Chapter 39
Classic Fuzz Testing: Command Line Studies

Revision 3.2, December 2025.

Objectives

- Learn about what was tested in four classic fuzz studies.
- Understand the type and quality of errors that were found in these studies.
- Learn about the programming practices that led to these errors.

39.1 Introduction

In this chapter, we will talk about the early fuzz studies of the reliability of command line applications. As we mentioned in the previous chapter, these studies include the original 1990 study, 1995 extended study, 2006 MacOS study, and the most recent 2020 study of free UNIX versions.

39.2 The 1990 Study

The first study, done in 1988 and published in 1990, tested a total of 88 different command line applications on six versions of UNIX. These versions included the notable BSD (Berkeley Software Distribution), SunOS from Sun Microsystems (now Oracle), SCO UNIX (Santa Cruz Operation), IBM's AIX, and Sequent's Dynix. Many of these distributions and companies are gone now, but they were important at the time.

The tests were based on reading from standard input. You could just pipe the random input from the fuzz program to application, or store the random input in a file (good for reproducibility) and then run the application with that test input file.

As we mentioned in the previous chapter, we could crash from 25% to 33% of the applications on all of the platforms. The list of applications that crashed or hung included debuggers, compilers, word processors, network utilities, and other important programs. In some ways, it was disturbingly easy to generate crashes.

Of course, we published everything – the fuzz tool and test jig source code, the test scripts, the data inputs, the results, and our suggested bug fixes.

39.3 The 1995 Study

Several years later, from our anecdotal experience, it did not seem to me that application reliability was getting any better. So, in 1995, we revisited the fuzz command line studies.

This time, we tested nine distributions of UNIX, including three that we tested previously in 1990. Overall, the results had gotten better, but just not enough better.

There were some First, all the three of the previously tested versions of UNIX had improved but were still in the 18% to 23% failure range. This improvement was just not good enough in our judgment. The types of programs that crashed or hung was similar to the list of programs from the 1990 study.

Second, disturbingly, some of the *exact same bugs* that we found in 1990 were still present in 1995. Either no one else had either found them or found them worthy of fixing. Certainly the results from the 1990 study did not have the impact that we had hoped.

Third, of the commercial versions of UNIX, we could crash from 15% to 43% of the applications that we tested. NeXT was the worst performer at 43% crashes. It was no surprise that the company folded shortly after that time.

The fourth and perhaps the most interesting result was that open source versions of the system utilities from GNU and Linux[1] had noticeably better crash rates than the commercial versions. This result generated a lot of discussion at the time as it was the first concrete and impartial comparison between open source to commercial (closed source) software. This discussion is still continuing now.

39.4 The 2006 MacOS X Study

In 2006, when we looked at the reliability of applications running on MacOS X. As part of that student, we tested its command line applications.

Note that MacOS X was a big advance in the design of MacOS. It was the first version of MacOS that was based on a real operating system kernel with strong protection. MacOS X was built on a FreeBSD base with the addition of Cocoa messaging system from the NeXT computer and the Aqua windowing system from previous versions of MacOS.

We tested a lot of applications, 135 in all. And the crash rate was good, only 7%. However, we still, we have to ask why it was not lower, given that by 2006, fuzz testing was getting pretty well known.

The programs that crashed included some surprising ones like the widely used `vim` editor, the `nroff`/`groff`/`ditroff` word processors, and `zsh` shell.

[1] Note that the GNU and Linux distributions were separate at the time.

39.5 The Most Recent (2020) Study

Our most recent study in 2020 was a reprise of the original command line study, this time looking at three of the most widely used versions of UNIX today, Linux, FreeBSD, and MacOS X. Given how widely fuzz testing had been adopted, and how much work had been done on new and advanced fuzz testers, we did not think that we would find many failures. How, we were surprised that we could still crash 12 to 19% of the applications that we tested using the original testing techniques without any of the modern enhancements.

