WHITEPAPER

SYNOPSYS®

Fuzz Testing Maturity Model

May 30, 2017

Jonathan Knudsen

Mikko Varpiola

TABLE OF CONTENTS

TABLE OF CONTENTS

Fuzz Testing Maturity Model

Fuzz testing is an industry-standard technique for locating unknown vulnerabilities in software. Fuzz testing is a mandatory portion of many modern secure software development life cycles (SDLCs), such as those used at Adobe, Cisco Systems and Microsoft. This document provides a framework to assess the maturity of your processes, software, systems and devices. At the heart of this document is a maturity model for fuzz testing that maps metrics and procedures of effective fuzz testing to maturity levels. The maturity model is a lingua franca for talking about fuzzing, allowing different organizations to communicate effectively about fuzzing without being tied to specific tools.

A key benefit of this model is that it enables organizations to make informed decisions about the time and resources needed to perform fuzz testing as part of product development or product verification. Furthermore, it helps you plan and understand where you are with your fuzzing efforts.

Even if your product development, validation, or acceptance testing procedures do not adhere to one of the aforementioned SDLCs, fuzzing remains a crucial technique for locating vulnerabilities in software.

This version of Fuzz Testing Maturity Model (FTMM) focuses on defining a framework which allows comparing and exchanging information about fuzzing using measurable and interchangeable metrics. It does not define or mandate how any particular organization chooses to implement it or measure it.

This document does not contain specific self-certification or third-party certification criteria or guidelines.

After a description of the maturity model levels, this document presents a summary of requirements and explains the meaning of the requirements in detail. This is followed by a discussion of practical considerations and desirable features of fuzzers. Finally, appendices provide example reports and information about documentation.

Introduction

Fuzz testing is an effective technique for locating vulnerabilities in software. This document describes a Fuzz Testing Maturity Model (FTMM) by mapping metrics and procedures to maturity levels. The maturity model is a *lingua franca* for talking about fuzzing, allowing different organizations to communicate effectively about fuzzing without being tied to specific tools.

The world of fuzzing can be roughly divided into network protocol fuzzing, file format fuzzing, and other disciplines. Different fuzzing disciplines have slight variations in execution, injection modes, and failure modes. Nevertheless, this maturity model is a viable unified framework to address all fuzzing disciplines.

Builders and buyers

Fuzzing is valuable to both builders and buyers. A builder creates software; a buyer consumes and uses it. For example, a medical device manufacturer is a builder, while a hospital is a buyer.

Builders use fuzzing to discover and fix more bugs during product development. Buyers use fuzzing to verify and validate the products they procure and use.

Builders frequently obtain some portions of their products from third-parties, which means many builders are also buyers. A builder can use fuzzing both for improving their own software as well as verifying the software they buy.

Fuzzing is a black box testing technique, which means access to the source code of the target is not required. While builders should have access to their own source code, buyers usually do not; fuzzing is an effective tool regardless.

Targets and attack surface

FTMM and FTMM levels apply to a specific target. A target is a single piece of software or a collection of software. The target is the thing that is fuzz tested, often referred to as System Under Test (SUT), Device Under Test (DUT), interface, firmware, system, service, etc.

Here are some example target types:

- A single executable file
- An operating system
- An industrial controller
- A network router
- A mobile phone
- A smart television
- An automobile
- A medical infusion pump

Consider, for example, a typical Web server, a Linux system running Apache or Nginx. Each place on the box that some piece of software accepts some kind of input, either through a network protocol or as a file, is an attack vector.

- The Web server accepts HTTP input.
- TLS protocol handling is most likely supported through the openssl third-party library.
- TCP and IP protocols are supplied by the Linux operating system.
- The server might have other services running as well, such as SSH.
- If the server is running an application, it probably accepts other types of input or files as well.
- USB and other ports on the server can take various kinds of input.

Each attack vector represents some body of code, code that contains vulnerabilities. The attack surface of the Web server in this example is the collection of all of its attack vectors. The attack surface of an entire network or a complex system is simply the sum of the attack surfaces of its component devices.

Fuzzers

In simplest terms, a fuzzer is a piece of software that tests for bugs in another piece of software, the target.

A great fuzzer has the following features:

- A savvy test case engine creates the malformed inputs, or test cases, that will be used to exercise the target. Because fuzzing is an infinite space problem, the test case engine must be smart about creating test cases that are likely to trigger failures in the target software. Experience counts—the developers who create the test case engine should, ideally, have been testing and breaking software for many years.
- Creating high-quality test cases is not enough; a fuzzer must also include automation for delivering the test cases to the target. Depending on the complexity of the protocol or file format being tested, a generational fuzzer can easily create hundreds of thousands, even millions of test cases.
- As the test cases are delivered to the target, the fuzzer uses instrumentation to monitor and detect if a failure has occurred. This is one of the fundamental mechanisms of fuzzing.
- When outright failure or unusual behavior occurs in a target, understanding what happened is critical. A great fuzzer keeps detailed records of its interactions with the target.
- Hand-in-hand with careful recordkeeping is the idea of repeatability. If your fuzzer delivers a test case that triggers a failure, delivering the same test case to reproduce the same failure should be straightforward. This is the key to effective remediation—when testers locate a vulnerability with a fuzzer, developers should be able to reproduce the same vulnerability, which makes determining the root cause and fixing the bug relatively easy.
- A fuzzer should be easy-to-use. If the learning curve is too steep, no one will want to use it and it will just gather dust.

Test cases

A test case is the basic unit of fuzzing. Each anomalized message or file is a test case. With network protocol testing, one test case may consist of a single interaction, with one or more messages exchanged, with an anomaly in one or more of them or in how the messages are sent.

