An Inquiry into the Stability and Reliability of UNIX Utilities

Brian L. Bowers
blbowers@cs.wisc.edu

Karlen Lie
soph@cs.wisc.edu

Gregory J. Smethells
smethegj@cs.wisc.edu

University of Wisconsin – Madison
Computer Sciences Department
1210 West Dayton Street
Madison, WI 53706 USA

Abstract

We tested a large set of UNIX utilities under two popular UNIX variants, a GNU/Linux platform and a Solaris platform, for characteristics of stability and reliability. The testing methodology we employed was simple, yet effective – we subjected the utilities to input streams of random characters. Four percent to eleven percent of the utilities we tested failed, either by crashing or by looping infinitely, and were subjected to scrutiny and debugging to determine the cause.

1 Introduction

Our goals were two−fold: (1) to produce an updated set of tools and (2) to compare the reliability of current UNIX utilities to the reliability found by earlier studies [1,2]. We chose GNU/Linux and Solaris as our test platforms because they are representative of UNIX variants commonly used on workstation available today. The tests were performed on machines running installations of Red Hat 6.2 and SunOS 5.7. After making minor substitutions with equivalents for a few utilities, we went on to test essentially the same set of UNIX utilities as has been previously examined by fuzz reports [1,2]. Our study's results displayed improvement for utilities on both platforms, but showed that the highest reliability continues to be found in systems running a GNU/Linux distribution.

To examine the reliability of these utilities, we used a program named *fuzz* that is able to produce a stream of pseudo−random bytes on its stdout. These bytes are then given as input to other utilities to determine their reliability and stability. No *crashes* or *infinite loops* were considered to be valid reactions to the input stream.

We took our definitions of a *crash* and an *infinite loop* from the original fuzz paper [1]. Any utility that we caused to *seg−fault* and produce a *core dump* using only the random input from fuzz was labeled as an application that crashed. Also, any utility that we caused to *hang* or produce output for a significant amount of time after the input was exhausted was labeled as an *infinite loop*. Generally, we allowed at least five minutes to pass before considering it to be a *significant* amount of time. Finally, any utility that *hung* waiting with input available was also labeled as an *infinite loop*. Utilities that were suspended through the operating system (by control−z) were not considered in this final case.

Initially, we assumed that we would see a similar proportion of crashes and hangs as in previous studies [1,2]. We also assumed that previously reported "defects" would be removed. We were surprised to see approximately 50% improvement on both platforms; we were equally surprised when we were able to discover bugs in certain utilities that existed in one form or another since 1990.

This paper is organized in the following manner: Section 2 describes the testing methods we have used during this research. Section 3 discusses how we updated the *fuzz* and *ptyjig* tools provided by the earlier studies. Section 4 comments on the results of running these tools on our GNU/Linux and Solaris workstations. Section 5 offers some discussion surrounding related works. Section 6 provides our conclusions about the results of this study. Lastly, Section 7 describes what further directions we feel this research could follow.

2 Methodology

The approach we employed is the same as that used in the original fuzz paper. We tried to repeat their experiments with as little change as possible to allow valid comparisons between all sets of results.

We used random streams of bytes generated by the program *fuzz* as input to the various utilities, either via *stdin* or a pseudo−terminal adaptor program, *ptyjig*. Each utility tested is expected to quit, with or without a verbose error message, if the input does not meet its criterion for valid input. Any crashes or infinite loops, as defined above, were not considered acceptable. Because the input streams we used cover a varied set of random characters, they seem to be fairly adept at exposing latent bugs, even though the approach does not adhere to any particular strict testing methodology. In the fuzz project's view [1], any method that can find a bug in a repeatable manner is a good method. Although the approach is crude, what it lacks in complexity is compensated by its speed and ease−of−use.

Non−interactive utilities were tested using several different types of random character input streams as described in Table I on the following page. For interactive utilities the streams described in Table I were split into lines of random length, having a mean of 128 characters to avoid "overflowing the input buffers on the terminal device" [1]. This method parallels that of the original fuzz paper and should allow valid comparisons between all results involved.

