File and protocol attacks

FUZZING

We explain how file or protocol fuzzing leads to direct improvements in code quality. You'll also learn more about available open source fuzzing tools. **BY KURT SEIFRIED**

When I started out in computers a long time ago, I remember you could open files (images, documents, you name it) without having to worry about viruses because, well, everyone knew you could only get viruses from executable files like *.exe*, *.bat*, and *.com* in the Windows world. In the Linux world, if you could actually get your email client to run an executable attached to an email automatically, you probably had the good sense not to. I remember laughing at someone who asked if they had to worry about getting a virus from looking at images sent to them via email. The times, they have changed.

File-based attacks are now commonplace; at the time of this writing, a PDF-based exploit had become public, and Adobe had announced that they would fix it – in about two weeks. In the meantime you can … uhh … not read any PDF files, I guess, if you use Adobe Reader to view files. Luckily, most Linux users do not use Adobe Reader to view PDF's, but many will have Adobe Flash installed (exploitable via SWF files before Flash Player 10.0.12.36 for Linux), or OpenOffice (EMF and WMF files before OpenOffice 2.4.2). The list goes on and on: image files, font files, web pages, you name it. Many major applications have at some point failed to parse the files properly that they were designed to handle, allowing attackers to create files that can execute arbitrary code when opened.

How Fuzzing Works

The basic premise of file or protocol fuzzing is you subtly (or not so subtly) create input that is malformed. This can be done by such methods as simply flipping random bits in a file or protocol stream to creating a full-featured program or test suite that knows how to generate valid (and thus invalid) files or input for various protocols. An excellent example of a file fuzzer is Michael Zalewski's mangleme CGI program. If you install this CGI program on a web server and a client web browser connects to it, the *mangle.cgi* script creates a randomly generated HTML file that includes a *META REFRESH* tag, causing the web browser to reload the script (and get a different randomly generated HTML page). This lets you point a web browser at the URL with the *mangle.cgi* script and wait for the browser to crash (which it will, eventually, in most cases). The use of *META REFRESH* addresses one of the biggest problems with fuzzing files: getting a program to load a series of files one after the other in an automated fashion so that a human being doesn't have to sit there opening files.

This leads to some interesting problems and opportunities when fuzzing. On the one hand, you can very quickly and very cheaply run a program against a battery of tests: Leave a browser pointed at mangleme for a few days and it will go through several tens of thousands of test cases. Each request generates a log entry with the identifying number of the test case, which allows you to reproduce the problem. Given this, however, it can still be difficult to reproduce or verify results in some cases. For example, with Internet Explorer version 7.0.5730.13, the mangleme test case *0x6dc61276* doesn't appear to have any effect, except on one machine on which I have PowerDVD 6.0 installed, in which case it will reliably crash Internet Explorer and create an empty file called *su0.mpg* on the desktop without any prompting. This shotgun approach to security testing is not terribly accurate, but, much like blasting away with a shotgun in a forest, you will eventually hit something of interest (it just might not be what you're looking for).

Intelligent Fuzzing

Random fuzzing presents several problems. For example, you are unlikely to find

certain boundary conditions by simply trying random junk; integer overflows and underflows are a common problem, but to trigger them you need to mangle a file or protocol traffic in a specific manner. Many programs use C's *signed long int* (which supports values from -2147483647 to 2147483647) to specify the length of data fields and so on. This leads to several interesting values: *0x7fffffff*, which is 2147483647 (the maximum), and *0x80000000*, which is -2147483648 (the minimum value that can be stored). If the program does not properly calculate values, you can end up with a situation in which adding two positive numbers together results in a negative number or adding a negative number to another number results in an even larger number. If these values are used to allocate memory for storing user-supplied data, then a classic buffer or stack overflow is the typical result, and these often are exploitable and can allow for arbitrary code execution.