Failures

Broadly speaking, target software fails when it behaves in a way its creators did not intend. In specific terms, the bugs uncovered by fuzzing cause various symptoms, including the following:

- Process crashes and panics
- Process hangs (endless loops)
- Resource shortages (disk space shortage, handle shortage, etc.)
- Assertion failures
- Handled and unhandled exceptions
- Data corruption (databases, log files, etc.)
- Any other unexpected behavior

> Target software fails when it behaves in a way its creators did not intend.

Symptoms revealed by fuzz testing may or may not lead to failure modes such as a denial of service (DoS), degraded performance, information leakage, and compromise of target integrity. These failures can have any number of consequences depending of the purpose and function of the software, where and when it is operated and so on.

When talking about the bugs it is important to keep in mind the following:

1. The relationships between a bug, the symptoms it causes, and the ultimate consequences are entirely unpredictable. A single bug can cause an infinite variety of bad behaviors, and one bad behavior can be caused by a conspiracy of several bugs.
2. Consequences of a bug are arbitrarily related to the cause in time, space, and severity. An extreme example of this is patient death because of a small programming mistake, using = instead of == in a comparison.

Finally, keep in mind that when well-written, well-tested software is deployed in a working environment, most remaining bugs cannot necessarily be attributed to a single line of code, or even to a single program. They result from the unpredictable and untestable conjunction of multiple pieces of software, their environment, the specifics of the configuration, the system state, and even the sequence of steps that led to the anomalous behavior.

FTMM use cases

FTMM gives builders and buyers a set of standard levels for communicating about fuzz testing.

Builders, for example, can promote a product by stating the specific FTMM level achieved. This level would be applied both to the software developed by the builder as well as the software components acquired by the builder.

A buyer asking for a particular piece of software could require a specific FTMM level from vendors. This gives the buyer assurance that the software submitted by vendors has a well-defined baseline of robustness and security.

In addition, a buyer could decide to verify and validate incoming software and devices to a specific FTMM level. From a process standpoint, this implies a specific level of risk awareness for the organization.

The FTMM level required by a buyer, or used by a builder, should be proportional to the value represented by the target to the potential adversary or attacker. For example, a builder of small-business accounting software might find FTMM Level 2 to be sufficient, while a nuclear power facility might require FTMM Level 5 from its software vendors. It is good to keep in mind the old truth that the system is only as strong as its weakest link—this could not be more true in the world of software and systems robustness and reliability.

In addition to builders and buyers, other members of the software ecosystem can also benefit from FTMM. For example, regulatory agencies wishing to improve software quality in a particular industry can require a minimum FTMM level for products from builder organizations. Such a requirement benefits all involved. Buyers get better products and builders save money by creating better products. Ultimately, consumers—ordinary people—benefit by getting better products and services.

Fuzzing in context

Fuzzing is not a silver bullet that will solve all your software vulnerability problems. Fuzzing is great for locating unknown vulnerabilities, but it is a component of a complete vulnerability management process. Along with fuzzing, organizations must follow best practices, manage known vulnerabilities, and use complementary tools for locating and managing vulnerabilities.

Maturity model levels

This section provides an overview of each of the maturity levels, their key requirements, and intended audience. The summary of the levels and the requirements is in section 2, Testing Requirements Overview per Target Attack Vector.

In general, the overall maturity level of a target is the lowest maturity level of any attack vector, with Level 0 representing “no fuzzing has been done.” If a target includes 27 attack vectors that have been fuzzed to Level 5, and 1 attack vector that has not been fuzzed at all, the target is at FTMM Level 0. Security is only as strong as its weakest link—the 1 attack vector that was not fuzzed will likely make the target highly vulnerable, despite the rigorous testing performed on the other attack vectors.

A prudent practice is to balance the risk associated with an attack vector against an appropriate FTMM level. It makes sense to fuzz remotely accessible attack vectors to a higher level than front-panel input, for example.

Fuzzing is great for locating unknown vulnerabilities, but it is a component of a complete vulnerability management process.

Level 0: Immature

If no fuzzing has been performed on any attack vector in a target, the target is at FTMM Level 0. If minimal fuzzing has been done, but does not meet the Level 1 requirements, then the target is still at FTMM Level 0.

Level 1: Initial

Level 1 represents an initial exposure to fuzz testing. Either generational or template fuzzing is used on the known attack vectors of the target, although a full attack surface analysis is not required. For each tested attack vector, fuzzing should be performed for at least 2 hours or 100,000 test cases, whichever comes first. Assertion failures and transient failures are acceptable but must be documented.

Level 1 is not comprehensive in any sense, but it is an excellent first step for organizations that wish to improve their security posture by becoming adept at fuzzing. Some fuzzing is better than no fuzzing. If the target has not been fuzzed previously, FTMM Level 1 can provide quick improvements in robustness and security.

Level 2: Defined

The starting point for Level 2 is an attack surface analysis of the target. For each attack vector, a generational fuzzer should be used for 8 hours or 1 million test cases, whichever comes first. If a generational fuzzer is unavailable for an attack vector, a template fuzzer can be used instead, for at least 8 hours or 5 million test cases, whichever comes first.

Level 2 introduces a more defined approach for performing fuzz testing.

What matters in fuzzing is catching failures. Instrumentation is the mechanism for catching failures, so while Level 2 does not require automated instrumentation, it is highly recommended.

Level 3: Managed

Both generational and template fuzzing must be performed for each attack vector in Level 3. The generational fuzzer must be run for 16 hours or 2 million test cases, whichever comes first, while the template fuzzer must be run for 16 hours or 5 million test cases.

Automated instrumentation must be used. The baseline test configuration must be documented.

