Practical Considerations Of Fuzzing: Generating Insight into Areas of Risk

Jonathan Knudsen

About the Author

Jonathan Knudsen is principal security engineer at Codenomicon in Raleigh, NC. E-mail: jonathan@codenomicon.com

Each attack vector represents some body of code—code that contains vulnerabilities. The attack surface of a network or system is the sum of the attack surfaces of its component devices.

For an introduction to the motivations and concepts of fuzzing, please see the article by Baker[1] on page 42 of this issue of *Horizons*. Additional background on fuzzing also has been reported previously.[2]

If you read the the article by Baker,[1] you are probably convinced that fuzzing is a crucial part of any software development life cycle or validation and verification process. What now? How can you get started?

This article describes where the rubber meets the road, starting from making a plan and progressing through how discovered vulnerabilities can be fixed. It provides an overview on mapping the attack surface of your target, prioritizing attack vectors, and creating a test plan. Specific recommendations on desirable fuzzer features also are described, as well as details on instrumentation, fuzzing roles, and how to help developers fix located vulnerabilities. Note that some portions of this article were adapted from the Fuzz Testing Maturity Model (FTMM).[3]

Creating a Test Plan

Before firing up your fuzzer and trying to discover vulnerabilities in your target device, you need a plan.

Attack Surface Analysis

The first step in creating a test plan is to map out the available attack surface. Devices and networks are vulnerable where any type of software takes any type of input. Each place at which a piece of software takes input is an attack vector. The sum of all attack vectors is an attack surface.

Consider, for example, a typical infusion pump. Each place on the device that some piece of software accepts some kind of input, either through a network protocol or as a file, is an attack vector. Let's assume the pump receives drug libraries over a network from a control computer.

- The drug libraries are likely described in an extensible markup language (XML) document. The XML parser is a target, as is the custom code that deals with the content of the XML document.
- The XML document might be carried over HTTP. An HTTP server is responsible for parsing HTTP messages.
- TLS protocol handling is most likely supported through openSSL.
- TCP and IP protocols are supplied by the underlying operating system, likely a variant of Linux.
- The pump might have other services running as well, such as SSH.

Each attack vector represents some body of code—code that contains vulnerabilities. The attack surface of a network or system is the sum of the attack surfaces of its component devices.

When mapping the attack surface, don't forget about nonnetwork inputs. For example,

some devices might accept files as inputs. A mobile phone, for example, can display images and play audio files, both of which are attack vectors.

Tools for Attack Surface Analysis

Several simple tools can help in mapping an attack surface. The first is documentation. Specifications and technical documents about your target can be very helpful in understanding which network protocols it uses and the types of files it might take as input.

If the target is based on Linux, the netstat command provides a useful list of listening network ports, which often is an excellent starting point in listing the attack surface.

For the "view from the outside," running nmap from a different computer will provide a list of open ports, though its results might be foiled by firewalls or other techniques.[4] The output of netstat run on the device itself is more accurate.

Assigning Priorities to Attack Vectors

Mapping the attack surface is useful, but putting it in order focuses the testing on the most dangerous attack vectors. The time and money spent on fuzzing always will be limited, so you want to be sure you are doing your most important testing first. The following criteria should be used when assigning priorities.

- **Accessibility.** How easy is it for an attacker to reach the vector? For example, some network equipment exposes protocols on a network interface that faces the public internet, while other protocols are usable only on a separate administrative network. The public-facing interface is much more likely to be attacked. Pay special attention to "trusted" interfaces that are allowed to pass through firewalls; these attack vectors will not be protected by a firewall or network perimeter and are critically important.
- **Experience.** Code that has been widely deployed and thoroughly exercised is likely to be more robust than freshly written code or older code that has not been closely examined. The Linux kernel IPv4 implementation, for example, is so widely deployed, scrutinized, and tested that it is resilient in the face of malformed input. In effect, it has been fuzzed by the real world over many years.
- **Authentication.** Some vulnerabilities can be triggered only after successful authentication. From an attacker's point of view, such vulnerabilities require an additional step (obtaining login credentials) before they can be exploited. A higher priority should be placed on unauthenticated vulnerabilities.

After you finish prioritizing your attack vectors, you can choose which tools you will use for testing. The prioritized list is now a simple test plan.

Choose Your Weapons

In simplest terms, a fuzzer is a piece of software that tests another piece of software (i.e., the target). Writing a fuzzer is easy, whereas writing a great fuzzer is fiendishly difficult!

Five lines of Python or Ruby are enough to create a rudimentary random fuzzer, but such a fuzzer has crippling shortcomings. First, the outputs produced by the fuzzer resemble noise and generally will be ignored by target software, as they do not look like any type of valid input. Second, the fuzzer has no way of monitoring the target and therefore no way of knowing if a failure has occurred. Third, if something does go wrong in the target, how can the failure be reproduced?

The following features make a fuzzer highly useful.

- The fuzzer should be generational (or model based), meaning that the fuzzer itself has total knowledge of the protocol being tested. An HTTP protocol fuzzer, for example, should know about all possible HTTP messages and all message fields, as well as the rules governing how messages are exchanged among endpoints. A generational fuzzer creates test cases for every field of every message and systematically breaks every rule of the protocol being tested.
- A savvy test case engine iterates through the protocol model and creates the malformed inputs, or test cases, that will be used to exercise the target. Remember, fuzzing is an infinite space problem, so 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, ideally, should have been testing and breaking software for many years.