OF the systems that tested previously, Linux's crash rate increased from 9% to 12% and MacOS X increased from 7% to 16%. This result was both surprising and discouraging.

FreeBSD, which we never tested before, came in at 19%. This was much higher than we expected. FreeBSD is frequently used in important systems like storage servers, so we hoped for better.

The open question as to why these failure rates were so high given the wide familiarity and availability of static analysis, fuzz testing, and memory safety checking tools?

39.6 A Comparison?

In Figure 1, we see a graph of the best and worst rates that we found in each command line study (noting that the 2006 study only include one system, Mac OS).

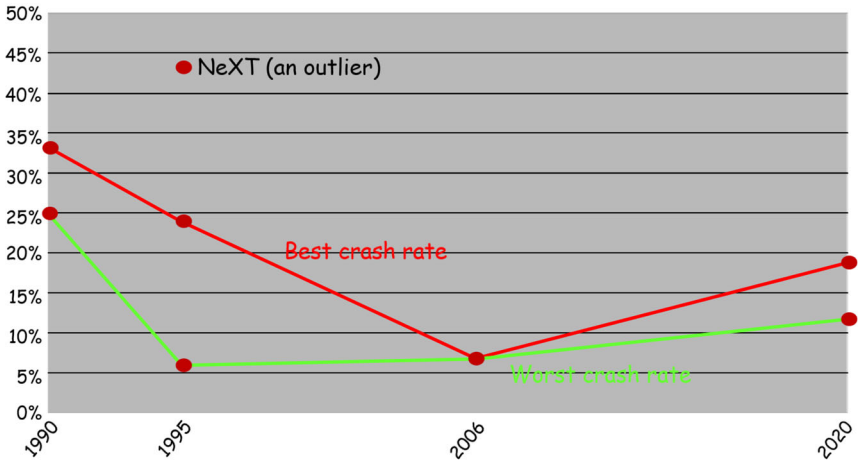

Figure 1: Best and Worst Crash Rate for the Command Line Studies.
Note NeXT Plotted Separately.

Note that in 1995, NeXT had such dismal performance that we graphed it as an outlier. NeXT was a company started by Steve Jobs in 1985 after he was forced out of Apple. They designed a sleek, stylish, high performance workstation. The NeXT operating system took FreeBSD and added the communication (message passing) system from the Mach research operating system project at Carnegie Melon University. This system was never that reliable and the company ultimately failed.

Figure 2 is a table that summarizes the command line failures on the various systems that we tested over the years. There are a few things to note.

First, in 1995, we retested three systems that we tested in 1990 (SunOS, HP-UX, ands AIX). In each case, these systems improved. Encouraging.

		# tested	# crash/hang	% crash/hang
SunOS	90	77	20	29%
	95	80	18	23%
HP-UX	90	72	24	33%
	95	74	13	18%
AIX	90	49	12	24%
	95	74	15	20%
Solaris	95	70	16	23%
Irix	95	60	9	15%
Ultrix	95	80	17	21%
NeXT	95	75	32	43%
GNU	95	47	3	6%
Linux	95	55	5	9%
	20	74	9	12%
MacOS	06	135	10	7%
	20	76	12	16%
FreeBSD	20	78	15	19%

Figure 2: Summary of Command Line Results

Second, also, in 1995, we tested two versions of UNIX from Sun: SunOS that was based on the BSD kernel and the new Solaris that was based on the System V kernel from AT&T. Not surprisingly, the applications had similar behavior on both of these systems.

Third, as previously mentioned, we tested Linux first in 1995 and MacOS first in 2006, and retested each of these systems in 2020. For reasons that we

do not understand, the reliability of both of these systems got noticeably worse over the intervening years.