Compared to previous layers, Level 3 emphasizes completeness and documentation so that it is easy to observe and improve the fuzzing process. This is an excellent baseline for builders.

Level 4: Integrated

Level 4 increases the fuzzing time per fuzzer type to one week. There is no longer a minimum threshold for test cases—for each attack vector, a generational fuzzer and a template fuzzer must both be run until the minimum required time is reached.

In Level 4, fuzzing must be incorporated in the organization's automated testing.

Level 4 also introduces component analysis, a type of static analysis in which a target binary is examined to understand its internal components, such as third-party libraries. These components might have known vulnerabilities, which could be exposed through the target software. The component analysis provides a comprehensive picture of the components of a binary and their associated known vulnerabilities. It sets the stage for fuzzing—if vulnerable components are present, they can be assessed and replaced to eliminate or mitigate known vulnerabilities. Once the known vulnerabilities have been addressed, fuzzing is used to search for unknown vulnerabilities.

Level 4 is intended for systems with high reliability and security requirements

Parallel execution can be used to reduce the elapsed testing time, while still meeting the required total testing time. For example, for an attack vector, if you can perform the generational fuzzing on eight identical targets, dividing the test cases evenly, then the required one week of generational fuzzing can be accomplished in 168 / 8 = 21 hours.

Level 5: Optimized

Level 5 increases testing time to 30 days for each fuzzing type, and requires the use of at least two different fuzzers per fuzzing type. Because fuzzing is an infinite space problem, and because different fuzzers work differently, using two generational and two template fuzzers increases the probability of locating vulnerabilities.

Target software must be run with available developer tools to detect and monitor subtle failure modes, and code coverage and component analysis must also be performed.

Again, parallel fuzzing can reduce the elapsed testing time. For example, a web browser target that could be virtualized and replicated in the cloud could achieve FTMM Level 5 for one of its required fuzzers by executing 100 parallel test runs in fewer than 8 hours!

For a single attack vector, Level 5 requires the use of two generational fuzzers and two template fuzzers, or four fuzzers run for 720 hours each. For a target with two attack vectors, the required total testing time is as follows:

2 x 4 x 720 hr = 3760 hr

Parallel testing shrinks this number to a manageable size. 250 parallel runs brings the elapsed time to just over 15 hours.

Be aware that Level 5 does not represent the ultimate in fuzzing, or an endpoint in the quest for software quality. More fuzzing can always be performed, but Level 5 represents fuzzing that is appropriate for systems with extremely high reliability and security requirements.

Testing requirements overview per target attack vector

The table shows the requirements for each level of the maturity model. The columns and abbreviations are fully explained in the subsequent sections.

The requirements shown here apply per attack vector on the target.

	Types	Test cases	Time (hours)	Instrumentation	Allowed failures	Test harness integration	Processes	Documentation	Other requirements
5 : Optimized	G T	infinite infinite	720	O,A,D	none	Yes	S,C	RR,S,P,B,C	U2
4 : Integrated	G T	infinite infinite	168	O,A,D	none		S,C	RR,S,P,B,C	
3 : Managed	G T	2,000,000 5,000,000	16	O,A	Tr		S	RR,S,P,B	
2 : Defined	G (T)	1,000,000 (5,000,000)	8	O	Tr		S	RR,S,P	
1 : Initial	G/T	100,000	2	O	Tr,As			RR	
0 : Immature									

G = Generational fuzzer
T = Template fuzzer
O = Human observation
A = Automated instrumentation
D = Developer tooling
Tr = Transient failures
As = Non-DoS assertion failures

S = Attack surface analysis
C = Software component analysis
RR = Results and summary
P = Test plan
B = Document baseline test configuration
U2 = Use two different fuzzers per type

Types

Random fuzzing, test cases are generated using a random or pseudo-random generator. Random fuzzers are minimally effective because the inputs they generate for target software are entirely implausible. Target software will, in general, examine and immediately discard random inputs because they do not look anything like valid inputs. Given enough time, of course, a random fuzzer will produce a test case that resembles valid input, but it takes a long time. In addition, a truly random fuzzer produces a sequence of test cases that cannot be repeated—if a failure is triggered, reproducing it will be problematic, if not impossible.

Template fuzzing, also known as block or mutational fuzzing, generates test cases by introducing anomalies into a valid message or file. Template fuzzers are more effective than random fuzzers, but have some important shortcomings. In particular, the effectiveness of the fuzzing depends on the quality of the template. Furthermore, a template fuzzer often has no knowledge of protocol features like checksums and session IDs, which limits the ability of test cases to penetrate into the target.

Template fuzzing is only as good as the template used to generated test cases. If the template or templates used do not cover a specific functional scenario, the corresponding part of the target code will not be exercised and lurking vulnerabilities will remain hidden. Use a best effort approach to selecting templates that best represent the intended functionality of the target. Automated harvesting and optimization of templates, or samples, is sometimes referred to as *corpus distillation*[1].

Generational fuzzing, or model-based fuzzing, the fuzzer itself is implemented from the specifications of the protocol or file format being tested. The fuzzer knows every possible message and field and fully understands the protocol rules for interacting with a target. The fuzzer will correctly handle checksums, session IDs, and

1 http://googleonlinesecurity.blogspot.com/2011/08/fuzzing-at-scale.html

other stateful protocol features. A generational fuzzer generates test cases by iterating through its internal protocol model, creating test cases for each field of each message. In general, generational fuzzing finds more vulnerabilities in less time than any other kind of fuzzing.

Generational fuzzing finds more vulnerabilities in less time than any other kind of fuzzing.

Test cases

The numbers indicated in the **Test cases** column are minimum amounts of test cases. Testing must be performed until either the minimum test cases are reached or the minimum time is reached, whichever comes first.