File #	Character Types	NULL char	Input stream size (in bytes)
1	printable + nonprintable	Yes	1×10^3
2	printable + nonprintable	Yes	1×10^4
3	printable + nonprintable	Yes	1×10^5
4	Printable	Yes	1×10^3
5	Printable	Yes	1×10^4
6	printable	Yes	1×10^5
7	printable + nonprintable		1×10^3
8	printable + nonprintable		1×10^4
9	printable + nonprintable		1×10^5
10	printable		1×10^3
11	printable		1×10^4
12	printable		1×10^5

Table I*: Various types of random input char stream types used during testing*
This table originally appeared in the first fuzz paper [1].

3 Software Update

One of our goals was to provide an updated version of the original *fuzz* and *ptyjig* programs used to do the random input testing. These programs were written during the original fuzz project [1]. One of the reasons for the update was in the interest of keeping the differences between the tests done by our project and those that preceded us to a minimum so that we could make valid comparisons between results. Also, we noted that it had been several years since the code had been updated. We assumed an update might be necessary to even allow some of the code to compile using current compilers and libraries. Lastly, to be of service to the community at large, we have been given permission to release these utilities under the GNU General Public License (GPL).

3.1 Fuzz

We modernized *fuzz* from Kernighan and Ritchie (K&R) C style to modern ANSI C. During these changes, we discovered, ironically, that *fuzz* itself contained a bug as shown below:

```
void myputs( char *s )
{
  int      c;

  while( s != 0 ) {
     ...
```

A comparison in a while−loop, attempting to find the end of an input string, was comparing a pointer to NULL (zero). The pointer never took on the value NULL; however, what it pointed *to* did. The fix was to simply look for the NULL−character which terminates C−style strings:

```
void myputs( char *s )
{
  int      c;

  while (*s != 0) {
     ...
```

The bug was in rarely used code. The code is only used when an "epilog" to the sequence of characters produced by *fuzz* is specified – such as "<esc>:q" to cause *vi* to quit properly. Since the scripts used for testing by the original project did not utilize this feature the bug remained hidden until now.

3.2 Ptyjig

The fixes to *fuzz* were extremely minimal compared to *ptyjig*, partly due to the system dependent nature of low−level pseudo−terminal code, which *ptyjig* uses extensively, and partly due to the evolution of standards in the C programming language.

Initial changes to the *ptyjig* code base were related to the differences in pseudo−terminal implementations. One of our major testing platforms was Solaris, which continues to have a BSD−style pseudo−terminal interface. Although *ptyjig* was originally written for BSD−style pseudo−terminals, the standards had changed. Header files had been moved, renamed, or simply eliminated. We added numerous **#include** lines when we could. In some cases we simply filled in missing MACRO constants with our own **#define** lines. Our other major platform was Red Hat Linux and that required a significant amount of porting of the *ptyjig* code to the current POSIX standard for pseudo−terminals used by that system. Much of this port was inspired by code found in The Linux Kernel Book by Card, Dumas, and Mevel [3].

Some of the programs we tested needed their own command line arguments. The *getopt* routine, used by *ptyjig* parses the entire command line as arguments to *ptyjig*. For example, we wanted *emacs* to run in the terminal where it was started, without spawning its own X−Window. If we provided emacs with the command line flag "−nw" it would behave just that way. We changed ptyjig so that the command "ptyjig −d 0.05 emacs −nw" would pass the "−nw" flag to emacs.

Also, to allow the code to compile on either Linux or Solaris, code segments related only to one system or the other were encased in guard statements (**#ifdef ... #endif**) to support conditional compiling. These modifications allowed the code, with proper **#define** statements set for the current OS, to compile and run successfully.

4 Discussion

Under the guidance of Dr. Barton Miller, we examined the set of utilities to be tested. Specifically, we looked for software that was unmaintained. We replaced the *ditroff* package of utilities in our test bed with the more modern *groff* package and replaced the venerable *vi* with *vim*. Despite our goal of reproducing the original fuzz tests with as little change as possible, we felt it would be a foolish consistency to use software that had no chance of being updated and that was being phased out of active use. Testing the current software gave us a better view of current reliability.

