# 2 Traditional Methods for Bug Hunting

Hunting for bugs in an application can be a daunting task especially when that application consists of hundreds of files and hundreds of thousands of lines of code. There are tools which simplify the process through automation. Source code analysis can catch a lot of errors, but even after these are all fixed and the application is compiled, other bugs can only be found through testing with both good and bad data. This chapter presents some background on the most frequently used types of automated tools for software validation.

## 2.1 Static Code Analysis

The first type of automated software validation tool to be created was the static code analyzer (SCA). It is presented here because it is foundational for all other tools that have been developed. Static code analysis is an automated process for scanning source code looking for bugs. The analyzer can process a single file, a small set of files, or an entire code tree at a time. "Such tools can make finding bugs, or software defects, faster and cheaper than manual inspections." (1) During the scanning process, the analyzer attempts to match source code to a variety of pre-defined rules to identify bugs. The rules that are used can be as simple as ‘do not allow the use of a particular function’, or as complex as ‘if a particular set of statements occur in this order, it is known to be vulnerable.’ Each analyzer has its own set of rules, and some rulesets are more comprehensive than others. Therefore it is recommended that at least two SCAs be used in conjunction and their reports compared. (2)

If an analyzer finds a bug, it writes a message to a report often listing the statement that is in error, and often giving suggestions on how to fix the problem. The biggest problem with static code analysis (SCA) is that the reports tend to contain a large number of false positives – pieces of code that have been flagged as a bug but which are fine. When analyzing an entire code tree, the reports can become very unwieldy. Usually a security analyst is assigned to sift through the reports and remove the false positives so the developers can concentrate on fixing the valid bugs. (1)

Listing 1 illustrates the type of errors that can be found by an SCA.

|  |
| --- |
| 1 int main**()**  **2 {**  3 int arr**[**10**];** // create a 10-byte array  4 arr**[**10**]** **=** 0**;** // assign a value past the end of the array  5 **return** 0**;**  **6 }** |
| Listing 1 – Inserting Outside an Array Bounds |

In this code segment, a ten-byte array is defined. The elements in the array are numbered from 0 to 9, but in the fourth line of the program, a value is assigned to element 11 of the array which goes outside the bounds of the array. Running this code through an SCA generated an error saying that the assignment was to an element out of bounds.

## 2.2 Fuzz Testing

Once all the bugs have been found by static code analysis, and the program is compiled, it is time for fuzz testing. While the program is running, invalid or malformed data is given to the program’s inputs and the program is monitored for failures. A program crash, hang, or unexpected program behavior is considered a failure. Fuzz testing or fuzzing is considered a negative testing technique – it looks for what isn’t in the program like input validation routines, and proper bounds checking. (3,4)

Fuzzing dates back to 1988 and a classroom assignment by professor Barton Miller to his operating systems class. (5) Miller had observed that electrical interference on his modem connection from a thunderstorm was causing some Unix programs to crash or behave unexpectedly. He termed this phenomenon fuzz. The assignment given to his class was to write a random generator to be used to provide purely random data to Unix utilities until they failed.

Miller continued his fuzz testing over the next almost 20 years, but the fuzzers he was using were very simple and only found very superficial bugs.

During the late 1990s and early 2000s, new types of fuzzers were being developed which could dig deeper into application and find more elusive bugs. These fuzzers followed one of two basic models: model-based and mutation-based – to be discussed next.

### 2.2.1 Model-based Fuzzers

Model-based or generation fuzzers utilize a protocol definition to structure the fuzz data for an application. Once the protocol definition is known, the fuzzer can ensure that the underlying structure of the data (where binary or character data should be, and how long the fields are), then within those constraints, the data is randomized. This approach allows the application being tested to ‘think’ it is receiving real data and it tries to process it instead of rejecting it off-hand because it just doesn’t look like valid data.

Often the protocol being tested can be broken down into small segments or blocks. A subset of model-based fuzzers that work with these types of protocols are termed block-based fuzzers and include Spike, the Sulley Fuzzing Framework, and PeachFuzzer. (6) These block-based fuzzers are especially well suited for fuzzing network protocols, and file formats because of their very regid well-defined structures.