The label **infinite** indicates that the fuzzer should be placed in a mode where it generates test cases indefinitely. In this case, testing should be performed for at least the indicated time.

Time

This is the minimum time, in hours, for fuzzing. Testing must be performed until either the minimum test cases are reached or the minimum time is reached, whichever comes first.

Care must be taken to ensure that the testing performed covers as many features of the tested attack vector as possible, resulting in testing as many code paths as possible. This is especially important when limiting a test run to a certain time. Generational fuzzers, in particular, create test cases by iterating through an internal protocol model. If only 20% of the available test cases can be delivered in the available time, the test cases that anomalize fields toward the middle and end of the model will never be delivered.

In a time-limited test run, ensure that the fuzzer is configured to deliver an even distribution of test cases to the target.

Discussion of test cases and time metrics

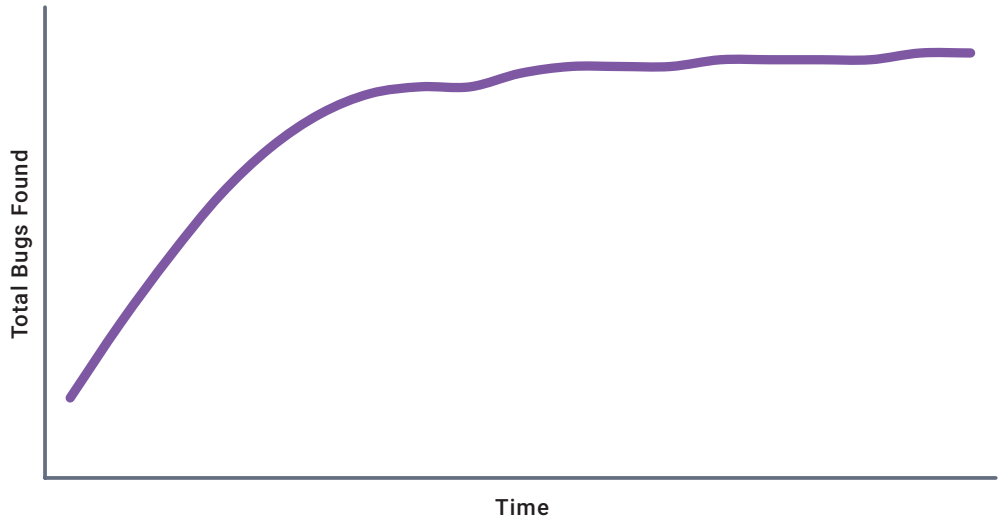
Are test case counts and fuzzing time measurements meaningful requirements? It is possible to achieve minimum test times cheaply by sending test cases very slowly. Likewise, it is possible to achieve minimum test case counts cheaply by sending lower quality test cases.

Anyone willing to invest time and resources to take either of these approaches might as well strive to do as good testing as possible. The ultimate goal is locating and fixing vulnerabilities, not achieving a certain FTMM level.

Likewise, is minimum testing time meaningful, given that test cases might be delivered at very different rates for different targets? Bear in mind that the same difficulties encountered in fuzzing a very slow target will also be encountered by anyone attacking the target. Attackers use fuzzing as tool for locating vulnerabilities—they will have the same challenges as you in fuzzing the target.

The test case counts and minimum testing times in this document are based on years of industry averages.

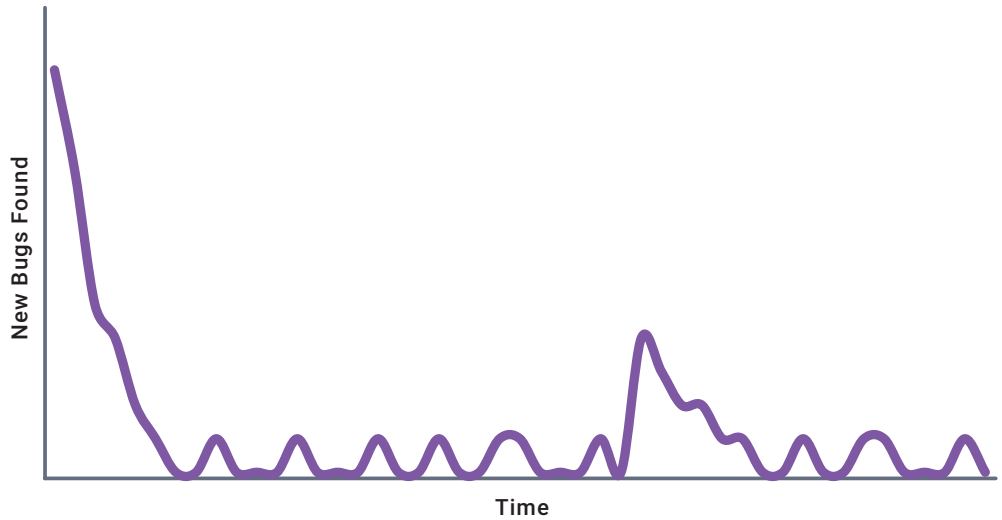
When software is initially fuzzed, many bugs are found. Over time, fewer and fewer new bugs are found. The following graph is typical:



You should keep track of the number of bugs found over time and use it as a measurement of your progress. If you continue to locate new bugs in a mostly linear ascent, consider increasing your testing, regardless of the test case count and testing time required for a particular FTMM level. Your goal should be to reach the crest of the hill, i.e. to get to the part of the graph where new bugs are found at a much slower rate.

You should keep track of the number of bugs found over time and use it as a measurement of your progress.

