What is Fuzzing: The Poet, the Courier, and the Oracle

Fuzzing is well established as an excellent technique for locating vulnerabilities in software. The basic premise is to deliver intentionally malformed input to target software and detect failure. A complete fuzzer has three components. A poet creates the malformed inputs or test cases. A courier delivers test cases to the target software. Finally, an oracle detects if a failure has occurred in the target. Fuzzing is a crucial tool in software vulnerability management, both for organizations that build software as well as organizations that use software.

January 7, 2015

Jonathan Knudsen
Principal Security Engineer

Mikko Varpiola
Founder

Table of Contents

1. Fuzzing in the Context of Software Testing

Fuzz testing, or *fuzzing*, is a type of software testing in which deliberately malformed or unexpected inputs are delivered to target software to see if failure occurs.

In this paper, we use *software* to mean anything that is compiled from source code into executable code that runs on some sort of processor, including operating systems, desktop applications, server applications, mobile applications, embedded system firmware, systems on a chip, and more.

When a piece of software fails accidentally due to unexpected or malformed input, it is a *robustness* problem.

In addition, a diverse cast of miscreants actively seeks to make software fail by delivering unexpected or malformed inputs. When software fails due to deliberate attack, it is a *security* problem.

A software failure that causes harm or death to humans is a *safety* problem.

Robustness, security, and safety are three faces of the same hobgoblin, software *bugs*. A bug is a mistake made by a developer; under the right conditions, the bug is triggered and the software does something it was not supposed to do. Improving robustness, security, and safety is a matter of finding and fixing bugs.

1.1. Positive and Negative Testing

Historically, software testing has focused on functionality. Does the software work the way it's supposed to work? In functional testing, a type of *positive testing*, test developers create code and frameworks that deliver valid inputs to the target software and check for the correct output. For example, if we press the big red button (deliver an input), does the software turn on the city's power grid (correct output)?

In a traditional software development methodology, the software design is a list of requirements for the target software. The test development team has a fairly straightforward task of translating the design requirements into test cases to verify that the software is performing as described in the specification.

Functional testing is certainly important—the target software must behave as expected when presented with valid inputs. However, software that is only subjected to positive testing will fail easily when released into a chaotic and hostile world.

The real world is a mess. It is full of unexpected conditions and badly formed inputs. Software must be able to deal with other software and people who will supply poorly formed inputs, perform actions in unexpected order, and generally misuse the software. *Negative testing* is the process of sending incorrect or unexpected inputs to software and checking for failure.

Be aware that different negative test tools will produce different results for the same test target. Each tool works differently and will test different kinds of badly formed inputs on the target software.

1.2. Software Vulnerabilities

Bugs are also known as *code vulnerabilities*. In the world of software, vulnerabilities come in three flavors:

1. *Design vulnerabilities* are problems with the design of the software itself. For example, a banking website that does not require users to authenticate has a serious design vulnerability. In general, design vulnerabilities must be hunted and killed by humans—automated tools simply do not exist at this level.
2. *Configuration vulnerabilities* occur when the setup of a piece of software has exposed a vulnerability. For example, deploying a database with default (factory-installed) administration credentials is a configuration vulnerability. While there are some automated tools that can assist in locating configuration vulnerabilities, much of the seek-and-destroy work must be performed by humans.
3. *Code vulnerabilities* are bugs. Positive testing, with manually coded test cases, can be used to find and fix bugs related to functionality. Negative testing, which can be heavily

automated, can be used to improve the robustness and security of the software.

In addition, software vulnerabilities are *unknown*, *zero day*, or *known*.

1. An *unknown vulnerability* is dormant. It has not been discovered by anyone.
2. A *zero day* vulnerability has been unveiled by one person or a team or organization. A zero day vulnerability is not published. The builder and users of the affected software are most likely unaware of the vulnerability. No fixes or countermeasures are available.
3. A *known* vulnerability is published. Responsible vendors release new versions or patches for their software to address known vulnerabilities.