For completeness we did perform initial tests on several utilities in the *ditroff* package, such as *ditroff, nroff, eqn,* and *refer*, all of which crashed under Linux and Solaris. Their counter−parts in the *groff* package did not crash at all when given the same input files of random characters. Table II is provided to show the commonly used name along with the specific name used during testing on given platforms.

Common Name	SunOS 3.2, 4.0, 4.1.3	SunOS 5.7	Slackware 2.0.1	Red Hat 6.2
dbx	dbx	dbx	gdb	gdb
csh	csh	csh	csh	tcsh
eqn	eqn	geqn	−−	geqn
groff	ditroff	groff	ditroff	groff
lex	lex	lex	flex	flex
nroff	nroff	gnroff	nroff	gnroff
refer	refer	grefer	refer	grefer
vi	ex	vim	−−	vim

Table II: *Names used in Table III under the column "Utility" that had equivalent versions under a different name, which we use during our testing.*

A similar table appears in the second fuzz paper [2].

Our initial test runs of *vim* took longer than expected. Our input files contained character sequences that put the program into a suspended state. *vim* did not "fail" our tests; the tests simply did not complete. Fortunately, *vim* is capable of being flagged as "−Z" or run as *rvim*, a restricted instantiation of *vim* that does not allow suspension of the process. Since our definition does not allow suspended states to be considered infinite loops,

testing in the above manner allowed us to fairly apply our criterion for failure or success to *vim* as well (use of "−Z" was not necessary on Red Hat 6.2).

We observed few crashes and infinite loops in the utilities under GNU/Linux. Only *emacs* became caught in an infinite loop and only *ptx* and *tcsh* crashed. Solaris utilities exhibited more crashes and hangs than Linux. Among those that crashed were *adb*, *col*, *dc*, *plot*, *ul*, and *units*. For infinite loops, there were two: *dbx* and *look*. In comparison to the 1990 and 1995 studies, these results are an improvement.

In the tradition of fuzz reports, we shall give examples of each category of failure type taken from the source code of utilities we tested.

4.1 Arrays and Pointers

Easily the most prominent cause of failures, pointers generally caused crashes of utilities because of a lack in the conditional tests of an enclosing control−loop. A prime example is found in *tcsh*:

file sh.lex.c: line 669

```
...
while ((c = getC(0)) != (−1)) {
    *np++ = (char)c;
    if(c == delim) delimcnt−−;
    if(!delimcnt) break;
}

...
```

No attempt is made to check for the end of the array pointed to by the variable *np* inside the loop condition. The variable *np* points to a fixed length array containing only 121 positions. Various ASCII input strings of proper length with minimal delimiters can put *np* beyond the array's bounds.

A simple solution would be to compare the position of *np* to the end of the corresponding array and produce an error if the input string does not fit a length criterion. Errors of this variety occur within the code of novices and experts alike. There exists an well−known association between an array and its length that cries for an object−oriented approach to glue these pieces into one and allow only certain methods of access. Certainly issues of efficiency versus reliability come into conflict here. However, in a computing environment growing ever more prone to security hacks, errors involving pointer manipulation should be minimized.

4.2 Not Checking Return Codes

A cause of infinite loops that continues to exist in utilities is code that does not check the return value of a function. The example we present is from *adb*. A bug of this form has existed in *adb* since the original fuzz paper, though the binary we tested was as recent as 1998. The function *rdc* in *adb* makes a call to *readchar* without checking the result returned:

file runpcs.c: line 279

```
...
while (!isspace(lastc)) {
       *p++=lastc;
       (void) readchar();
}
...
```

The variable *lastc* is set by the function *readchar* and contains the value −1 if *readchar* attempts to read past the end of input (EOI). If EOI is reached before a space character is encountered, this while−loop will continue to iterate unabated.

4.3 Dangerous Input Functions

Some utilities still depend upon dangerous input functions for parsing data. Generally, the function uses a pointer parameter to store data, but no parameter related to the length of the array over which it will iterate. Here we show code defining a function name *getstr*, in *plot*, which is similar in nature to *gets*:

file crtdriver.c: line 150:

```
...
getstr(s, fin) char *s; FILE *fin; {
      for ( ; *s = getc(fin); s++)
            if (*s == '\n')
                  break;
      *s = '\0';
}
...
```

An obvious addition of a length parameter would help prevent this function from causing crashes, as would the use of a loop structure that better defines how this function operates, such as a while−loop. The supposed while−loop's condition would check the return from *getc* and exit the loop if a newline was reached or if the length of the array would be exceeded.