Equally important is a sustained commitment to fuzz testing. The graph of new bugs found over time typically looks like this:



After the initial rush of bugs, the software stabilizes. Don't assume that it is safe to stop fuzzing at this point! When a new version is written by the developers, a new group of bugs is introduced. If these bugs are not located and fixed, they will accumulate over time as the software continues to change, until the software is just as buggy as before the fuzzing.

Even if a particular component hasn't changed, changes in surrounding components or the environment in which the software runs can result in new bugs being uncovered.

Your initial work with fuzzing might seem like a large effort, but once you crest the hill, the effort will ease off. The rate of new vulnerabilities tapers off, and you will grow accustomed to the tools. Once things are running smoothly, consider automating fuzzing as part of your software testing. Test harness integration is required at Level 5, but often a prudent and helpful step at earlier levels.

Instrumentation

Instrumentation is the method a fuzzer uses for monitoring the target during testing and for collecting telemetrics. This maturity model defines the following instrumentation methods:

Human observation uses human cognitive ability to identify failures. While the fuzzer delivers test cases to the target, a human tester observes the behavior of the target. This can be accomplished by looking at log files or console output for the target, looking at the front panel (if present), or monitoring existing sessions involving the target. Fundamentally, the tester is looking for target behavior that is out of the ordinary, including the failure modes mentioned in section 3.6, Allowed failures. The tester should be familiar with the functionality of the target and be able to differentiate between normal behavior and anomalous behavior. Typical facilities such as log files and other management user interfaces should be used in addition to any available developer tools. Human observation can be effective, but automated instrumentation is recommended and required for higher FTMM levels.

In automated instrumentation, the fuzzer automatically checks on the health of the target during testing, usually after each test case is delivered. One simple and effective method is valid case instrumentation, in which every test case is followed by a valid message from the fuzzer. If the target responds with a valid response, the fuzzer considers the target healthy and continues by sending the next test case.

Developer tooling makes use of advanced utilities to examine memory usage, profile execution paths, analyze test coverage, and perform other advanced monitoring and measurement in the target. Failed assertions, automatic recovery via try-catch mechanisms, and other subtle signs of distress can be observed when running target software with developer tooling. These conditions, which can be dangerous, are not usually otherwise visible. Sometimes developer tooling can be hard to achieve, especially when testing systems built by others where access to development tools and to the underlying operating systems may not be feasible.

Apart from discovering vulnerabilities, developer tooling is an excellent way to track target behavior, highlight performance bottlenecks and observe other undesirable behavior. In general, developer tooling makes it easier to detect something going wrong. Additionally, tooling is useful for pinpointing vulnerabilities for fixes once the vulnerabilities have been located.

Note that appropriate developer tooling might not be available for buyers. For example, on an embedded device, a buyer might be simply unable to gain access to the device or find appropriate tools for analysis. If the tools are not available, buyers will be unable to achieve FTMM Level 4 or Level 5 and should, instead, ask their builder vendors to achieve these levels if necessary.

Allowed failures

This column describes the types of failures that are allowed to remain after testing. One of the challenges of fuzzing is that software can fail in many different ways:

- Crashes
- Kernel panics
- Unhandled exceptions
- Assertion failures
- Busy loops
- Resource consumption

Resource consumption usually refers to processing power, available memory, and available persistent storage, but the important resources are ultimately determined by the target and its environment. Monitoring resource consumption is a matter of defining baseline and critical threshold values for resource consumption, documenting these values in the test plan, and then comparing the resource values during testing to the defined thresholds.

Resource monitoring can be as simple as a human observing the output of the top utility on a Linux-based target to automated retrieval of SNMP values for targets that support SNMP.

When excessive resource consumption is non-permanent, meaning the target returns to normal behavior after fuzzing, the resource consumption is not considered a failure in context of this framework. For example, a small embedded system can easily be overwhelmed by volume of network traffic generated by a fuzzer. If the device resumes normal operation when the testing stops, then it is not a failure. It might be a performance issue, but that's outside the scope of fuzz testing.

Some failures are a byproduct of both the volume of fuzz test cases as well as unusual conditions within the target caused by anomalous input. When failures are observed, but cannot easily be reproduced, they are transient.

Assertion failures occur when built-in software checks fail. This might result in a warning message in a log file but does not necessarily interrupt the function of the target.

See the table above for the allowed failure modes in each maturity level. Even when failures are allowed, they must be noted in the documentation.

Test harness integration

Builder organizations will initially run fuzz testing tools manually. Over time, however, usage will naturally migrate to automatic fuzz testing as part of an overall automated testing process. This integration is a sign of maturity in an organization's use of fuzzing.

With the required times in higher FTMM levels, test harness integration and test automation are crucial. Such automation can ease the transition to parallel testing that is likely necessary to achieve higher FTMM levels.

This column does not apply when fuzzing is being used as a verification and validation tool.

Processes

Fuzzing can be performed on any available attack vector. Testers with a basic knowledge of the target will know about at least some of the available attack vectors.

A comprehensive analysis of the attack surface of the target, in its intended configuration, is required for rigorous testing. The end result of attack surface analysis is a list of all attack vectors for the target. Note that the attack surface consists of only those attack vectors that are active in the used configuration. A target might have additional capabilities that would expose additional attack vectors, but if they are not enabled in the used configuration, they do not need to be fuzzed to achieve a specific FTMM level for the target in this configuration.

Component analysis can be an effective precursor to fuzzing. Component analysis allows an organization to flush out known vulnerabilities inherited from third-party software components, while fuzzing follows up as an effective method for locating unknown vulnerabilities. Methods exist to enumerate components from both source code and binary packages and to cross reference these with known vulnerabilities.

Fuzzing can be performed on any available attack vector.

