The Importance and Implementation of Effective Quality Assurance Techniques

Anonymous
Rochester Institute of Technology

ABSTRACT

Effective quality assurance is a key element of secure coding. While many errors can be fixed before they are a problem when writing code, programmers are imperfect. Quality assurance helps to stop some of the problems that programmers unintentionally added to the source code. This paper analyzes the techniques, practice, and implementation of fuzzers, penetration testing, and source code auditing.

ACM Reference format:
Anonymous. 2019. The Importance and Implementation of Effective Quality Assurance Techniques. In *Proceedings of NA, NA, 2019,* 5 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 OVERVIEW

It is difficult to ensure that code is written properly and securely. Software verification is even an undecidable problem [5]. There has been a lot of study over the years in verifying that software meets its demands, but much work is still needed. Several publications were analyzed to provide further knowledge on this topic. Some of these publications follow. You et al. present a description of an enhanced method of fuzzing a program, called ProFuzzer [11]. This fuzzer understands, to some extent, the program that it is fuzzing. This allows the fuzzer to drastically reduce the search space. As a result of this, the team that created the ProFuzzer found 42 zero-days in 10 intensively tested programs, generating 30 CVEs. Antunes and Vieira present an examination of penetration testing and source code analysis on the detection of vulnerabilities in web services [1]. They found three critical observations from their experiments: the effectiveness of static code analysis is typically better than penetration testing, false positives occur at a relatively high frequency, and different tools implementing the same analysis and technique frequently find different results. Ko, Cai, Lyu, and Wong discuss useful application schemes of quality assurance testing in the emerging technology of component-based software [8]. They explain how component-based software is a new idea that is not implemented widely yet, but has the potential to be implemented widely because it is much cheaper to produce. Their proposed quality assurance is to test the components individually then in the entire system. The individual component tests must be tested to be free from all known defects and they must have these tests done in the target environment. McGraw and Chess provide an explanation of the importance of static analysis of code in order to increase security [2]. They explain the benefits and drawbacks of manual and automatic source code testing with tools, and why most people choose automatic source code testing with tools. They explain the overall evolution of static source code analysis tools. A simple overview of this evolution is that the tools have been understanding the source code more and more and how they will run when compiled and the addition of many patterns that can be indicative of errors and bugs. Spadini et al. present a relatively novel concept of static analysis of code, test-driven code review, in which the tester examines a patch of code by writing tests for the code [10]. Test-driven code review is not a common practice in the code production. This team had more than 100 developers use test-driven code review. The results show that the reviewers found the same proportion of defects in source code but find many more errors in the test code, compared to the traditional code review technique.

The rest of this paper is structured as follows: section 2 describes the background and history of quality assurance. Section 3 discusses fuzzing, its common techniques and future progress. Section 4 discusses penetration testing with its common techniques and implementation as well as the direction that it is heading in the future. Section 5 explains how source code auditing was first used and has evolved in the workplace as well as improvements that can be made.

2 BACKGROUND

Quality assurance for computer software has been a necessity for almost as long as computers have been programmable. Quality assurance is the final step in the software development life-cycle, and if any errors are undetected by the quality assurance then large and expensive problems can occur. In 1945, Grace Hopper first coined the term ”computer bug” relating to errors in the hardware or software of a system, so errors in source code have been around for many years. Quality assurance has been an important part of software development for a long time and is here to stay.

3 FUZZ TESTING

Fuzz Testing, or fuzzing, is the practice of inputting massive amounts of data into a system in the attempt of causing undesired effects [7]. Fuzzers can be implemented naively by inputting random data, or can be implemented with the tested program in mind. Both approaches have their benefits and drawbacks. The random fuzzer is the better option in situations in which the tested program's internals are unknown or too complicated to understand or if no specific target for breaking the tested program is made. The random fuzzer is also very easy to implement, sometimes making environments with a small range of possible inputs the perfect candidate for random fuzzing. The main drawback of the random fuzzer is that it is slow. Programs with just a few inputs can possibly have a number of combinations greater than a trillion or quadrillion. The number of combinations of the possible inputs increases exponentially as the size of the inputs increases. For example, a password that can use letters and numbers only needs to be 8 characters long to be intractable to try all combinations. A random fuzzer would have a much smaller chance of finding an error as opposed to the heuristic fuzzer, given the heuristic fuzzer has a little more guidance on what inputs to give the program. The following code shows the implementation of a random fuzzer written in Python3.

```
counter = 0
while counter < 10000:
    for i in range(3):
        inputs[i] = random.randint(0, 255)
    for i in range(3, 6):
        inputs[i] = random.randint(0, 10000)

    get_score()

    if len(breaking_values) > 3:
        return breaking_values

    counter += 1
```

