resources, it was not possible to analyze every line of code. Nevertheless, knowing our configuration could prove helpful to someone seeking to perform a similar analysis.

6.2 Directory Structure

Here is the directory structure:

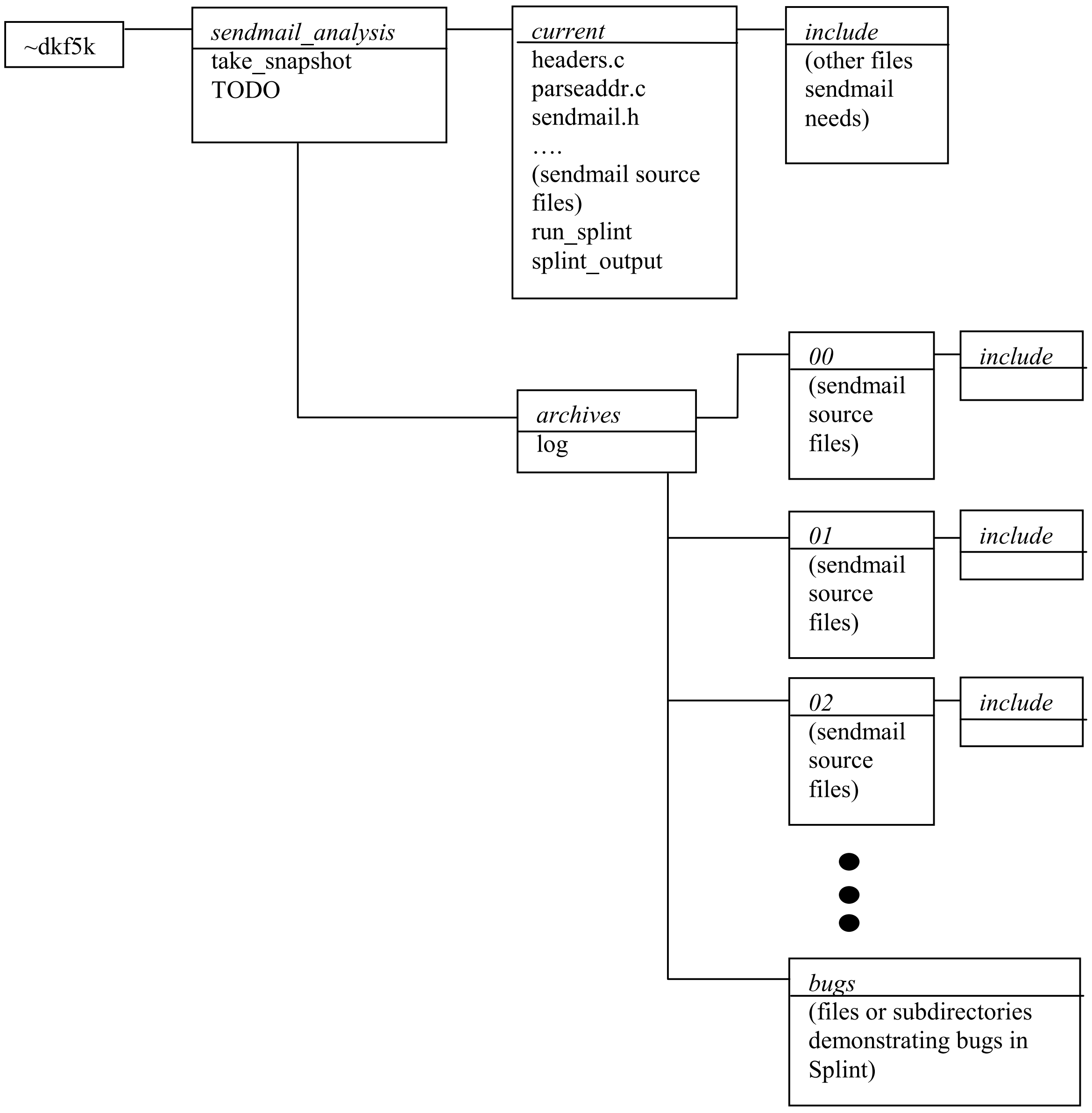

Figure 2: Sendmail Analysis Directory Structure

The analyst spends the bulk of his time working in the `current` directory. He runs Splint (using the run_splint Perl script) and compares Splint output with the code.

The other branch of the directory tree is the `archives`. Here is where we store periodic "snapshots" of the current analysis state numbered sequentially starting from `00`. The `take_snapshot` script automates the snapshot process. When the analyst wants to take a snapshot he runs the program which copies the `current` directory and all of its subdirectories to the next sequentially numbered archive directory. Subdirectory `00` contains the unannotated code downloaded from the Sendmail web site. The program then prompts the user to enter a log entry which has the following form:

```
04:Thu Mar 14 14:03:27 EST 2002
Finished checking --- 206 code warnings
Made a change to the
annotation in the standard library
function memset.
```

The `run_splint` script automatically generates the subdirectory number, date, and number of warnings Splint generated for the snapshot in that archive. The analyst then enters a brief log entry.

The other subdirectory `bugs` contains code demonstrating bugs in Splint.

7 Appendix B: Analysis Log

```
00:Tue Mar 12 18:33:11 EST 2002
Finished checking --- 439 warnings.

Started the analysis. Inside this
directory is the original sendmail
source code.

01:Tue Mar 12 19:43:54 EST 2002
Finished checking --- 438 warnings.

Ignored the first warning. Can't
be a buffer overflow vulnerability
since it strcpys a string literal.

02:Tue Mar 12 21:10:41 EST 2002
Finished checking --- 436 code warnings

All of the warnings in alias.c
involved the same issue where
there was a macro named
newstr:

#define newstr(s) strcpy(xalloc(strlen(s) + 1), s)

which splint thought could be
a  buffer overflow
vulnerability because of the
call to strcpy, but is in fact spurious


03:Thu Mar 14 00:14:37 EST 2002
Finished checking --- 238 code warnings

Great. With a very simple annotation
to the

#define newstr(s)\
      /*@-boundswrite@*/ strcpy(xalloc(strlen(s) + 1), s)
/*@=boundswrite@*/

macro we got rid of a whole lot of warnings.
Also, from
now on any annotations or code added
will be preceeded by /* dkf .... */
so somebody can grep for all the stuff
I add.



04:Thu Mar 14 14:03:27 EST 2002
Finished checking --- 206 code warnings

Made a change to the
annotation in the standard library
function memset:
```

```
void /*@alt void *@*/ memset
(/*@out@*/ /*@returned@*/ void *s, int c, size_t n)
/*@modifies *s@*/
/*requires maxSet(s) >= (n - 1)*/
/*@ensures maxRead(s) >= (n - 1) @*/ ;
```

Got rid of the requires clause.

This is okay since memset
is copying a constant
byte into a buffer.

This step
got rid of two dozen or so warnings
related to memset.

Note: change made to the standard
library saved in the subdirectory lib/
within the source directory and appropriate
change made to the Makefile.

```
05:Fri Mar 15 20:35:09 EST 2002
Finished checking --- 160 code warnings
```

Using the implictconstraint flag eliminated
40 or 50 errors.

Working with David Larochelle we annotated
the parseaddr.c file till Splint no longer
produced any errors for that file.