Documentation

A fuzzing report should include the following information:

- A summary table providing an overview of testing, including the following information for each attack vector:
  - Fuzzing tool and version
  - Test run verdict
  - Instrumentation method
  - Number of test cases
  - Testing time
  - Date of test run
  - Notes
- For each attack vector, detailed results must be submitted. These must be generated from the fuzzing tool and include the following:
  - Data for each test case delivered to the target, such as test case verdict, time, duration, and amount of output and input.
  - The log of the fuzzer.

Documentation is a crucial component of effective, repeatable fuzzing. The test plan can be adapted from the attack surface analysis and should include information about the tools and techniques that will be used for testing. The baseline test configuration should include information about the test bed, target configuration, and fuzzer configuration.

Other requirements

Recall that fuzzing is an infinite space problem—each fuzzer chooses some subset of the infinite number of malformed inputs that can be delivered to target software. Even multiple fuzzers of the same type will choose different sets of test cases. For the most comprehensive testing, use as many fuzzers as possible.

Practical considerations

This section outlines some of the common challenges related to implementing fuzzing and meeting FTMM requirements.

Minimizing attack surface

The first and most obvious action to help meet FTMM requirements is to minimize the attack surface of your target. More attack vectors equals more time and money spent testing. Services should be removed if they are not necessary to the functionality of the target.

Scope

Builders often wonder if they are responsible for code they did not write. The answer is an emphatic "Yes!". As a builder, you are responsible for everything that goes into your product, regardless of its origin. If your customers have catastrophic security breaches because of third-party code in your product, a response of "We didn't write that code" will be little comfort. Your name is on the product; your customers hold you responsible when it fails.

If you perform Level 1 or higher fuzzing on your own code but your third-party code is at Level 0, your overall maturity level is 1.

If you perform Level 1 or higher fuzzing on your own code but your third-party code is at Level 0, your overall maturity level is 1.

The only possible shortcut would be if the code you acquire has already been fuzzed. In this case, you could accept the incoming code and its FTMM level and just fuzz the code you have written yourself. However, note that the fuzzing performed by your vendor is in a different configuration and a different environment than the fuzzing you would perform on the complete product. See section 4.5, Fidelity. Even if your code is fuzzed to Level 5, and your vendor's code is fuzzed to Level 5, the integrated system might still yield bad behavior. When integrating third-party code that has been fuzzed, at least perform a smoke test on the applicable interfaces.

Viable Interfaces

Some interfaces are extremely difficult to fuzz, either because of a lack of fuzzing tools or because test cases are hard to deliver.

For example, some radio baseband protocols cannot be fuzzed without specialized hardware. Likewise, serial wire protocols can make test case delivery a challenge. Some proprietary protocols have no existing generational fuzzers.

Another difficult attack vector is user input: without an army of robots, fuzzing a user interface is a slow, manual process. Luckily, many user interface automation solutions exist.

Attack vectors that are hard for you to fuzz will also be hard for attackers to fuzz. However, bear in mind that attackers might have better funding than you, in which case they might be able to obtain tools that are out of your reach. Alternately, attackers might figure out a more efficient way to fuzz a particular attack vector. Use your best judgment about the availability of fuzzers; if you are confident that some attack vectors are impractical for attackers, make a note in your test plans that they are deliberately excluded.

Another factor that determines the viability of attack vectors is proximity. Attack vectors on networks are easy to reach from afar, while other vectors such as keypads, serial interfaces, or USB interfaces require physical access to the target.

We expect organizations and regulators that adopt this maturity model to provide specific guidelines for selecting appropriate attack vectors.

Attack vector complexity

Clearly, attack vectors vary widely in their complexity. For a simple network protocol, 100,000 generational fuzz test cases could provide extremely thorough testing, while for a more complex protocol, the same number of test cases might just scratch the surface.

This implies that a target's implementation of a simple attack vector tested to Level 3 might be more robust overall than the implementation of a complex attack vector tested to the same level.

A logical solution would be to use the attack vector complexity in calculating the number of required test cases or the minimum testing time. However, given the difficulties of measuring complexity in attack vectors, this document uses test cases and time as measureable and easy-to-compare metrics.

Fidelity

Fuzzing is black box testing—all that is needed is a running target. That being said, the testing performed is specific to the target you use. Testing results can have sensitivity to how the target was built, how it is configured, and the environment in which it lives.

The same source code compiled with different build flags, or built for two different architectures, can exhibit different failures. Similarly, software configuration affects the visibility of vulnerabilities.

Ideally, fuzzing should happen on the same binaries that are used in production, in the same configuration, in the same environment. This ideal is unattainable. The goal is to keep the configuration and environment as close as possible to a production setup.

Fuzz testing on debug builds, or builds with additional logging enabled, often makes finding and tracing vulnerabilities easier. However, these changes can alter the target's behavior, either exposing vulnerabilities that would not be present in a production build, or hiding vulnerabilities that would be present in a production build.

Note also that testing performed by a builder will likely use a different configuration and environment from buyers. Strictly speaking, even if a builder has attained a certain maturity level, cautious buyers should do their own fuzzing for verification and validation. A target fuzzed to an advanced FTMM level by a builder might still exhibit vulnerabilities in a buyer's environment. However, retesting is often impractical, and buyers will probably rely on the builder's achieved FTMM level as an assurance of quality.

Smaller devices present their own challenges. The limited resources of some embedded devices means that sending large volumes of test cases should be done by running the target code "off-device" in an emulator on a desktop computer. This is a deliberate tradeoff—the target being tested is in a significantly different environment than production code, but the emulation enables significantly more testing. Any testing performed off-device should be noted as such in the test report documentation. If most testing is done off-device, smoke testing should be performed on the real device.

