Blind Security Testing – An Evolutionary Approach

Scott T. Stender

scott@isecpartners.com

Introduction

Software testing, or more broadly, software quality assurance, is a difficult problem. Comprehensive testing of any kind, whether functional or non-functional, is impossible. In *The Art of Software Testing,* Myers demonstrated[1] how even programs of trivial size can have a natural test set that, to execute fully, would require millions of years to execute.

To make things worse, Myers' trivial example addresses positive functional testing – evaluating a system with known inputs for desirable outputs. Security testing requires that functional testing be covered, for example by ensuring that an authorization mechanism grants or denies access where appropriate, in addition to testing for non-functional aspects of the system, a much less tractable test.

Functional vs. Non-Functional Testing

For comparison, let us examine the tests required to evaluate a security-sensitive functional portion of the system versus those required to evaluate a non-functional security requirement. First, consider an authorization scheme that encompasses three classes of authenticated users: unauthenticated users, low-rights users, and administrators. A basic test set to exercise this scheme could look like the following:

- Access system as unauthenticated user, access system functionality
    - Verify no access granted
- Access system as authenticated user, access system functionality
    - Verify access is limited to only assets appropriate for low-rights users
- Access system as authenticated administrator, access system functionality
    - Verify that access allows complete control over system assets

The alert reader should be able to identify several more tests for such an authorization scheme. Even with a rigorous, comprehensive test set, one can find a meaningful, finite set of tests that can be executed in a limited amount of time that delivers a high degree of assurance in the proper functional behavior of the authorization scheme.

Now, consider a non-functional security requirement such as "System cannot be compromised." This requirement is not associated with a specific feature nor can a simple test be performed to make sure it is met. Even so, most security-aware developers

and testers can identify several types of flaws that could lead to breaking this requirement. A small subset of such attacks would include:

- Buffer overflows
- System command injection
- SQL injection
- Authenticated session hijacking

The list goes on and on, and a complete yet theoretical set would include classes of flaws that have not yet been identified but that are present in the system. Clearly, the test suite to cover known areas is difficult enough.

If one focuses on just one subset, the problem becomes even more difficult. Consider a test set designed to identify buffer overflows. One might start with a simple test:

- Provide a long string as system input
    - Verify that application does not crash

But what happens if "long string" is taken to mean 100 characters, and the destination buffer is 128 bytes in length? A test for "greater than 128 characters" should be added. What if it must be in a specific range between 437 and 439 bytes in length? Or, what if long strings in general are handled properly, but long sub-strings delimited by a special character cause problems? It is apparent that a general-purpose test suite that has comprehensive coverage will be impossible even for a small subset of the negative testing that must be carried out to consider a system "secure."

A Better Way to Create Test Cases

The exercise above can be carried out ad infinitum for almost every class of common security flaw today – one can always concoct a flawed code snippet that will escape detection for a given set of test cases. Furthermore, if one were to create a test suite that gave a high degree of confidence that most common flaws would be found, it would likely be so large that it could not run in a meaningful time frame.

This paper proposes a method that uses a hybrid evolutionary algorithm to optimize test sets. Commonly-used algorithms and test sets are employed to provide security test case populations, but optimization of these test populations, and individual cases within those populations, is governed by an evolutionary process that, over time, will create an optimized set of test cases that is most likely to find faults.

Test Case Generation

In most security testing tools, test sets are generated according to one of two high-level approaches.

The first approach could be called "Flaw-Specific Test Generation", where test cases that are specifically crafted to identify security flaws are crafted. A number of common tests fall into this:

- Providing apostrophes in input fields for testing SQL Injection

- Providing long strings in input fields for testing for Buffer Overflows
- The use of alternate data encodings to bypass input validation[2]

The second approach, test case generation through the use of random data, is commonly known as Fuzzing[3]. In this approach, random data or random mutations of data are supplied as input.