Utility	SunOS 3.2/4.0 (1990)	SunOS 4.1.3 (1995)	SunOS 5.7 (2001)	Slackware 2.1.0 (1995)	Red Hat 6.2 (2001)
adb	C	C	C	X	X
as					
awk					
bc				X	
bib		C			
calendar				X	
cat					
cb				X	X
cc					
ccom			X	X	X
checkeq				X	X
checknr				X	X
col	C	H	C		
colcrt			X		
colrm			X		
comm					
compress					
cpp			X		
csh	H				C
ctags	X		X	H	X
dbx	[C]		H	C	X
dc			C	X	
deqn	C	X	X	X	X
deroff	C	C		X	
diction	X	C		X	
diff					
dtbl		X	X	X	X
emacs	C	X		X	H
eqn	C	C		X	
expand					
f77				X	X
fmt					
fold					
ftp	C	C			
graph				X	X
grep					
grn		X		X	
groff	C	C			
head					
ideal		X	X	X	X
indent	C, H	H		C	X
join	<C>				
latex					
lex	C	C		C	
lint				X	X

Table III: C – Crash, [C] – SunOS 3.2 only, <C> – SunOS 4.0 only,H – Hang, X – Not tested

Utility	SunOS 3.2/4.0 (1990)	SunOS 4.1.3 (1995)	SunOS 5.7 (2001)	Slackware 2.1.0 (1995)	Red Hat 6.2 (2001)
look	H		H		
m4					
mail			X		
make					
more					
nm					
nroff		C			
pc			X	X	X
pic		X		X	
plot	H	X	C, H	X	X
pr					
prolog	C, H			X	X
psdit				X	
ptx	C	C	X	X	C
refer	[C]				
rev			X		
sed					
sh					
soelim				X	
sort					
spell	C	C		X	
spline				X	X
split					
strings					
strip					
style	X	C		X	
sum					
tail					
tbl					
tee					
telnet	C	C			
tex					
tr					
troff	X	X		X	
tsort	[C]				
ul	C	C	C	C	
uniq	C	C			
units	C	C	C	X	X
vgrind				X	X
vi				X	
wc					
yacc					
# tested	83	80	76	55	68
# crash/hang	24	18	8	5	3
% failed	29%	23%	11%	9%	4%

Table III: C – Crash, [C] – SunOS 3.2 only, <C> – SunOS 4.0 only,H – Hang, X – Not tested

5 Related Works

Two previous studies [1,2] from The University of Wisconsin − Madison have tested UNIX utilities. Members of the Internet community have found these studies and have added their efforts to increase the reliability of the various utilities.

An article on Slashdot [6] pertaining to the studies done at Madison inspired Ben Woodward to create an alternate version of fuzz, for Linux, and release it under the GPL [4]. His utility runs more automated fuzz tests, on a specified utility, in the spirit of the original fuzz program. Bug reports can be emailed back, if enabled during the first run of the utility, to a central location to aid in collecting information about bugs in utilities in Linux distributions (the version of Woodward's fuzz we tested unfortunately did not compile on Solaris). The source code to this fuzz utility and more information about its use can be found on SourceForge [4].

Another web−site, humorously entitled "The Bulletproof Penguin", is focused on bringing about the fixes of all bugs found in the 1995 fuzz paper and elsewhere [5]. This page is maintained by Scott Maxwell, who has been diligent enough to send in patches to maintainers for all the bug fixes he has found. This, no doubt, has led to the increase in stability, since the 1995 study, of the utilities we tested on our GNU/Linux platform.

6 Conclusions

Fuzz testing treats software as a black−box. Random inputs are fed to the program and the results are observed. If the program crashes or hangs, it has failed the test and has bugs. Unfortunately, it is not possible to say that a program has no bugs, simply because it did not crash and did not hang; the lack of a failure only indicates that the test failed to find any bugs. However, if a bug is found, the method makes repeatability very simple, and hence, tracking down a bug easier. It has made creating reliable software that much easier and, at the very least, UNIX utilities on the GNU/Linux platform have benefited [5].

