# 2 Traditional Methods for Bug Hunting

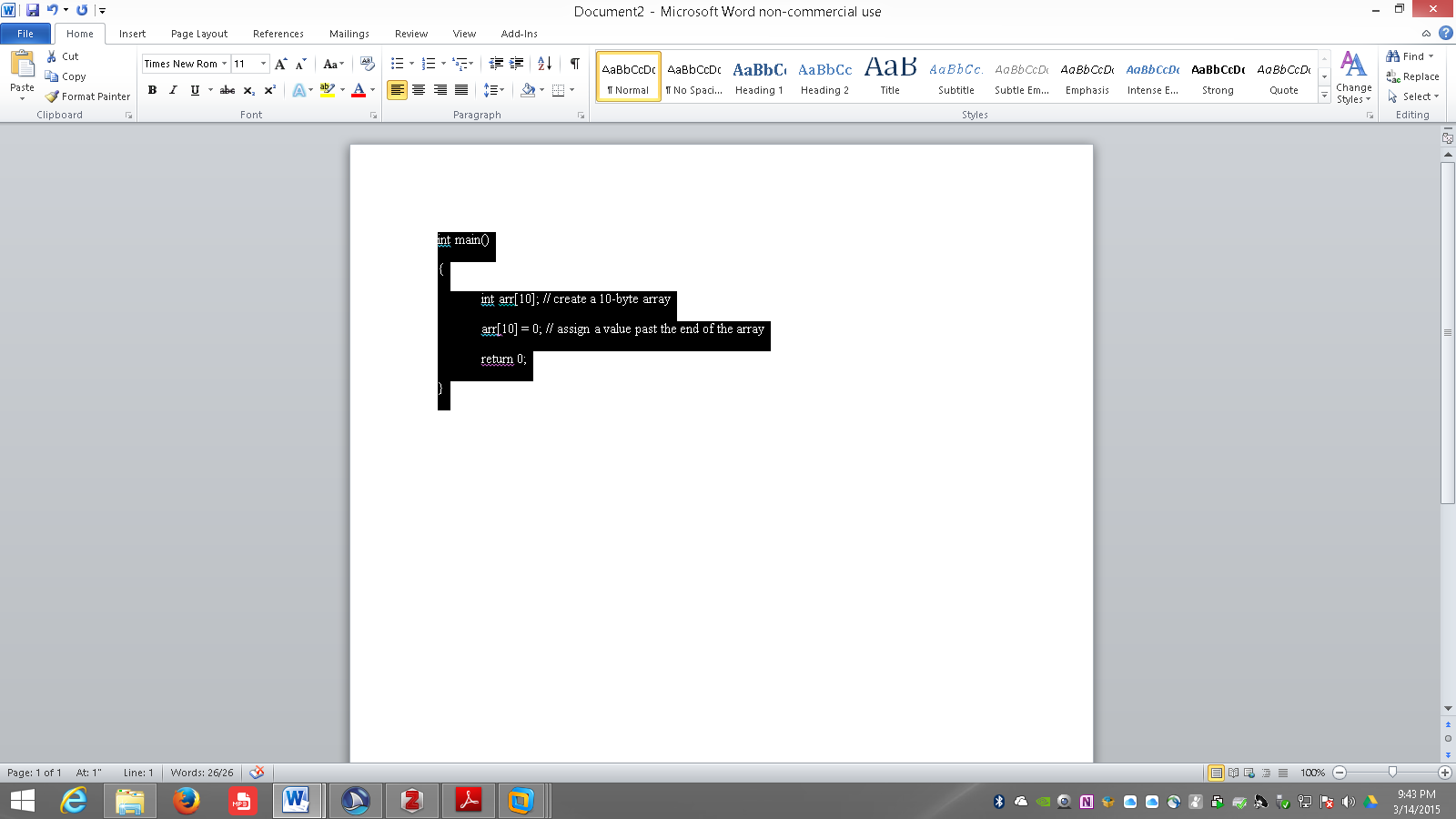
Hunting for bugs in an application can be a daunting task especially when that application consists of hundreds of files and hundreds of thousands of lines of code. There are tools which simplify the hunting process through automation. Even after checking the source, there is another problem – when the application is compiled, the compiler and linker can introduce bugs into the final product. It takes several types of tools to track down those elusive bugs so the final released program is as secure and bug free as possible. In this chapter, I will address several of the tools used by developers, quality assurance engineers, and security analysts to find bugs.

## 2.1 Static Code Analysis

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)

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

Figure 1

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.

Figure 2-1 illustrates an input file to Spike to perform a simple http request from a web server.

[[insert figure here]]

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

The third in the set of tools used to hunt for vulnerabilities is a relative new-comer to the field. Symbolic Execution or dynamic test generation has been under development since the mid 2000s. Symbolic execution provides a theoretical 100% code coverage possibility, but this high coverage rate is only possible with small applications because of the extreme resource requirements for test generation.

During symbolic execution, the input datat to tan application is traced through the application, and all possible paths are enumerated along the way. If the input data causes a particular branch to be taken, this is recorded for later use. Once the initial test run has completed, the model of all possible paths traversed by the data is examined. (7) Starting at the terminal end of the paths each decision point is considered. The dynamic test generator determines how the original input data would have to change to force the application to take an alternate branch at that final decision point, and a new set of test data is created to force that alternat path to be taken. This process of evaluating decision loints and the data required to change outcomes is iteratively repeated until new test cases have been generated to theoretically test all alternate paths through the application. As one could imagine, with a large application, the number of alternate paths could become enormous, so some level of weighting is applied to determine the best paths to traverse and test thereby reducing the code coverage from the 100% possible limit. (8)