Most test tools that are built around such tests cases are effective at finding critical security flaws – a sad commentary on the current state of secure software engineering. As a result, smarter test case generation has not been necessary. However, these rather-simple test approaches have limitations which are increasingly apparent as software security improves:

- Both classes, Flaw-Specific Testing and Fuzzing generate test sets large enough to make complete execution infeasible.
- Flaw-Specific tests rarely uncover anything other than what they are intended to uncover
- Fuzzing avoids the narrow scope of Flaw-Specific testing, but at the expense of having a far-from-optimal test set

In short, one has to choose between two test generation methods that have a shared problem of infeasibility of execution but with orthogonal benefits that prevent a tester from choosing just one or the other.

Evolutionary Algorithms

One solution that allows the tester to keep the benefits of the traditional test case generation mechanisms while reducing the total number of cases feasible is to employ an evolutionary algorithm. Evolutionary algorithms attempt to identify optimal solutions to problems using a method inspired by biological systems. Put simply, evolutionary algorithms attempt to find an optimal solution using the following approach:

1. Potential solutions (aka population) are generated
2. Potential solutions are evaluated according to fitness criteria
3. New solutions are generated based on "mixing" criteria
4. Repeat steps 2-3 until a pre-set number of iterations or an optimal solution is identified

In the case of security testing, generating an optimal test set is a natural fit for an evolutionary approach. The difficulty of the problem using traditional heuristics alone and the desire to avoid local optima are both common justifications for preferring an evolutionary approach over other heuristics.

The general evolutionary algorithm described above does not specify several variables that, together, define a given application of an evolutionary algorithm. In order to apply an evolutionary approach to a given problem, one has to define populations and their representations, evaluation criteria for solutions, and selection methods for subsequent

generations. Application of evolutionary algorithms to various problems is an active subject of computer science research[4].

Security Testing and Evolutionary Algorithms

The use of evolutionary algorithms to improve security testing is not new; academic researchers[5,6] and security practitioners[7,8] have performed work in this area. Many practical evolutionary algorithm-based test systems use debugging and/or code coverage to evaluate the effectiveness of a given test case.

An alternative method for test case selection and evaluation is necessary. Though code coverage is a commonly-cited means of evaluating the effectiveness of a test set, its use as a sole criterion is controversial. Refer again to Myers' evaluation of the 100 trillion test cases of a simple program – even though such a test set could exhaustively test the program from a code branch perspective, flaws based on a system dependency, logic errors, and specification errors cannot be found through such means. Furthermore, in security testing, one often does not have interactive access to the system under test. Though interactive access certainly improves test case evaluation by allowing analysis of code coverage, debugging data, and log analysis, a general method that can improve test cases without this dependency can be of great value.

Blind Security Testing – An Evolutionary Approach

One such alternative becomes apparent if one seeks to optimize the test set being used to test for security, in contrast to percentage of code coverage or likelihood of being able to identify a specific kind of flaw. In such a system, one can generate test populations based on traditional security testing techniques and have them "compete" for inclusion in the optimized test set.

Generating Test Populations

For example, if a given application may be vulnerable to Buffer Overflows, SQL Injections, Cross-Site Scripting attacks, and Format String bugs, one can generate populations of Flaw-Specific cases to test each of these areas. The discussion around creating test suites for Buffer Overflow testing should have illustrated that each individual test set can be quite large. Flaw-Specific tests alone can easily generate hundreds of thousands of test cases for a small number of parameters and still be incomplete.

Though we have a large population of test cases, one still has the problem of "local optima", meaning that test cases are highly optimized for identifying specific types of flaws. To escape such optima and identify unexpected kinds of flaws, one can pull from random test sources, including purely random data and random mutations of data.