39.7 The Bugs that Were Found

Now, we will dive into the results and look at what caused these failures. As mentioned previously, in each study, we debugged each failure and then organized them into categories to try to understand them better. Here's what we found:

1. **Arrays and pointers:** The number one category was buffer overrun or overflows, i.e., running off the end of an array because of misuse of either pointers or array subscripts. Since most of the applications were written in C (and later in C++), this should not be such a big surprise. Clearly this is an argument for using a more modern programming language like Rust or Go. What is surprising is that this type of error is still widely prevalent in code today.
2. **Ignoring return codes:** There were a surprising number of cases where error return values were ignored. This is just careless programming. In Chapter 41, we will talk about a further investigation of this category.
3. **Signed characters:** There was the careless use of signed characters. Note that the char type in C and C++ is typically an 8-bit integer. The ISO standard is silent on whether it is a signed or unsigned integer. It could be signed on one platform or compiler and unsigned on another.

   If you use the 8-bit ASCII standard, then you can end up with the high order bit on, which can make the number look to be negative. Any calculations based on the character's numeric value can end up as a negative number. If the number is subsequently used as an array index, you walk off the bottom of your array. This type of calculation is common in hash tables.
4. **Subprocesses:** You need to be careful who you trust. If your program starts another program as part of its functionality, and if that other program fails, then your program is likely to fail. It is quite common for one program to start another. For example, when you type gcc or clang, you do not actually run the compiler. Instead you run a program that executes a sequence of other programs, including the preprocessor, compiler core, optimizer, and linker.
5. **Bad error handlers:** We see error handling appear again. In this case, the programs detected an error, but just handled it in an ineffective way. Given that error cases are rare compared to normal execution, it's not surprising (and strangely ironic) that error handling code is often less reliable than the normal code.

6. **Complex program state:** In our most recent study, we observed that programs are keeping increasingly complex internal state. Often unnecessarily complex. Random input can cause that state to be inconsistent, resulting in crashes or hangs.
7. **Other causes:** We had a grab bag of other causes, including using unsafe input functions, divide by zero errors and incorrect end-of-file checking.

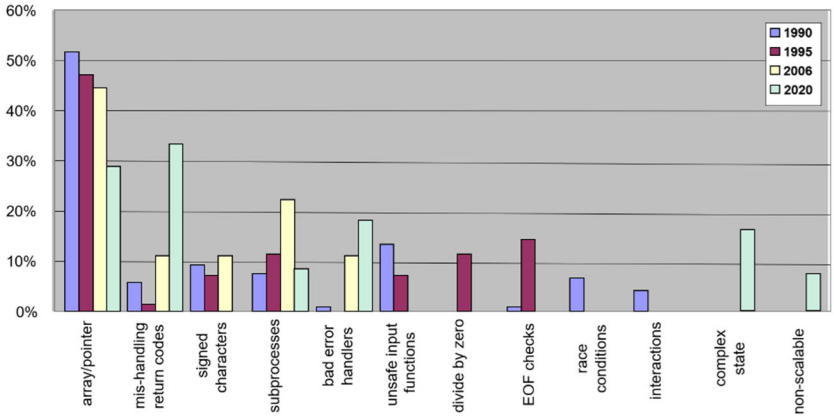

Figure 3: Comparison of Failures Across Categories for Command Line Studies

Figure 3 compares the percent of each type of failure across the command line studies. Note that the last two categories, complex state and non-scalable, only appeared in the most recent (2020) study.

39.8 Analysis of Some of the Coding Errors Found

Now, we will dive a bit deeper and look at a few examples of the coding mistakes that allowed these failures. When you are looking at these examples, try to visualize how you would program in these cases.

We note that sometimes these bugs just stay around for years and sometimes they just get worse. We have seen multiple examples where either the bug was not fixed or that it was fixed and that fix just introduced a more complex version of the bug.

