Chapter 41
Classic Fuzz Testing: Other Studies

Revision 3.0, December 2025.

Objectives

- Learn about fuzzing network services and a window system.
- Learn about fuzzing program internal behavior, specifically checking of return codes.

41.1 Introduction

In the previous two chapters, we have seen how fuzz testing can be applied to command line and GUI-based application programs. As fuzzing is a general testing technique, we can apply it to a wide variety of situations. For example, in Chapter 38, we mentioned that it could be used to test multithreaded programs, and in the exercises in Chapter 40, we mentioned applying it to phone applications. And, if you if you read the fuzz testing literature, you will find it has been applied to many other situations.

In this chapter, we describe three other ways in which fuzz testing was applied during the early (1995) fuzz studies.

The keys to applying fuzz testing to these new situations are:

- Identifying where and how to inject the random input.
- Creating a test oracle that can detect a simple failure criteria that is independent of the test input being generated.

41.2 Testing of Network Services

we decided to apply fuzz random testing to network services. It was a fairly simple step to build a test jig that could attached to a network service at a particular host and port number and send it random input. We did this for a test host, where we could monitor the services to see if they crashed. The services that we tested included `rlogind` and `telnetd` (early predecessors to `sshd`), `fingerd` (the service that allowed the first internet worm to spread), timed, `ftpd`, and `rpcd` (that was used as part of the NFS network filesystem).

As network services either use text (ASCII) commands or binary messages, this testing was an application of structure fuzz testing. The testing was just an initial exploration of this approach, so it was not surprising that these tests did not expose any crashes or hangs. More recent research and commercial development has shown that this approach can find new sources of errors.

You have to be careful when testing network services. First, you should restrict the testing to computers to which you have authorized access. Sending random network requests to someone else's computer could be perceived as an aggressive action. Second, you local computing environment may have network intrusion detection and endpoint protection enabled, so your rand testing might trigger alerts that will concern your IT and security staff.

41.3 Testing a Window Server

In the previous chapter, we talked about testing GUI-based applications on UNIX by injecting random events from the window system to the application. If you look at the diagrams from this testing, they are based on being able to intercept communication between the application and the windows server. This approach worked well of UNIX X-Windows, Microsoft Windows Win32, and MacOS X Aqua windowing systems.

But what about the reliability of the window system itself? How can we test that? Given that we can intercept and modify messages between the window client (application) and the window system, we can modify and inject input to the server as easily as we can to the client.

We performed some basic experiments on the UNIX X-Windows system to try to cause crashes. Communication between the client and the server is based on a packet protocol[1], where each package contains an operation and parameters. As with the GUI-based application testing, this is another application of structured fuzz testing. So, the tests consisted of filling in the fields of the packets with random values.

The bottom line is that these tests were not able to crash the X-Windows server. In retrospect, this result was not surprising. In the early days of GUI-based applications on UNIX, most programmers used the low-level Xlib as their interface to the windowing system. This library exposed all the details of each packet type and parameter. And the protocol between the client and server was quite complicated. So, a programmer's first attempts at an application that used X-Windows was, in effect, a random request generator. So early uses of X-Windows exposed many bugs based on the crazy requests that were coming from the novice programmers.

41.4 Testing the Checking of Function Return Values

From our earliest tests of command line programs described in Chapter 39, we saw how prevalent were cases where crashes were caused by not

[1] https://en.wikipedia.org/wiki/X_Window_System_core_protocol

checking the return value of a function, especially in a call to a system library. As result, we decided to use random testing to try see if it would expose this type of error.

Specifically, we focused on one particular (and important) class of system library functions, dynamic memory allocation. We looked at calls to the `malloc` family of memory allocation routines (including `calloc` and `realloc`) that are used extensively in C, and form the foundation of operations like `new` in C++ and other languages. When these calls are used to request new memory be allocated, they should return either the address of the allocated memory or a `NULL` (zero) pointer when there is no more memory to allocate. No more memory means that the system is out of swap (paging) space and no user program can start or acquire more memory.