In the known description of this program, the first three inputs to the program have a range of possible values from zero to 255 and the last three inputs have a range of possible values from zero to ten thousand. There are 1.6581375E+019 possible combinations for this program. The benefits of the heuristic fuzzer is that it can find breaking input to programs with much, much more input than the random fuzzer can in a shorter amount of time. The drawbacks of this fuzzer is that it requires a significant more amount of work to write and the program to be tested must be known or have targetable properties. The following code shows the implementation of a heuristic fuzzer written in Python3.

```
def climb():
    global inputs

    for n in range(100):
        #find the best value to change
        change = 1
        best = get_score()
        climbed = False
       best_inputs = [inputs[0], inputs[1], inputs[2],
            inputs[3], inputs[4], inputs[5]]
        if(best == "zero"):
            return True

        for i in range(len(inputs)):
            for diff in [-1*change, change]:
                inputs[i] = inputs[i] + diff
                score = get_score()
                if score == "zero":
                    print("breaking values: ", inputs)
                    breaking_values.append(inputs)
                    return

                if float(score) > float(best):
                    best = score
                    best_inputs = [inputs[0], inputs[1],
                        inputs[2], inputs[3], inputs[4],
                          inputs[5]]
                    climbed = True

                inputs[i] = inputs[i] - diff

        if climbed is not True:
            return best

        inputs = best_inputs

    return best


def random_restart_hill_climb():
    print("climbing...")
    global inputs

    counter = 0
    while counter < 1000:
        for i in range(3):
            inputs[i] = random.randint(0, 255)
        for i in range(3, 6):
            inputs[i] = random.randint(0, 10000)

        climb()

        if len(breaking_values) > 3:
            return breaking_values

        counter += 1
```

This fuzzer was designed to target the system for any divide-by-zero errors. From mathematics we know that vertical asymptotes occur when dividing by zero. This knowledge can be used to deduce that when approaching an input value of the vertical asymptote, the output value will either approach infinity or negative infinity. This fuzzer only implements the positive infinity approach. It is only a reverse of a greater than sign to a less than sign. This approach to finding the maximum value is called hill climbing and is incomplete. When a local maximum is found, this approach fails. Repeating this process with random inputs is complete and will find all global maximums. This program will find all possible global maximums given enough repetitions. This method is called random-restart hill climbing.

The techniques for the two classes of fuzzers, random and heuristic, are very different. There is no more to discuss on the random fuzzer. The heuristic fuzzers are an ever-evolving topic in the world of quality assurance. The better fuzzers are the ones that ”understand” the program the best. These fuzzers can reduce the search space dramatically while still keeping the possible values that can break the program in the search space. Currently, there are many cutting edge fuzzers available for commercial use. These include AFL, AFLFast, VUzzer, Driller, and QSYM [11]. They all attempt to reduce the search space by one way or another of understanding the program. While these fuzzers are effective, they can be improved.

A whole new class of fuzz testing is called Whitebox Fuzz Testing. This class of fuzzers uses the outputs of the program to learn how the program works, by relating the inputs to the outputs, and changing the inputs to update to this [4]. This type of fuzzing can be used to fully understand a system and can therefore find the breaking inputs faster. This class of fuzz testing is currently in its infancy and can be studied more to produce formidable fuzzing tools in the future.

4 PENETRATION TESTING

Penetration testing is a security practice where a cyber-security professional or company attempts to identify and exploit vulnerabilities in a computer system [3] The purpose of this is to find the common weaknesses that can be exploited before any malicious actors can do this first. Penetration testing comes in the form of black box testing, white box testing, or gray box testing. In black box testing, the ethical hackers attempting to exploit the system have no knowledge of the system, just as malicious actors would. White box testing means that the ethical hackers have complete knowledge of the system. In gray box testing, the penetration testers have partial knowledge of the system. Penetration testing usually refers to an entire IT infrastructure test suite, but this analysis will focus on the code [9].

Penetration testing has several phases: data collection, vulnerability assessment, actual exploit, and result in analysis and report preparation. There are several penetration testing types, but only one is directly stems from secure coding, web application testing. Web application testing attempts to find the vulnerabilities in an application.