39.8.1 Signed Characters Bugs

The first example, in Figure 4, is a crash caused by the careless use of signed characters, which then caused an array subscript to be out of range. The crash occurred in the spell checker program that was available on UNIX in the late 1980's.

```
00  char *wp, *bp;
01  long h;
02  for ( wp = bp, h=0, ...; ...; ++wp ...)
03  {
04    h += *wp ...;
05    ...
06  }
```

Figure 4: Signed Character and Pointer Bug (spell, 1990)

On line 2, we see a loop that is going through a character buffer, adding up the values of the characters. Note that wp a pointer to a character, an 8-bit value that, on this platform, is signed. Further note that h is a signed long integer. On line 4, the character value pointed to by wp is implicitly recast the match the type of h. The input that caused the crashed had characters with the high-order bit on, so were negative numbers. When that 8-bit negative numbers were recast to be long integers, the sign bit was extended. h was subsequently used as an array index, so the reference went beyond the bottom of the array.

```
01  register int h;
02  ...
03  register char *s = name;
04  for (h = 0; *s != '\0';)
05    h += *s++;
06  h %= TBLSIZE;
```

Figure 5: Sign Character and Pointer Bug (eqn, 1990 and 1995)

Figure 5 shows an example from the 1995 study, where we see a similar error in eqn, the equation formatting tool for the troff word processing package. Again, we are looping through an input character buffer, summing the contents in h, a variable of type signed integer. Given input with the high-order bit on, we again get negative numbers. Even though we subsequently use the mod ("%") operator on h, we still end up with a negative number since the mod of a negative number is defined to be negative. The crash happens when h is later used as an array subscript.

```
01  char c_left;
02  ...
03  c_left = *input_line_pointer;
04  op_left = (operatorT)op_encoding[(int)c_left];
05  ...
```

Figure 6: Sign Character and Array Subscript Bug (as, 2006)

Figure 6 shows another example from the 2006 study, where we see a similar case in the assembler (a pretty important piece of software). We take a character from the input buffer on line 3 and then on line 4 immediately use it as an array subscript. When the input character is negative, we index beyond the bottom of the array.

39.8.2 Sentinel Value Bugs

A common kind of error is the misuse of a sentinel character. If you scanning through a buffer in a loop and terminate the scan when you find a particular value, that value is called a sentinel value. Note that this means that the stopping condition for the loop is finding a particular value in the loop, not when you reach the end of the buffer.

```
01  void null_terminate(char *s) {
02    while (*s != ' ')
03      s++;
04    ...
05  }
```

Figure 7: Sentinel Character Bug (`bibtex`, 1995)

In Figure 7, we see code from `bibtex`, a citation program for the `latex` word processor. The loop starting on line 2 is looking for a space character. However, if the buffer does not contain any spaces, there is nothing to stop the loop from referencing beyond the end of the buffer.

What went wrong here? Good programming practice says that anytime that we loop through a buffer, one of the termination conditions of the loop must make sure that our pointer or subscript does not go past the end of the buffer.

Figure 8 shows another example from the 2006 study, where we see a similar bug in the `zic` time zone conversion program. On line 3, we see a loop that is trying to find a matching double quote character in the input buffer, so the sentinel character is the double quote. And, of course, there is nothing in the loop termination condition about the size of the buffer.

```
01  ...
02  while ((*dp = *cp++)!= '"')
03    if (*dp != '\0')
04      ++dp;
05    else
06      error(_("odd number of
07              quotation marks"));
08  ...
```

Figure 8: Sentinel Character Bug (`zic`, 2006)

```
01  char line[4*BUFSIZE];
02  ...
03  sp = line;
04  ...
05  do {
06      *++sp = c = getc(inf);
07  } while ((c != '\n') && (c != EOF));
```

Figure 9: Sentinel Character Bug (ctags, 1995)

Figure 9 shows our last example, this time from ctags, the C program cross reference tool. In this case, the programmer was trying to be careful, but still allowed the loop to walk off the end of the buffer.

We start by noting this strange variable declaration on line 1. The multiply-by-4 is just odd. Burying constants in your code is a bad habit.