After this exercise, a test population generated using traditional heuristics or known best practices would consist of several Test Classes, each of which encompasses a large number of test cases:

- Buffer Overflow Testing
    - Long strings of a single character

    - Lengths of strings with common boundary conditions:
        - 128 bytes, 256 bytes, 1024 bytes, 65535 bytes…
    - Varying string patterns
    - Random lengths of strings
- SQL Injection Testing
    - Apostrophe
    - Quotation mark
    - Comma
    - Bracket
    - Alternate encodings of the same
- Cross Site Scripting Testing
    - Less-Than Sign
    - Greater-Than Sign
    - Quotation Mark
    - Apostrophe
    - Alternate encodings of the same
- Format String Testing
    - %s, %x, %n
    - Various repetitions of the same
- Random Data Testing
    - Purely random data included in requests
    - Purely random data included as parameters
    - Encoded random data included as parameters
- Random Mutation of Valid Data Testing
    - Bit flipping of known legitimate data
    - Bytestream sliding within known legitimate data

Of course, the classes of tests and individual tests within them can be expanded far beyond what space would permit – this serves only as an illustrative starting point.

Measuring Test Quality

The "Blind Security Testing" reference in the title of this paper refers to how feedback into the evolutionary algorithm is accomplished. In order to avoid dependence on interactive system access, this approach relies only upon the behavior of the system under test and the natural qualities of the test sets.

The most broadly-applicable means for identifying the behavior of the system under test is to identify the difference between a control case, one where known valid data is presented and behavior is acceptable, versus the test case. This difference has two qualities:

- The magnitude of the difference
- The presence of behavior that suggests errors

Consider a web application and its behavior under test. One can establish control cases by examining the output from HTTP attack proxies or by non-malicious "spidering" of the web site. When tests are executed against a given request, the results of the test are compared with the control case. For each such execution, a simple textual diff of the response would be sufficient to identify both added and removed data from the test result.

The magnitude of the test case can be calculated by taking the length of the removed data and adding it to the length of the added data. One can account for vastly-different sized requests by dividing this value by the size of the response in the control test. Such a metric would capture system behavior as divergent as "Server did not respond," "Server responded with an exception trace," and "Server returned the entire database". The magnitude of the test case is significant in that one would expect normal behavior to mesh reasonably well with the control case – identifying large differences from this is indicative of a problem.

The presence of errors falls into both "general" and "flaw-specific" test evaluation. If one considers the same web application, it can be assumed that any test case whose result included the word "error" in the added portion of the calculated diff would be of interest. The same could be said of "exception", "trace", or other words that are often associated with general errors. In addition, one can improve the evaluation of the Flaw-Specific tests, like SQL Injection, by including test-specific evaluations. Such an evaluation would look for words like "database", "odbc," "jdbc," or other words that are typically associated with a SQL injection attack. An even more-sophisticated approach would be to attempt blind exploitation of the probable flaw to increase confidence in the evaluation.

Together, both magnitude and error detection provide a reasonable basis upon which the test case can be evaluated.

Test Case Selection

Once a test population and fitness criteria have been established, the means by which test cases are selected must be defined. As a reminder, the goal is to find an optimal test set for a given application.

One can take advantage of the fact that applications are often written by a development team that employs common technologies and coding guidelines. In such cases, the application is usually more susceptible to specific kinds of flaws. For example, it is mostly a waste of time to test for format string attacks on a web application implemented in managed code. Most experienced penetration testers would focus instead on flaws that are more common with such environments such as Cross-Site Scripting and SQL Injection.

Test Case selection follows this general idea. High-level Test Classes, such as "SQL Injection Testing" or "Random Mutation Testing" are each assigned an initial probability of execution. Test cases within each class are arranged in a prioritized queue – tests arranged towards the front of the queue have a greater chance of being executed than those at the end of the queue. During a given test execution, a test case is selected first by probabilistic selection of the Test Class, and then by probabilistic selection of a case within that class.

The test case is then evaluated according to the fitness criteria established in the previous section. If the test case is determined to have been effective at finding a flaw two things happen: the probability that the Test Class will be executed will be increased, and the test case itself will be moved up the queue within its test class.

This arrangement is beneficial in that it is consistent with the considered judgment of experienced penetration testers: First, Test Classes are emphasized and de-emphasized based on their applicability to a given application. In the example above, one would expect Cross-Site Scripting and SQL Injection tests would be executed more often than buffer overflows. Second, test cases within the Test Classes that are most effective at identifying flaws will be executed more often. In the example above, one would expect that apostrophe or its alternate encodings will be executed more often than other potentially bad SQL characters.