Penetration testing uses a large list of possible vulnerabilities and check if the system is susceptible to any of the vulnerabilities. Some example vulnerabilities follow. Verifying that all usernames and passwords are encrypted and transferred over secure connections, such as https. This relates to CWE-311: Missing Encryption of Sensitive Data and CWE-319: Cleartext Transmission of Sensitive Information. Verifying information stored in website cookies is not in a readable format. This relates to CWE-311: Missing Encryption of Sensitive Data. Verifying that previous vulnerabilities are no longer vulnerable. Verifying all passwords meet the required standards and that the standards are strong. This relates to CWE-521: Weak Password Requirements. Usernames are not easily guessable, such as ”admin” or ”administrator”. This also relates to CWE-521: Weak Password Requirements. The username is a credential that is required and a guessable username causes a weaker set of credentials. The application login page should be locked upon a few unsuccessful login attempts. This relates to CWE-307: Improper Restriction of Excessive Authentication. Generic error messages and no internal system details are revealed in any error or alert. This relates to CWE-209: Information Exposure Through an Error Message. Verifying special characters are handled properly. This relates to CWE-138: Improper Neutralization of Special Elements. Custom error messages are used in the event of failure. This relates to CWE-756: Missing Custom Error Page and CWE-209: Information Exposure Through an Error Message. There should be no hardcoded usernames or passwords in the system. This relates to CWE-798: Use of Hard-coded Credentials. Verifying application for SQL Injection. Verifying the application for Cross-Site Scripting. Verifying if the application is returning more data that it is required. This relates to CWE-200: Information Exposure [6].

There are many, many more vulnerabilities and weaknesses to search for. Many of these vulnerabilities are direct consequences of insecure code and can be categorized as one of the common weaknesses enumerated by MITRE. These vulnerabilities can all be tested manually or a tool that automatically runs these tests can be used.

The code written to accompany this examination has several vulnerabilities that penetration testing would find. The first vulnerability relates to CWE-311: Missing Encryption of Sensitive Data. The code for this is:

```
def password_authentication():
    host = ''
    port = 20020
    with socket.socket(socket.AF_INET,
        socket.SOCK_STREAM) as s:
        s.connect((host, port))

        for p in range(3):
            username = input("username: ")
            password = input("password: ")
            print("\033[A", end="")
            print("password: ", end="")
            for c in password:
                sys.stdout.write(' ')
            print()
            total_string = username + " " + password
            s.sendall(total_string.encode())
            time.sleep(1)
            recieved = s.recv(1024).decode()

            if len(recieved) > 0:
                if recieved[0] == '1':
                    print("Welcome!")
                    return True

            print("password incorrect")

    print("You shall not pass[word]!")
    sys.exit(0)
```

This code sends the username and password without encryption over the network. The next vulnerability relates to CWE-521: Weak Password Requirements. There are no password requirements enforced. The code written does not handle all exceptions, so any errors found would be the Python errors. This relates to CWE-209: Information Exposure Through an Error Message. The program stores the personal data of the user with an encryption key. This encryption key is hard-coded. This relates to CWE-798 Hard-coded Credentials. The code with this vulnerability is shown below.

```
def get_personal_data():
    password = "hard_to_guess_password"
    encoded_password = password.encode()
    salt = b'salt_'
    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length=32,
        salt=salt,
        iterations=100000,
        backend=default_backend()
    )

    for p in range(3):
        user_input = input("encryption key: ")
        if user_input == password:
            key = base64.urlsafe_b64encode(
                kdf.derive(encoded_password))
            f = Fernet(key)
            file = open("personal_data.txt", "rb")
            encrypted = file.readline()
            decrypted = f.decrypt(encrypted)
            print(decrypted.decode())

            return

        else:
            print("encryption key incorrect")

    print("invalid encryption key. exiting...")
    sys.exit(0)
```

The password, "hard_to_guess_password", is hard-coded. Even when this program is compiled, the `strings` command can find the complete password. On the server, the usernames and passwords are also stored in cleartext. This is shown by the following code.

```
user_pass_dict = {"user": "pass",
    "arh2913": "secure_password124$$"}
```

Even though the user will not have access to the server code, the server can still be hacked and the usernames and passwords can be released.

5 SOURCE CODE AUDITING

Source code auditing is the practice of examining the text of a program statically, without the program being run. Static analysis is run with the hopes of finding vulnerabilities in the source code. Static analysis comes in two forms: manual and automatic. Manual auditing requires a human to evaluate the code. This is time consuming and requires the human to understand the software as well as patterns to look for to identify any vulnerabilities. Automatic auditing is usually the preferred choice for static source code analysis. Automatic auditing does require a human, but only in the sense that the human must analyze the results that the automatic audit generates. This human does not need to have extensive knowledge about the software or the patterns to look for to identify vulnerabilities. The automated auditing is also much faster and requires much less time for the user to analyze the code [2].