Now, looking at the loop, we see a call to getc on line 6 to get the next input character and put it into the buffer. The termination conditions for this loop include checking for the newline character and end-of-file. That is all correct but neglects to also check to see if the code has reached the end of the buffer.

39.8.3 Return Value Bugs

Now we will look at another common programming error and see an example that turned up in the lldb debugger in the 2020 study. This error is caused by the programmer using a cool-looking compact style that hid an unchecked error condition.

```
persistent_var_sp =
   GetScratchTypeSystemForLanguage(nullptr, eLanguageTypeC)
      ->GetPersistentExpressionState()
      ->GetVariable(exp);
```

Figure 10: Return Value Bug (lldb, 2020)

This code in Figure 10 can be pretty hard to read so take a moment to try to understand what is going on. This code is basically one statement based on a call to GetScratchTypeSystemForLanguage.

Note that GetScratchTypeSystemForLanguage will return a null pointer when the input values are poorly structured, as they are likely to be when used with random input. However, there is no check on the return value of this function and this pointer return value is then immediately dereferenced (that is the first "->" operator), of course causing an error.

39.8.4 Dangerous Input Functions

Programming without regard for the length of a buffer can also be caused by the use of dangerous input function. This practice is all too common and dates back to the attack where the Robert Morris Jr. worm[2] took down the *entire Internet* in 1987. The `gets` system library call was the source of the main vulnerability used in this attack, and was also the cause of failures in both ftp and telnet in our 1995 study. As you probably know, `gets` takes a pointer to the buffer in which to put the input, but has no parameter to specify the length of the buffer.

And this behavior for `gets` was not a surprise at the time. The Solaris manual page for `gets` in the 1990's was quite explicit: they "strongly recommended that `gets` be avoided in favor of `fgets`", which does have a length parameter).

(And if that warning was not explicit enough (and apparently it was not), the current Linux man page has the following warning:

> 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.

Certainly, these warnings (which we highlighted in red) are direct and serious. Which leads to the question that if this function is so dangerous then why do not they just remove it from the system library?

Unfortunately, if your code uses `gets`, then switching to `fgets` can be surprisingly annoying. First, `fgets` takes a parameter that specifies the I/O stream, and passing that extra parameter down a long call chain could cause a lot of code to change. Second, and more annoying, is that `gets` removes the newline character from the input and `fgets` keeps it.

```
char buff[BUFSIZE]
...
cin >> buff;
```

Figure 11: Example of Use of C++ ">>" Operator for Character String Input

We should not just blame UNIX and C for this kind of dangerous function. If you look at the C++ input operator "`>>`", which is used on `iostream` input, there is no default limit on the input size. In Figure 11, we see a string being read from the standard input stream, `cin`, and the results placed in `buff`. Note

[2] `https://en.wikipedia.org/wiki/Morris_worm`

that there is nothing to specify the size of the buffer being used with the "`>>`" operator.

You can limit the size of the input with the `width` method on the iostream, but that interface is problematic for two reasons. First, you have to remember to call `width`. And second, each time you have a different buffer size, you have to call `width` again. So good practice is to call it before each use of the input operator (which is annoying).

39.9 Summary

In this chapter, we have learned about the command line fuzz studies that were based on the classic fuzz testing technique. We have seen how often this can expose serious programming errors, even on software released with recent operating systems.

And, more importantly, we have looked into some of the programming practices that allowed these errors to happen. Understanding these errors will help you to think defensively when you program, hopefully avoiding these and other errors.

39.10 Exercises

1. Write a simple fuzz test for the command line and run it on a few simple commands that take standard input. Hint: for UNIX systems, you can use `/dev/urandom` as a source of random bytes. Warning: for safety, avoid using commands that might cause damage, such as `rm`.
2. Write a simple program that reads standard input and writes standard output that contains an intentional bug. For example, the program might output lines with an even number of characters twice, or three times for odd length, but lines that begin with "!" an infinite number of times. Fuzz test your program with your testing from Exercise 1 to find the bug.