Write Once, Run a Million Times

One of the things I find most fascinating about software is that almost all the cost in creating it is in the first copy, once you've done that you can stamp out a million copies for virtually nothing. Writing a file fuzzing or protocol fuzzing tool is no different. Understanding a file format is not cheap. The basics of PDF, for example, are defined in a 756-page document [1]. This does not cover the entire contents of what can be in a PDF because you can embed various image formats and even JavaScript into a PDF file now.

Now I'm guessing that a complete set of documentation for PDF, all the image files it supports, JavaScript, etc. would run to several thousand pages (which is why running a PDF file viewer with fewer features is probably a good idea if you are worried about security). However, once you have read this documentation and built a full-featured fuzzer, chances are good that you would be able to find exploitable bugs quickly. An example of this is the PROTOS test suite [2] from the University of Oulo in Finland. Using a framework from Codenomicon Ltd., a relatively small group with a limited budget was able to write protocol testing suites for WAP, http, LDAP, SNMP, SIP, H.323, ISAKMP, and DNS. The test suite was so successful at finding bugs, the CVE project was unable to assign the normal one CVE number per bug and instead had to aggregate them under a few numbers because so many flaws were found.

Open Source Fuzzing Tools

If you're looking to play with fuzzing tools, or just generally stress test your system and software, a number of options are available (Table 1).

Some of the tools, like mangleme and QueFuzz, can be up and running in minutes. Others, like SPIKE, have a pretty steep learning curve and are aimed more at people wanting to write their own custom fuzzing tools for research purposes (they have a learning curve shaped much like the Matterhorn).

Where Does This Leave You?

The good news is that fuzzing tools have lead to direct improvements in code quality. It's hard for a developer to argue with a test case (in the form of a file or a network data stream) that causes your application to fall over or otherwise behave badly. In a best case scenario, this could even lead to developers writing more robust code that isn't as prone to bad or malformed data inputs, although if history is any indicator, this isn't likely to happen anytime soon. The bad news is that as bad guys get smarter, they to will start using fuzzing tools to find flaws that they can exploit (witness the current 0-day attacks against Adobe Acrobat and Microsoft Excel and the Conficker worm, which had reportedly infected 15 million Windows systems as of January 26, 2009). ■

Table 1: Fuzzing Tools

Tool	Description	URL
mangleme	A random HTML generator	*http://lcamtuf.coredump.cx/soft/mangleme.tgz*
Browser Fuzzer 2	A browser fuzzer that generates random CSS, DOM, HTML, and JavaScript	*http://www.krakowlabs.com/dev/fuz/bf2/bf2.tar.gz*
fzem	Email client fuzzer (generates invalid server replies, etc.)	*http://www.krakowlabs.com/dev/fuz/fzem/fzem.tar.gz*
fsfuzzer	Filesystem fuzzer (ntfs, ext3, ext2, vfat, iso9660, etc.)	*http://projects.info-pull.com/mokb/fsfuzzer-0.6.tgz*
FileP and FileH	Python- and Haskell-based fuzzers that mutate a list of files and feed them to applications	*http://www.isecpartners.com/file_fuzzers.html*
ProxyFuzz	A man-in-the-middle fuzzer that can randomly mutate network traffic	*http://theartoffuzzing.com/downloads/proxyfuzz/proxyfuzz.py*
Peach Fuzzing Platform	An entire fuzzing platform with data modeling and state modeling	*http://peachfuzzer.com/*
GPF	Takes network captures, modifies the data, and sends it to a server to see what happens	*http://www.vdalabs.com/tools/efs_gpf.html*
SPIKE	C-based fuzzer creation toolkit for network fuzzing	*http://www.immunitysec.com/resources-freesoftware.shtml*
QueFuzz	Uses iptables to intercept and mutate network packets	*http://code.google.com/p/quefuzz/*

INFO

[1] PDF specification: *http://www.adobe.com/devnet/acrobat/pdfs/PDF32000_2008.pdf*

[2] PROTOS: *http://www.ee.oulu.fi/research/ouspg/*

THE AUTHOR

Kurt Seifried is an Information Security Consultant specializing in Linux and networks since 1996. He often wonders how it is that technology works on a large scale but often fails on a small scale.