While fuzzing is typically used for locating unknown code vulnerabilities, it can also trigger vulnerabilities caused by poor design or configuration.

1.3. Black, White, and Gray Box Testing

In *black box* testing, the test tool does not have any knowledge of the internals of the target. The tool interacts with the target solely through external interfaces.

By contrast, a *white box* tool makes use of the target's source code to search for vulnerabilities. White box testing encompasses static techniques, such as source code scanning as well as dynamic testing, in which the source code has been instrumented and rebuilt for better target monitoring.

Gray box tools combine black box and white box techniques. These tools interact with the target through its external interfaces, but also make use of the source code for additional insight.

Fuzzing can be black box or gray box testing. This flexibility makes fuzzing an extremely useful tool for testing software, regardless of the availability of source code or detailed internal information. As a black box technique, fuzzing is useful to anyone who wants to understand the real life robustness and reliability of the systems they are operating or planning to deploy. It also is the reason why fuzzing is the number one technique used by black hat operatives and hackers to find software vulnerabilities.

Even without source code, the ability to more closely monitor the vitals of the target software improves the quality of the testing. Log files, process information, and resource usage provide valuable information that can be used during fuzzing to understand how the anomalous inputs are affecting the target system.

If the source code is available, debugger tooling and other instrumentation can be used to make fuzzing a gray box technique. This improves the accuracy of the tests and allows more rapid resolution of located defects.

1.4. Static and Dynamic Testing

Vulnerabilities are pursued using both *static* and *dynamic* testing techniques. A mature development process should leverage a variety of static and dynamic techniques in order to drive risk down to the desired level. This section highlights common static and dynamic software testing techniques, one of which is fuzzing.

Static techniques can be used without actually running the target software. They include the following:

1. *Source code reviews* are carried out by developers. They read through the source code and look for unknown vulnerabilities. While this can be effective, it is very slow and success depends heavily upon the skill of the reviewers.
2. *Automated source code analysis* tools scan the target source code and report on patterns of programming that might be vulnerabilities. A human developer must review the results of this scan.
3. *Static binary analysis* tools scan compiled (executable) code and report on contained libraries and associated known vulnerabilities.

Dynamic test techniques are performed on the target software as it is running and include the following:

1. *Load testing* delivers large and numerous inputs to the target software to see if it fails due to heavy volume.
2. *Interoperability testing* validates that two implementations are able to converse using a specified protocol, language, or notation.
3. *Conformance testing* validates that tested system and its behavior conforms to relevant specifications.
4. *Fuzzing* delivers anomalous inputs to software to see if failure occurs. It is an excellent method for locating unknown vulnerabilities.

1.5. What is Fuzzing?

Fuzzing is the process of sending intentionally malformed inputs to a piece of software to see if it fails. Each malformed input is a *test case*. Failure indicates a found bug, which can then be fixed to improve the robustness and security of the target software.

A fuzzer is a piece of software that tests a piece of target software. A proper fuzzer consists of three components:

1. The *poet* is responsible for creating the test cases. The poet is also known as the test case generator, test case engine, or anomalizer.
2. The *courier* sends the test cases to the target. The courier is also known as the injector, delivery mechanism, or test driver.
3. The *oracle* determines if the target has failed.

Not all fuzzers implement all three parts, but to harness the power of automated fuzzing, all parts should be present.

When using fuzzers to improve robustness and security, the end goal is not just finding bugs, but fixing bugs. A useful fuzzer must keep records, produce actionable reports, and provide a smooth remediation process to reproduce failures so that they can be fixed.