Automated source code analysis is very fast compared to manual analysis. As a result, it can be run much more frequently. The first automatic source code analysis in the software development process is the compiler. The compiler is run very frequently to ensure that programs or sub-programs are running correctly. Errors in the source code can be egregious for the compiler to notice and flag. Even if the code is approved by the compiler, it can still be insecure. This requires more in-depth tools to find patterns in the code that can lead to vulnerabilities. These tools look for patterns indicative of vulnerabilities based on patterns. These tools are very useful for identifying vulnerabilities, but cannot find all vulnerabilities. If a tool does not know about a pattern that can cause vulnerabilities, the tool will never find that vulnerability [2].

Static analysis has evolved over the years. At first, tools like `grep` were used. This searched for patterns in the code, but was very simplistic and had a lot of room for improvement. The static analysis tools currently on the market and in further development tokenize the code and attempt to understand how the compiler will implement the code. These static analysis tools will get better the more they understand how the code will be implemented with the compiler. For now, it is safest to use source code analysis along with another form of quality assurance [2].

6 CONCLUSION

Quality assurance is quite possibly the most important element in the world of secure coding. Three common aspects of quality assurance are fuzz testing, penetration testing, and source code auditing. Fuzz testing can be very simple or very complicated. There is a lot of work going into the study of the best fuzz testing tools currently. Penetration testing is most commonly used on entire systems but includes aspects of secure coding. There are many vulnerabilities, enumerated as common weaknesses, that are commonly checked for when evaluating code. Source code auditing is a very effective way to find vulnerabilities in written code. The tools find common patterns in vulnerabilities and report these to the tool operator. Quality assurance, when using these approaches, improves the quality of the tested code and ensures more security in the code.

REFERENCES

[1] N. Antunes and M. Vieira. 2009. Comparing the Effectiveness of Penetration Testing and Static Code Analysis on the Detection of SQL Injection Vulnerabilities in Web Services. In *2009 15th IEEE Pacific Rim International Symposium on Dependable Computing*. 301–306. DOI: http://dx.doi.org/10.1109/PRDC.2009.54
[2] Gary McGraw Brian Chess. 2004. Static analysis for security. *IEEE Security Privacy* 2 (Nov.-Dec. 2004), 76–79. Issue 6.
[3] Inc. Cloudflare. 2019. What Is Penetration Testing? (2019). https://www.cloudflare.com/learning/security/glossary/what-is-penetration-testing/
[4] Patrice Godefroid, Michael Y Levin, and David Molnar. 2008. Automated Whitebox Fuzz Testing. *Network and Distributed System Security Symposium* (2008).
[5] Richard Mayr. 2003. Undecidable problems in unreliable computations. *Theoretical Computer Science* 297, 1-3 (2003), 337fi??354. DOI: http://dx.doi.org/10.1016/s0304-3975(02)00646-1
[6] MITRE. 2019. Common Weakness Enumeration. (2019). https://cwe.mitre.org/
[7] Margaret Rouse. 2019. What is fuzz testing (fuzzing)? (2019). https://searchsecurity.techtarget.com/definition/fuzz-testing
[8] M.R. Lyu Kam-Fai Wong Roy Ko, Xia Cai. 2000. Component-based software engineering: technologies, development frameworks, and quality assurance schemes. *Proceedings Seventh Asia-Pacific Software Engeering Conference. APSEC 2000* 1 (2000), 372.
[9] SoftwareTestingHelp. 2019. A Complete Penetration Testing Guide with Sample Test Cases. (2019). https://www.softwaretestinghelp.com/penetration-testing-guide/

[10] Davide Spadini, Fabio Palomba, Tobias Baum, Stefan Hanenberg, Magiel Bruntink, and Alberto Bacchelli. 2019. Test-driven Code Review: An Empirical Study. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. IEEE Press, Piscataway, NJ, USA, 1061–1072. DOI: http://dx.doi.org/10.1109/ICSE.2019.00110

[11] Shiqing Ma Jianjun Huang Xiangyu Zhang XiaoFeng Wang Bin Liang Wei You, Xueqiang Wang. 2019. ProFuzzer: On-the-fly Input Type Probing for Better Zero-day Vulnerability Discovery. *2019 IEEE Symposium on Security and Privacy* 1 (May 2019), 882–899.