Listing 2-2 illustrates an input file to Spike to perform a simple http request from a web server.

|  |
| --- |
| 1 s\_string("GET /"); // The manditory first part of an http get request  2 s\_string\_variable("myapp/index.html"); // The string to fuzz is the URL  3 s\_string\_var s\_string(" "); // a second variable to be fuzzed  4 s\_string("HTTP/1.1\r\n"); // Fixed mandatory http version field of the get request  5 s\_string("User-Agent: Wget/1.13.4\r\n"); // the first of several http headers to ensure completeness of the request  6 s\_string("Host: 192.168.1.1:80\r\n"); // ip address of the host running the webserver  7 s\_string("Accept: \*/\*\r\n");  8 s\_string("Connection: Keep-Alive\r\n");  9 s\_read\_packet(); // Read a response from teh web server  10 sleep(3); // delay before trying again  11 s\_read\_packet(); // try reading a response from the server again |
| Listing 2-2 A Spike HTTP Request |

Based on this script, Spike will send a series of GET requests to the webserver at address 192.168.1.1 port 80. The basic URL is <http://192.168.1.1/myapp/index.html>. Spike will fuzz the URL to create a wide variety of new URLs to test.

### 2.2.2 Mutation-based Fuzzers

A mutation-based or replay fuzzer starts with a known set of good inputs like a captured network packet, and fuzzes specific parts of that input one-at-a-time to generate new inputs to send that look almost like the original. Many of the web fuzzing frameworks like BurpSuite, Zed Attack Proxy and WebScarab use this style of fuzzing. The user captures a network packet, selects specific parts of the packet to fuzz and resends the packet with the fuzzed data included.

## 2.3 Symbolic Execution

Symbolic execution was first developed in 2005 by Microsoft. During testing, the symbolic executer builds a logical formula defining the branching conditions in the program (7). This formula is then checked to see if any combination of variable values makes the formula true or ‘satisfied’. If the formula is satisfied, then this particular set of variable values is saved as a testcase to ensuring that this particular path can be tested in the future.

After the testcase is saved, the symbolic generator negates the last condition in the formula leading to the last branch taken, and checks the formula again to see if it can be satisfied. The process repeats until all paths through the program have been tested or an artificial limit of tests has been reached (8).

Theoretically, a symbolic executer could check every possible path through a program yielding 100% code coverage. In practice, however, because medium to large programs have thousands of possible execution paths, the executers can and often do run out of memory to hold all the constraints in the logical formula. Various modeling heuristics have been used to identify the ‘best’ paths to test so the number of constraints can be reduced (8–12). These heuristics mean that the actual code coverage is reduced.

## 2.4 Concolic Execution

Concolic execution is similar to symbolic execution in that a running program is traced and a logical formula of the execution paths is built and solved. However, concolic execution starts with ‘concrete values’ and transforms them to symbolic variables. Rather than checking every possible execution path through the program, a concolic executer is only concerned with those constraints that are affected by ‘tainted data’, data that comes from user inputs (13–16). If an execution path is not affected by tainted data, then the concolic executer ignores the path.

Listing 2-3ss smple jva tst metod whch uld beseigSmoathFndfo AS.

|  |
| --- |
| 1 public int myMethod(int x, int y) {  2 int z = x + y;  3 if (z > 0) {  4 z = 1;  5 } else {  6 z = z - x;  7 }  8 z = 2 \* z;  9 return z;  10 } |
| Listing 2-3 Java test method |

Listing 2-3 shows part of a test java application used for demonstrating the Symbolic PathFinder concolic executer written and maintained by NASA (17). This method has two conditions and two input variables. Since both variables “x” and “y” could be affected by user input to this method, they will be checked by the concolic executer.

|  |
| --- |
| Test Case 1: y = -9999999, x = 10000000  Test Case 2: y = -10000000, x = 10000000 |
| Listing 2-4 Generated testcases |

The two testcases in Listing 2-4 (17) that were generated correspond to the two conditions from listing 2-3:

* “if z > 0” and
* “if z <= 0”

The generated values for x and y will result in the program’s taking the two alternate paths through this method.