UNIX utilities have improved since the original Fuzz testing [1]. We were only able to crash or hang 4% to 11% of the current utilities as compared to 9% to 23% in the previous study [2]. Despite the improvement, we still found bugs that were similar to bugs found in the original paper eleven years ago and these bugs occur despite well−known warnings.

The following quote is taken, by way of the original fuzz paper, from the Solaris 2.3 manual page for *gets* and is related to the bug described in Section 4.3:

> *When using gets(), if the length of an input line exceeds the size of s, indeterminate behavior may result. For this reason, it is strongly recommended that gets() be avoided in favor of fgets().*

An updated version of the *gets* manual page, from Mandrake−Linux 7.2, says the following in its BUGS section:

> *Never use gets(). Because it is impossible to tell without knowing the data in advance how many characters gets() will read, and because gets() will continue to store characters past the end of the buffer, it is extremely dangerous to use. It has been used to break computer security. Use fgets() instead.*

The majority of the errors we found were pointer related. In fact, even the failure due to using a *gets* variant is related to pointers. Many of these errors were simply caused by moving pointers beyond an array's fixed sized bounds. These errors can be found in commonly used utilities, such as *tcsh*. Books, including Deitel and Deitel's C++: How to Program, warn about over−running array boundaries [8]:

> *When storing a string of characters in a character array, be sure that the array is large enough to hold the largest string that will be stored. ? If a string is longer than the character array in which it is to be stored, characters beyond the end of the array will overwrite data in memory following the array.*

Pointer errors were the prominent cause of crashes in utilities in each of the fuzz studies [1,2]. From this, we know these types of errors occur because programmers did not heed the warnings and experiences of others. We speculate that making pointer arithmetic available in a language will cause some pointer related errors. However, based on the fact that GNU/Linux utilities attained 4% failure rate we believe that at least this level of reliability is possible and these types of errors are preventable.

7 Looking Ahead

Another logical next step is to systematically test network related daemon software, including BIND, ftpd, fingerd, and sendmail. Events such as the Internet Worm in 1989, were a warning that local software problems could affect many systems. Recent Distributed Denial of Service attacks have raised some awareness of software vulnerabilities in network related packages. Each day more businesses and homes get broadband connections to the Internet. Attempts are made on a daily basis to remotely *crack* systems using stack smashing and buffer overflows. Fuzz generated input streams can be used to find software vulnerable to these types of attack. We, as a community, need to perform benevolent attacks on our own systems, to find and repair defective software.

Acknowledgements

We would like to thank Dr. Barton Miller for his suggestions and guidance during this project. Thanks is also due for the assistance provided by David Parter and the Computer Systems Lab during our examination of the Solaris source code.

Bibliography

[1] Miller, Bart, Lars Fredriksen, and Bryan So. "An Empirical Study of the Reliability of UNIX Utilities", Communications of the ACM 33.12 (1990): 32−44.

[2] Miller, Bart, et. al., "Fuzz Revisited: A Re−examination of the Reliability of UNIX utilities and Services", Madison: U of Wisconsin, 2000.

[3] Card, Remy, Eric Dumas, and Franck Mevel. The Linux Kernel Book. Paris: Editions Eyrolles, 1997. English Ed., John Wiley and Sons, Ltd., 1998.

[4] Woodward, Ben. "Fuzz home page." 29 Nov. 1999. SourceForge.Net. 17 Apr. 2001. <http://fuzz.sourceforge.net>.

[5] Maxwell, Scott. "The BulletProof Penguin." 1 Aug. 2000. Pacbell.Net. 17 Apr. 2001. <http://home.pacbell.net/s−max/scott/bulletproof−penguin.html>.

[6] Hemos. "Linux and GNU at Their Best." 10 Jan. 1999. Slashdot.Org 1 May 2001. <http://slashdot.org/articles/99/01/10/173223.shtml>.

[7] Deitel, H.M. and P.J. Deitel. C++: How To Program. 3rd Ed. New Jersey: Prentice Hall, Inc., 2001.