1.6. Zooming Out: Vulnerability Management

A fuzzer will not solve all of your problems. It must be part of an arsenal of tools and part of a process for software vulnerability management. Other tools that you might also use for software vulnerability management are as follows:

- Manual security reviews
- Reverse engineering
- Static binary code analysis
- Known vulnerability scanning
- Patch management tools
- Fuzz testing

1.7. Zooming All the Way Out: Risk Management

Software vulnerability management is part of a larger picture - *risk management*. An organization seeking to lower, or at least understand, its overall risk will use software vulnerability management in conjunction with other risk management techniques. Fuzzing is a powerful technique for assessing the robustness and security of software, which is directly related to risk.

Now that you understand who uses fuzzing, how fuzzing relates to other software testing techniques, and where fuzzing is used in the world of vulnerability management, we will move ahead by discussing techniques and algorithms used in fuzzing.

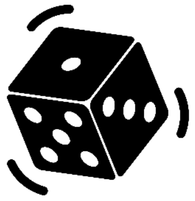
2. The Poet

Fuzzing is an infinite space problem. For any piece of software, the set of invalid inputs is unbounded. An effective poet must be clever enough to craft test cases that are most likely to trigger bugs in the target software. In essence, this comes down to having a poet that creates test cases that are close to what the target expects, but *malformed* in some way.

The method of generating test cases has a profound effect on the quality of the test case material.

2.1. Random

The simplest yet least effective fuzzing method is random fuzzing. The poet simply uses *random* data as test cases.

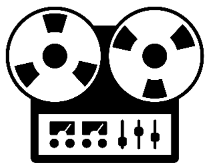


Random fuzzing is usually ineffective because the test cases are nothing like valid input. The target examines and quickly rejects the test cases. For the most part, the test cases fail to penetrate into the target code.

In theory, a random poet will eventually produce a test case that resembles valid input. However, even with relatively short valid inputs, the probabilities of producing a mostly correct test case are very low, which makes the time required to wait for such a test case very long.

2.2. Template

A *template* poet introduces anomalies into valid inputs to create test cases. In general, template test cases are much more effective than random test cases because they are mostly correct. The target software will process test cases and the anomalies exercise the target's ability to handle unexpected or malformed inputs in a safe, robust, and secure manner.



Getting started with template fuzzing is usually quick, as suitable template files or network captures are often readily available. Also, a template poet is easily adjustable for target software that uses unexpected or non-standard elements in its input.

However, template poets have significant limitations. First, for protocols that include some kind of integrity validation, like message checksums, testing effectiveness will be limited. The template poet will introduce anomalies, but does not know to update checksums. The template poet can create millions of test cases for the target, but if the integrity checks don't pass, the target software will not bother trying to parse the rest of the messages and the corresponding parsing code remains untested.

Second, for protocols that include stateful features, a template poet has the same kind of problem. The template poet does not understand the semantics of the protocol and will blindly replay its anomalized valid messages. It cannot correctly set stateful features like session identifiers, so its effectiveness in testing such protocols is limited.

Finally, for protocols that are partially or completely encrypted, a template poet cannot be used directly. If unencrypted valid messages can be obtained, the template poet can create test cases that can be subsequently piped through an external encryption mechanism for delivery to the target.

Template fuzzing is often limited by the availability of good templates. The quality of the used templates dictates the quality of the results.

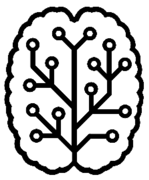
Modern template fuzzers can overcome many of the aforementioned limitations. For example, Codenomicon's Traffic Capture Fuzzer (TCF) is a template fuzzer that creates test cases based on a packet capture file (pcap). TCF consults a traffic analyzer to see if the packet capture file can be dissected. If it can, TCF uses the information supplied by the protocol dissector, specifically message structure and field boundary information, to create a better-focused set of test cases resulting in more effective testing. In addition, TCF is able to correctly calculate length and checksum fields for many protocols, greatly increasing its effectiveness.

Codenomicon's Defensics Universal Fuzzer (DUF) is another example of an enhanced template fuzzer. DUF uses a collection of valid files (a *corpus*) as the basis of test cases. DUF analyzes the valid files to infer their structure and create high-quality test cases for the target.