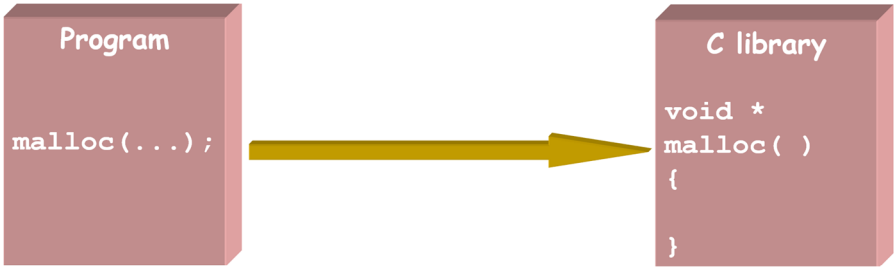


Figure 1: A Program Calling `malloc` in the C Library

To dynamically allocate memory, a program calls `malloc`, which is part of the standard C library, `libc`, as shown in Figure 1. Our testing goal was to change the return value to zero on randomly selected calls. To make this change, we needed to intercept the calls to `malloc` with our own proxy version of `malloc`. This form of interception is called *function wrapping* or function *interposition.*

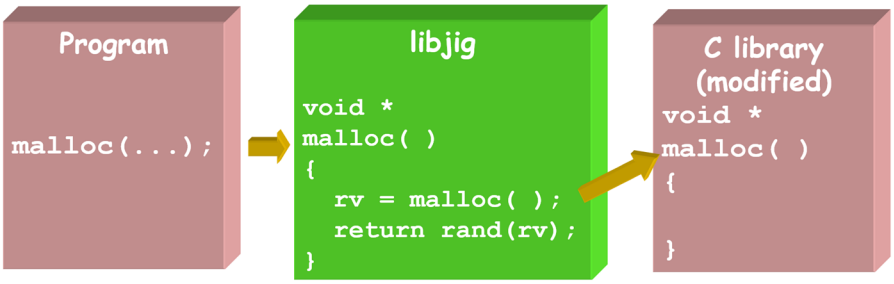


Figure 2: Intercepting the Call to `malloc`

Our proxy `malloc` calls the real `malloc`, and then either returns the real return value from `malloc` or zero, indicating a failure. The choice of real return value or zero is based on random selection. The oracle was a familiar one: did the program crash? The goal was to see if programs checked the

return value from the `malloc` family of calls and how well they handled this error case (if at all).

The short answer is that they did not handle this well. 23 of the 53 programs crashed, for a failure rate of 47%. Almost half the programs that we tested in this way crashed. And some pretty important ones crashed.

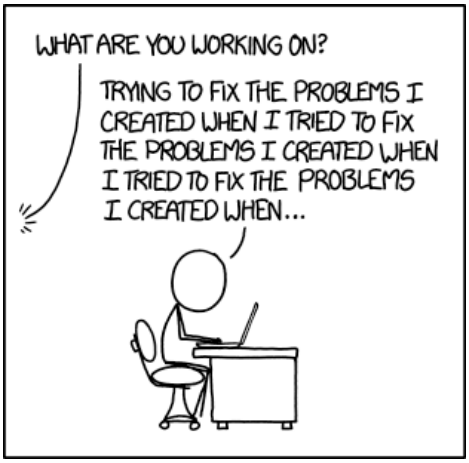
bar	**df**	**login**	rup	tsortl
cc	**finger**	**ls**	ruptime	**users**
checknr	graph	man	rusers	vplot
ctags	iostat	mkstr	sdiff	**w**
deroff	**last**	**rsh**	symorder	xsend

Figure 3: Programs that Crashed when Tested with Random `malloc` Error
Notable programs labeled in **bold/blue**.

Remember that this kind of crash occurs when the system runs out of swap space. So, when it occurs, you might want to know which file system partition filled up using the `df` utility, but that would not be productive as it crashed when malloc returns a zero. Or you might want to see who is logged in to the host with utilities such as `users`, `w`, or `finger`, but they all crashed. Or you might try to login in as root to fix things, but `login` crashed.

These results were definitely not a pretty picture and showed a problematic failure scenario. Note that, in this case, we tested only memory allocation calls. It would be interesting to test other types of library and system calls, perhaps starting with various file open calls.

41.5 We Give xkcd[2] the Final Word



[2] `https://xkcd.com/1739/`. Used by permission of the author.

41.6 Summary

Fuzz random testing has an almost unlimited potential for application to different areas of software and hardware development. In this chapter, we shared some early explorations of a few of these areas. In the years since fuzz testing was introduced, the research community has experimented with so many interesting applications of fuzz testing. You should be encouraged to experiment with your own variations.

41.7 Exercises

1. The testing of function return values in Section 41.4 produced a lot of failures. Try the same kind of testing use various file open calls.
    a. First build a proxy version of the file open calls that you will test.
    b. Determine how you will random change the return value of the open calls.
    c. Select a variety of programs that you have written and system programs, and test them with your proxy version.
2. Besides memory allocation calls and file open calls, what other types of system library calls would be productive to test?