This two-level arrangement ensures that a single test case will not overwhelm the test set – one does not want to quickly drive towards the local optima of Cross-Site Scripting tests to the exclusion of the rare Buffer Overflow that might afflict a small part of an application. A key requirement of this system is to avoid local optima. By taking a probabilistic approach to both Test Classes and individual cases, it can be ensured that the test set will not be overly optimized to find a single kind of flaw.

One potential problem of the above scenario arises if classes of security flaws tend to "bunch" within the application. This could be the case when different developers or development teams write code that is vulnerable to different classes of security flaws. In such a case, the probabilities associated with a given Test Class or test case will thrash between competing populations.

In such a scenario, one can gradually reduce the change in probability associated with a successful or unsuccessful test. This is a similar parameter to the "temperature" parameter associated with Simulated Annealing. By choosing the rate of temperature change, one can tune the test engine to evaluate enough cases to be able to make reasonable decisions around optimization but not thrash once an optimized test set is available.

Limitations

One should note that the above approach does not ensure an optimal test set – it will always provide an optimized test set that takes into account both flaw-specific testing and random testing. In addition, like all general techniques (and especially blind security techniques) there will be applications whose behavior foils the general nature of the evaluation criteria. Though such examples no doubt exist in the set of applications worldwide, the approach outlined here is sufficient to be applicable to most applications.

Furthermore, the general nature of this approach means that one can simply plug in different test populations or evaluation criteria to augment the security tests performed. Indeed, if interactive access is available to the system under test, the use of code coverage, debug information, and log analysis would be useful input for the test engine.

Future Research

This general framework can be extended in many ways. The most obvious and already-cited improvement would be the addition of detailed feedback from the system under test to further improve the fitness tests. In addition, the test case selection criteria presented here are rather simple and request-centric. By introducing stateful tests into the test selection and cross-Test Class splicing of test cases, one could further refine the test set to be both more optimal and identify flaws that would not be found in the approach presented here.

Conclusion

This paper has presented an approach that uses an evolutionary algorithm to optimize test sets based on traditional security test cases and random data sources. The goal, to create an optimized test set based on these sources is achieved, though much optimization remains possible. Though the optimal test set remains unidentified, and it can be argued it is unidentifiable, this approach can drastically reduce the number of test cases required to get a given level of security assurance, and is structured in such a way as to be available with minimal dependencies on the system itself.

---

[1] Myers, Glenford J., Badgett, Tom, Thomas, Todd M., and Sandler, Corey. The Art of Software Testing 2ed. New Jersey: Wiley, 2004.

[2] Stender, Scott T. "Attacking Internationalized Software", *Black Hat USA 2006*, Las Vegas, August 2006

[3] Forrester, J.E. and Miller, B.P., "An Empirical Study of the Robustness of Windows NT Applications Using Random Testing", *4th USENIX Windows Systems Symposium*, Seattle, August 2000.

[4] Michalewicz, Zbigniew and Fogel, David B. How To Solve It: Modern Heuristics, 2ed. Verlag Berlin Heidelberg New York: 2004

[5] Michael, C. C., McGraw, G. E., Schatz, M. A., and Walton, C. C. *Genetic algorithms for dynamic test data generation*. Technical Report RSTR-003-97-11, RST Corporation, Sterling, VA, May 1997.

[6] Pargas, R. P., Harrold, M. J., AND Peck, R. R. *Test data generation using genetic algorithms*. The Journal of Software Testing, Verification and Reliability 9 (1999), 263-- 282.

[7] Embleton, Shawn, Sparks, Sherri, and Cunningham, Ryan. "'Sidewinder': An Evolutionary Guidance System for Malicious Input Crafting", *Black Hat USA, 2006,* Las Vegas, August 2006

[8] DeMott, Jared, Enbody, Richard, and Punch, Bill. "Revolutionizing the Field of Grey-box Attack Surface Testing with Evolutionary Fuzzing", *Black Hat USA 2007*, Las Vegas, August 2007