Corpus distillation is a method for overcoming some of the limitations of template-based fuzzing related to quality of the templates. Corpus distillation finds and selects the templates that will be used to create test cases. The basic technique is to select valid cases that best represent the protocol or file format overall. For example, in a protocol with multiple message types, the templates selected should include all message types. For a file format with optional parts, good templates should include all optional features.

2.3. Generational

A *generational* poet understands the protocol, file format, or API that it tests. This means it knows every possible structure and message type, all the fields in every message, and rules about how messages are exchanged. Because the generational poet knows all the rules, it can systematically break all the rules.



Furthermore, because the generational fuzzer has full knowledge of the protocol or input type, it can correctly handle the things that confound a template poet. A generational fuzzer can keep track of stateful features like session identifiers and it can correctly set values like checksums that can be a limiting factor for other fuzzing techniques.

A generational poet creates high-quality test material that looks legitimate to the target. For protocols with multiple-message conversations, this allows the generational fuzzer to exchange several valid messages with a target, driving it to a certain state, before delivering the anomalized test case.

Codenomicon's Defensics fuzzing platform has more than 250 generational fuzzers for a wide variety of network protocols and file formats.

2.4. Evolutionary

An *evolutionary* poet uses feedback about the target's behavior to influence how subsequent test cases are created. Some measurement related to the target's response to test cases is used to score test cases that have already been sent to the target. The poet creates more test cases based on the highest-scoring previous test cases.

A "pure" evolutionary fuzzer is a black box fuzzer. Supplied with a destination address and port, the evolutionary fuzzer sends increasingly relevant inputs to the target, based solely on the target's response. Defensics uses evolutionary methods to shape its test material. An interoperability scan discovers which features are implemented in a target, then configures the test suite so that only the implemented features are tested.

3. The Courier

The *courier* is responsible for delivering the test cases created by the poet to the target software. Fuzzing encompasses a variety of disciplines, each with their own challenges.

3.1. Network Protocol Fuzzing

One common application for fuzzing is network protocol testing. A protocol is a set of rules for how different pieces of software communicate over a network. The code that interprets protocol messages is an attack vector. Fuzzing is an excellent technique for locating unknown vulnerabilities in protocol-handling code.

Testing network software is further divided into fuzzing different roles and types of components. Many protocols include concepts of *client* and *server*, where the client initiates a connection and the server responds.

In *server* testing, the job of the courier is straightforward. The target software listens for incoming connections. All the courier has to do is make a connection to the target server and send a test case.

Client testing is often more complicated. The courier must act like the server, listening for incoming connections. Each time the target client makes a connection to the courier, the courier responds with a test case. Usually, the client needs to be encouraged to repeatedly make connections to the courier so that the courier can continue to deliver test cases.

End to end or *pass-through* testing is another variety. Some network components are not directly addressable, but they do examine network traffic as it passes through. A good example is a firewall with network address translation (NAT) and application layer gateways (ALG) to support voice over IP (VoIP) interactions. To test such a component, the fuzzer is placed on one side and some terminating software is placed on the far side. The fuzzer courier delivers test cases through the target to the terminator. The terminator needs to respond appropriately to the courier and needs to be robust enough to handle the test case material without failing.

3.2. File Fuzzing

In *file fuzzing*, intentionally malformed files are delivered to a piece of software. The courier's job here is hard to define, as different pieces of software do not consume files in any standardized way. Sometimes the best method is to write a custom wrapper or code that facilitates effectively feeding the test cases to the target.

Codenomicon's Defensics file fuzzers offer a variety of open-ended delivery options. Test cases can be written to the file system for later delivery, they can be sent over network connections, they can be delivered to specific commands, or they can be served up by a built-in HTTP Server.

3.3. API Fuzzing

API fuzzing is a different animal in which the various methods or functions of an Application Programming Interface (API) are tested to see how they respond to malformed input. The courier in this case must create source code from the test cases, compile (if appropriate) and run the code, and then see if a failure occurs.