Finally, note that upper levels of FTMM call for running target software with developer tooling. While this enables catching subtle failures that would otherwise be missed, it changes how the target software behaves. Ideally, testing should be performed both with and without the developer tooling to catch any vulnerabilities that might be masked by the disruptive effects of the tools.

Desirable qualities in fuzzers

This document does not specify what features a fuzzer must have. A random fuzzer can be written in five lines of code, while a sophisticated generational fuzzer could be the result of years of research and development and be packed with features.

This section is here to help you understand the qualities and features that allow you to execute the best possible tests in shortest amount of time. Understanding what makes a great fuzzer is important in building an effective arsenal of test tools.

Specification coverage

The goal of fuzzing is to find vulnerabilities. Rigorously exercising as many code pathways as possible is a good method. Having great specification coverage gives a fuzzer superior code coverage.

Feature coverage

The fuzzer should be able to exercise all the features described by the protocol specifications. For example, a SIP fuzzer should have a way to initiate calls, redirect them, and hang up.

State coverage

The best target penetration will be achieved when the fuzzer is able to correctly exercise protocol states as described by the protocol specification. Generational fuzzers are able to do this because they know the rules about how protocol messages are exchanged. This means that, in the eyes of a target, a generational fuzzer is highly believable as a protocol endpoint. Furthermore, a generational fuzzer is capable of moving the target through valid protocol states during testing.

Element and structure coverage

Protocol specifications describe messages and their elements. A great fuzzer knows the structure of every message and element.

Optimizing testing for supported features

Many network protocols have optional components. In order to test efficiently, the fuzzer should discover the features that are supported by the target and optimize the testing so that no time is wasted testing features that are not present.

Optimizing test coverage within available test time

As noted previously, when running time-limited test runs, care must be taken to use an even distribution of the available test case material. If you will time-limit test runs, make sure your fuzzer is capable of randomizing the test case execution order.

Controlling the amount of test cases

Fuzzing is an infinite space problem, so it is important to be able to control the amount of test cases generated by the fuzzer. For example, you should be able to run various subsets of the available test cases so that you can perform smoke testing or reduced test runs.

Diligent maintenance

Specifications and interfaces change over time. A good fuzzer is actively maintained to accommodate for these changes.

Fuzzing specifics

Good fuzzers are smart about generating malformed inputs for software. This section describes several specific areas where your fuzzer should shine.

Type Length Value (TLV) Fields

Various anomalies in TLV structures are obvious, such as invalid types, incorrect lengths, and invalid values. In addition, the fuzzer should also repeat TLV fields and create test cases where the TLV fields are out of order.

CRCs, MACs, and lengths

CRCs and MACs are “fingerprints” of a protocol message or file. When software receives input, it can use CRCs and MACs as an attempt to verify that information has arrived with being damaged in transit. If CRCs or MACs do not match up, the target is likely to drop the incoming input without examining the rest of the message. By correctly calculating CRCs and MACs for anomalies elsewhere in test cases, a fuzzer helps ensure that test cases will penetrate further into the target, increasing the likelihood of finding bugs.

A fuzzer that correctly sets a length field generates test cases that will be more effective against target software.

Likewise, message length fields (such as Content-Length in HTTP) serve as a simpler check of message integrity. A fuzzer that correctly sets a length field generates test cases that will be more effective against target software. Keep in mind that some test cases specifically target the length field itself, in which case the length field will have malformed, anomalous values.

Conversations with multiple messages

In some network protocols (e.g. TLS or SIP), interactions between endpoints consist of conversations with multiple message types exchanged. A good fuzzer should know the rules of these conversations and systematically break those rules. Examples include test cases that repeat messages, omit messages, and send messages out of order.

Cryptography support

Some protocols are delivered over encrypted network connections. Your fuzzer should implement encryption according to the protocol specification and fuzz both the data inside the encryption as well as the encryption headers.

Anomaly type controls

In certain situations you will need to generate specific types of test cases to exercise specific portions of your target. In this case, it is very useful if your fuzzer has controls that allow you to adjust the types of anomalies that are used in generating the test cases.

Conclusion

All software contains vulnerabilities. Like death and taxes, software vulnerabilities are inescapable. You can significantly reduce your risk by adopting a higher maturity level, which will make it harder and more time consuming for adversaries to find exploitable vulnerabilities.

Finding and fixing more vulnerabilities increases the overall security and robustness of your target and reduces your risk profile.

This maturity model gives software builders and buyers a standard scale for describing fuzz testing performed on target software and the associated risks.

References

"SDL Process: Verification." Microsoft. 27 Oct. 2013.
<http://www.microsoft.com/security/sdl/process/verification.aspx>

"CSDL Process." Cisco. 27 Oct. 2013.
<http://www.cisco.com/web/about/security/cspo/csdl/process.html#~tab-4>

"Adobe Secure Product Lifecycle." Adobe.
<http://www.adobe.com/security/pdfs/privacysecurity_ds.pdf>

Knudsen, Jonathan. "Make Software Better with Fuzzing." ISSA Journal, July 2013.
<http://www.codenomicon.com/news/editorial/Make%20Software%20Better%20with%20Fuzzing.pdf>

Knudsen, Jonathan. "That Warm, Fuzzy Feeling...and How You Can Get It." Professional Tester, April 2012.
<http://www.codenomicon.com/news/editorial/professional_tester_0412_that_warm_fuzzy_feeling.pdf>

Takanen, Ari, et al. Fuzzing for Software Security Testing and Quality Assurance. Artech House, 2008.