Note that remote procedure call APIs such as DCERPC, SunRPC, and Java RMI fall under network protocol fuzzing. Similarly, XML/SOAP and more modern RESTful/JSON based remote APIs should be considered protocol fuzzing.

3.4. User Interface Fuzzing

Many pieces of software offer a user interface (UI) so that humans can see information and provide input. Input can be provided using a keyboard, a keypad, a mouse, or some other method. UI fuzzing is the process of providing unexpected or malformed inputs to the UI. For example, in an application on a traditional desktop, a fuzzer might deliver mouse clicks on opposite sides of the screen at nearly the same time.

Most modern operating systems and programming environments provide a programmatic way to deliver input events so that such testing can be accomplished entirely in software. However, for some smaller devices, the only way to fuzz user input is to use a mechanical device capable of pressing buttons.

4. The Oracle

The *oracle* determines whether a test case passes or fails. It checks the target to see if a failure has occurred. Knowing when a failure has occurred is crucial to the success of fuzzing. It doesn't do any good to cause failure in your target software if you can't make it fail reproducibly so that it can be fixed.

This section describes several approaches to the oracle challenge. The good news is that multiple approaches can be combined. Using multiple methods to check for target failure increases the likelihood of detecting failures when they occur.

4.1. Types of Failures

Software fails when it behaves in a way its creators did not intend or anticipate. In traditionally applied fuzzing, failure modes come in four categories:

- Crashes
- Endless loops
- Resource leaks or shortages
- Unexpected behavior

These failure modes vary based on type of the system or software being tested, the underlying operating system, and more. A crash might be just a crash, or it might lead to a denial of service of the target, degraded performance, information leakage, security compromise, or something else. The consequences depend on the purpose and function of the software, where and when it is operated, and so on.

The job of the oracle is to detect failure. With such a wide range of failure modes, this is not an easy job!

4.2. Traditional Oracles

The field of fuzzing continues evolving rapidly and one current area of innovation is the oracle. This section describes "traditional" approaches to monitoring a target during fuzz testing.

4.2.1. Eyeballs

Even in an age of heavily automated testing, human observation still has high value and should not be underestimated. A human who understands the target software can observe its functionality and monitor vital signs such as log files and resource usage, noticing things and drawing conclusions in a way that is very hard to automate. Of course, human observation is expensive and does not scale well to dozens of targets or weeks of testing.

For test setup and initial testing, human observation is an excellent oracle technique.

4.2.2. Valid Case or Functional

A *valid* case oracle checks the health of the target by sending valid input and looking for a valid and timely functional response. This can be done after the courier delivers each test case. If a test case causes a failure that makes the target unable to respond to valid input, the oracle will find out right away.

A valid case oracle has the following advantages:

- It is simple and easy to automate.
- It is a black box technique; it can be used without knowing anything about the source code or internal workings of the target.
- It is focused. For example, when fuzzing HTTP on a Web server with multiple open ports and protocols, using a valid case oracle with valid HTTP messages is a good way to make sure that the HTTP server process is still running.

A valid case oracle will let you know if there is a crash or freeze in your target. However, the valid case oracle cannot "see" more subtle signs of trouble such as memory leaks, unusual resource consumption, assertion failures, and more.

Codenomicon Defensics includes extensive support for valid case oracles in its test suites.

4.2.3. Resource Monitoring

Another useful technique that can be automated and generalized for multiple types of targets is resource monitoring. As fuzz test cases are presented to the target, the oracle examines the usage of memory, disk space, processing power, and other resources to look for anything out of the ordinary.

For targets that support Simple Network Management Protocol (SNMP), an oracle can retrieve SNMP values related to resource usage and other vital signs during fuzzing. Such an oracle allows straightforward use for any target supporting SNMP.