Appendices

Example attack surface analysis, test plan, and report

Note that this same type of document can be used for three purposes. When the attack surface analysis is performed, use the table to list out the attack vectors of the target. For test planning, specify the fuzzing tool you will use for each attack vector. After testing is complete, fill in the information about the test runs and use the resulting document as a high-level report.

FTMM v1.0 Attack Surface Analysis, Testplan and high level test report

Test target identificaton	
Test target	Secure Networks Embedded Router (SNER)
Software version	3.78.1337revB (2014.12.3.332) (firmware ver 12.2T.893)
Address / Location	IPv4: 172.12.24.44, IPv6: 2001:café:baba::affa:0001, Test lab C1, Building A

Attack Surface analysis, & Test plan

Test n:o	Port	Layer	Protocol	Test Suite	Test suite version	VERDICT (*	VERDICT ADJUSTED	Number of executed test cases	Test run time	Date executed	Notes
1	n/a	IP	IPv4	IPv4	3.6.9						
	n/a	IP	IPv6	IPv6	4.0.7						
	n/a	IP	ICMP	ICMP	2.4.2						
	n/a	IP	ICMP6	ICMP6	3.1						
	n/a	IP	TCP	TCP-Server	4.0.6						Use HTTP as a payload
	n/a	IP	UDP	IPv4	3.6.9						
	80	TCP	HTTP	HTTP Server	5.1						username: basic, password secret
2	179	TCP	BGP4	BGP4 Server	2.3						target ASN 65002
3	443	TCP	TLS/SSL	TLS Server, TLS1.2 Server	5.3.1						Target requires client certificate authentication
4	443	TCP	X.509	X.509v3	2.3.1						For x.509 client certificates carried inside TLS/SSL.
5	639	TCP	MSDP	MSDP Server	1.1.2						
6	50100	TCP	unknown	TCF	4.0.1						Use Traffic Capture Fuzzer for unknown protocol
7	53	UDP	DNS	DNS Client	4.2						DNS Client on SNER
8	68	UDP	DHCP	DHCP Client	3.3						DHCP Client on SNER
	521	UDP	RIP								Do not test. Will not be enabled in final release
9	521	UDP	RIPng								Do not test. Will not be enabled in final release
10	5060	UDP	SIP	SIP-UAS	6.2.1						
11	5060	UDP	SigComp	SigComp Server	1.0.7						Signaling compression with SIP
12	dynamic	UDP	RTP	RTP/RTCP	3.0.1						RTP port is determined by SIP signaling
13	89	IPv6	OSPFv3	OSPFv3	2.0.2						

NOTES:
(* Either PASS, FAIL, SKIP or INCONCLUSIVE. FAIL meaning that 1 or more bugs were noted during test run

General notes and observations

Examples of test documentation to preserve

In order to support reproduction of test results, it is important to consider what kind of documentation should be maintained and preserved after the tests have been executed. This section contains recommendations on types of documents to preserve. These documents should be coupled with the used test plans and high level test reports, and maintained in a repository from which they can be retrieved as needed.

Target configuration

Fuzz testing is very dependent on the active configuration of the test target. For example, if the active configuration doesn't enable a specific service, or a feature within the service, then that service or feature cannot be tested. From a standpoint of results validation, and fault reproduction, it is essential to maintain a database or library of exact test target software (or firmware) versions or builds and configurations used when test runs were performed.

Examples of configurations are ASCII-based configuration files and binary files containing the active configuration.

Fuzzer configurations

Maintaining fuzzer configurations guarantees that performed tests can be executed again at a later time. Modern fuzzers have near infinite ways of generating test cases based on various parameters, underlining the need to maintain and store the used configurations with other test run details.

Test bed documentation

While a test bed should have minimal impact on executed tests, its role is nevertheless critical. Especially with more complex systems, the target system may require other components to be present before it functions correctly and is ready for testing. Interactions with these devices are likely to affect the test outcome. Thus, good test bed (or test ecosystem) documentation should take into account variables such as the following:

- IP addresses used in the system
- Network topography used
- Link speeds
- Other devices (with versions) used in the network

Test logs

Reproducing a past test run may be impossible because the test bed no longer exists or is reserved for other use. In these types of situations, it is imperative that all the available logs gathered during the test run are preserved for later analysis. Initial analysis commonly misses some key piece of information which must be extracted later from test run logs. To this end, all the logging produced by the fuzzer, the test target, and the ecosystem should be preserved.

For example, on Linux-based targets, such logging might include */var/log/messages* or the syslog facility, while Windows-based targets have the Windows Event Log. Other examples would be SNMP traps emitted by the target or application-specific log files.

While collecting logs, the volume of logs can be daunting. Especially if full debug logging is enabled, generated log files from fuzz testing can be tens of gigabytes in size. Instead of storing all this information, consider alternatives such as saving only logs of reproduction test runs for specific faults. Similarly, saving traffic captures and traces of entire test runs is often infeasible; such captures and traces should be limited to reproduction of faulty behavior.

THE SYNOPSYS DIFFERENCE

Synopsys offers the most comprehensive solution for integrating security and quality into your SDLC and supply chain. Whether you're well-versed in software security or just starting out, we provide the tools you need to ensure the integrity of the applications that power your business. Our holistic approach to software security combines best-in-breed products, industry-leading experts, and a broad portfolio of managed and professional services that work together to improve the accuracy of findings, speed up the delivery of results, and provide solutions for addressing unique application security challenges. We don't stop when the test is over. Our experts also provide remediation guidance, program design services, and training that empower you to build and maintain secure software.

For more information go to www.synopsys.com/software

SYNOPSYS®
185 Berry Street, Suite 6500
San Francisco, CA 94107 USA

U.S. Sales: (800) 873-8193
International Sales: +1 (415) 321-5237
Email: software-integrity-sales@synopsys.com