Codenomicon Defensics includes support both for browsing and selecting SNMP values as well as built-in automated support for using an SNMP oracle. The diagram (Refer to Figure 1) shows a graph of resource consumption during fuzz testing.

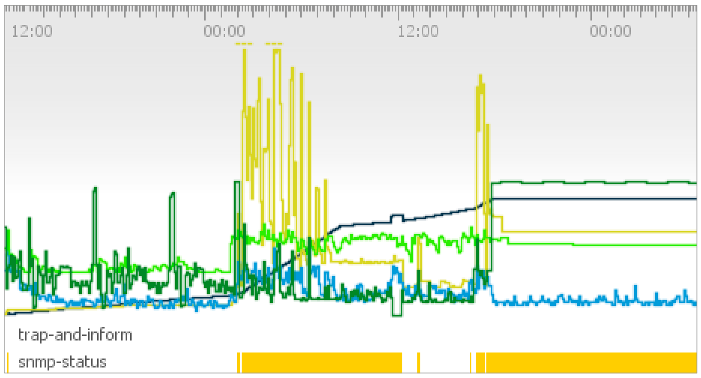


Figure 1: Resource Monitoring

4.3. Advanced Oracles

Better monitoring of a target results in more accurate detection of vulnerabilities and quicker resolution. This section lists some powerful approaches to the oracle challenge in fuzzing.

4.3.1. External

Because software targets come in all shapes and sizes, it's impossible to make a generalized oracle that works for everything. The valid case oracle works well because it uses the same interface that's being tested.

Target-specific oracles usually have to be created on a case-by-case basis. When this happens, it's important that your fuzzer makes it easy to integrate the oracle you've created.

Codenomicon Defensics provides a mechanism called *external instrumentation* in which it will call a user-supplied script to determine if a failure has occurred on the target. This mechanism is invoked after every test case so that when a failure occurs, the test case that caused it can be identified and run again. External instrumentation scripts can examine log files, monitor resource usage, check on processes, or do anything else that can be scripted.

4.3.2. Dynamic Binary

One of the strengths of fuzzing is that it is a black box technique—you can fuzz software even when you don't have the source code.

Even without source code, some advanced methods are available for examining the target software as it's running.

- Tools like strace, for example, can show how the target software interacts with the underlying operating system.
- Dynamic Binary Instrumentation (DBI) is a promising technique in which executable code can be modified on the fly to allow detailed examination of its workings without the need for source code.
- Microsoft's PageHeap is a tool that enables heap allocation monitoring for any executable code.

4.3.3. Source Code Instrumentation

If you do have the source code available, other advanced oracular techniques are available to detect failures. Depending on your fuzzer and the technique you're using, you might be able to integrate these oracles into the fuzzing process to detect failures precisely.

- If you build the target software with debugging symbols, you'll be able to run the target in a debugger such as gdb. Among other things, this will show you when assertions fail, which might or might not indicate some kind of failure. Furthermore, errors that are hidden by poor use of try-catch structures can be revealed when running software in a debugger.
- Similarly, target software compiled with debugging symbols can be run with valgrind's memcheck tool to analyze memory use and check for errors.
- Target software built with AddressSanitizer (ASan) will quickly expose memory usage errors.
- Other types of target instrumentation might also be available.

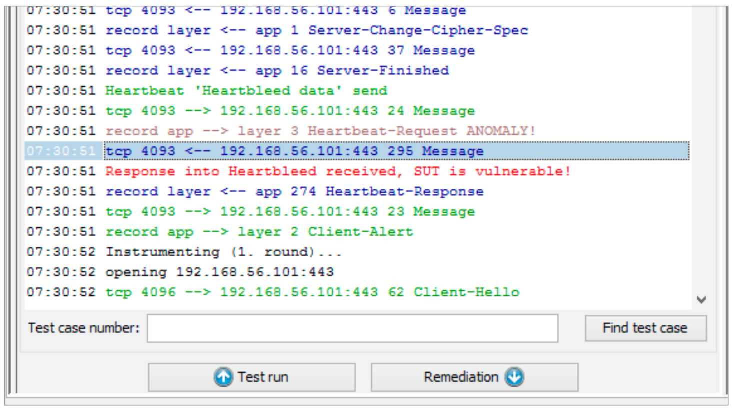
4.3.4. Functional and Behavioral Checks

Certain attack vectors can be examined during fuzz testing to detect functional failures. For example, while fuzzing TLS messages, it's an obvious functional failure if the fuzzer supplies bad authentication credentials, but the target allows the fuzzer access anyhow.

Depending on the attack vector, a variety of functional checks can be implemented, including authentication bypass, code injection, message amplification, and more.

For functional checks to be possible the fuzzer must already have a complete model and grammar of the tested protocol or file format, which means the fuzzer must be generational.

Codenomicon discovered Heartbleed while adding such functional checks (called SafeGuard) to its TLS fuzzer. The following screenshot (Refer to Figure 2) shows a section of a Defensics test log in which the Heartbleed vulnerability has been detected in target software.



```
07:30:51 tcp 4093 <-- 192.168.56.101:443 6 Message
07:30:51 record layer <-- app 1 Server-Change-Cipher-Spec
07:30:51 tcp 4093 <-- 192.168.56.101:443 37 Message
07:30:51 record layer <-- app 16 Server-Finished
07:30:51 Heartbeat 'Heartbleed data' send
07:30:51 tcp 4093 --> 192.168.56.101:443 24 Message
07:30:51 record app --> layer 3 Heartbeat-Request ANOMALY!
07:30:51 tcp 4093 <-- 192.168.56.101:443 295 Message
07:30:51 Response into Heartbleed received, SUT is vulnerable!
07:30:51 record layer <-- app 274 Heartbeat-Response
07:30:51 tcp 4093 --> 192.168.56.101:443 23 Message
07:30:51 record app --> layer 2 Client-Alert
07:30:52 Instrumenting (1. round)...
07:30:52 opening 192.168.56.101:443
07:30:52 tcp 4096 --> 192.168.56.101:443 62 Client-Hello
```

Figure 2: Functional and Behavioral Checks

5. Wrap Up

Fuzzing is an excellent technique for locating vulnerabilities in software. The basic premise is to deliver intentionally malformed input to target software and detect failure. A complete fuzzer has three components. A *poet* creates the malformed inputs or test cases. A *courier* delivers test cases to the target software. Finally, an *oracle* detects target failures.

Different fuzzing techniques have a significant effect on fuzzing effectiveness. For the most part, the poet is more effective when it is able to create test cases that are almost correct, but anomalous in some way. Different oracle techniques provide varying levels of failure detection capability. Multiple oracle techniques can be used together to help detect the maximum number of failures.

Fuzzing is a crucial tool in software vulnerability management, both for organizations that create software as well as organizations that use software. Fuzzing must be deployed as part of a process. Builders use fuzzing as an integral part of a Secure Development Life Cycle, while buyers use fuzzing as a crucial tool in verification and validation. Financially speaking, fuzzing saves money simply because it is much less expensive to fix bugs earlier rather than later. Bugs that are fixed before deployment or product release are no big deal. Bugs that are located and possibly exploited in production scenarios can be hugely expensive.

In the broader context of risk management, finding and fixing bugs proactively with fuzzing provides protection against other types of damage. If a bug in your product leads to a catastrophic failure or a massive data breach, your reputation might not recover and your legal liability could be insurmountable. If you provide a service, such as healthcare, power, communications, or another critical infrastructure, bugs in the products you're using could lead to human harm or environmental damage.

Finding and fixing bugs saves money, protects your customers and your reputation, and in many cases, saves lives.

For more information about fuzz testing theory or Codenomicon's fuzz testing solutions, please contact us at **sales@codenomicon.com** or visit us at **www.codenomicon.com**.