The time and money spent on fuzzing always will be limited, so you want to be sure you are doing your most important testing first.

- High-quality test cases are 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 detect whether a failure has occurred. This is one of the fundamental mechanisms of fuzzing. Several instrumentation methods are discussed below.
- When outright failures or unusual behavior occur 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.
- The fuzzer should be easy to use. If the learning curve is too steep, no one will want to use it.

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 with the defined thresholds.

Defining Failure

One reason that fuzzing is difficult is the many failure modes of software targets, including process crashes, kernel panics, unhandled exceptions, assertion failures, busy loops, and 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 with the defined thresholds.

Resource monitoring can be as simple as a human observing the output of the "top" utility on a Linux-based target or as complex as automated retrieval of SNMP values.

Detecting Failure

How does a fuzzer monitor the health of its target during testing? Software targets come in all shapes and sizes, so vital signs that make sense for a network router probably won't make sense for an implantable medical device.

One of the simplest and best approaches is valid case instrumentation. After each test case is delivered to the target, the fuzzer delivers a valid message and looks for a valid response. If the target is still able to respond appropriately, it is considered healthy. If the target cannot respond, the preceding test case is marked as a failure.

Another approach that works for certain classes of devices is SNMP instrumentation, in which vital signs are retrieved from the target and examined. Vital signs can include statistics about memory and processor consumption. Of course, this only works for devices that support SNMP.

More elaborate instrumentation can be performed using custom-developed scripts. The scripts can do anything you want to assess the health of the target, such as logging in, checking network connections, examining processes, and interfacing to custom hardware.

In theory, fuzzing is a black box discipline, meaning that the fuzzer has no inside knowledge of the target. The fuzzer interacts with the target through one of its attack vectors (i.e. an external interface).

In practice, the tester's knowledge and access to the internals of the target will make fuzzing more effective. Examining log files or console output on the target, during or after testing, can provide valuable insights into the behavior of the target and its response to fuzzing. Inspecting front panel lights or any other visible indications also can provide insight into the state of the target. Finally, setting up existing sessions (e.g., communication link between a glucose monitor and an insulin pump) might allow the tester to see an interruption if a failure occurs on the target.

Furthermore, running test targets in a debugger or using other developer tools can provide valuable information about the behavior of the target during testing. Using developer tools, failed assertions and other conditions can be detected that might otherwise go unnoticed.

Generational fuzzing is a highly effective method for locating vulnerabilities in software.

Server, Client, and File Testing

The exact method of delivering fuzzing test cases to target software depends on how the target takes its input.

The simplest scenario is server testing, in which the target has the server role in a protocol conversation. The server simply listens for incoming messages from clients. Fuzzing a server is simple: the fuzzer acts like a client and repeatedly connects to the target, delivering a new fuzz test case each time. For example, a web server listens for incoming HTTP requests from a client. The fuzzer takes the role of client, in essence acting like a web browser.

Client testing is a little trickier. Here, the fuzzer acts like a server and listens for incoming client messages. The client target must be coerced into repeatedly connecting to the fuzzer. For example, to test a web browser, the fuzzer acts like a web server, listening for incoming HTTP requests. Whenever one arrives, the fuzzer sends back a test case, a malformed HTTP response. Client testing frequently requires scripting, both to drive the testing forward and to detect errors in the target.

Some targets are transparent devices. For example, a firewall might examine network communications as they pass through the target. Fuzzing such a target usually involves sending test cases from one side of the target to a very robust endpoint on the other side.

Finally, target software might take input in the form of files. The target expects the files to have a certain structure. Fuzzing is a great way to challenge those expectations and probe for vulnerabilities. DICOM files and the images that can be contained in DICOM files are both excellent examples. Suppose you create 10,000 fuzzed DICOM files. How do you get your target software to attempt to parse them? No tool can anticipate every possible delivery mechanism. Solutions usually involve some amount of scripting, just as with client testing.

Client testing is a little trickier. Here, the fuzzer acts like a server and listens for incoming client messages.

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, and 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.

Also of note, testing performed by a builder likely will use a different configuration and environment from buyers. Strictly speaking, even if a builder has thoroughly tested a product, cautious buyers should do their own fuzzing for verification and validation.

Generational fuzzing is a highly effective technique for locating vulnerabilities, but the target software does not become more robust and more secure until vulnerabilities are fixed.

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 considerably different environment than production code, but the emulation enables substantially more testing. If most testing is done off device, smoke testing should be performed on the real device.

Fixing Vulnerabilities

Generational fuzzing is a highly effective technique for locating vulnerabilities, but the target software does not become more robust and more secure until vulnerabilities are fixed.

Vulnerabilities are fixed by developers, usually in a separate team from the people who are doing the fuzzing. A smooth remediation workflow ensures that testers can effectively communicate results to developers and that developers have all the information they need to fix vulnerabilities.

Here are several common approaches for fixing vulnerabilities:

- In the best case, a remediation workflow allows developers to see the same results and information as those seen by testers and allows developers to rerun the same test cases that caused failures in the target. This allows developers to reproduce the same failures, using debuggers and other tools, which should make it easy to fix the vulnerabilities.
- Another approach is to supply developers with packet captures or raw test case material that corresponds to target failures. This gives developers the anomalous input that caused a failure in the target, which should enable them to track down and fix the vulnerability.
- Finally, core dumps, log files, and other artifacts of failures can be useful in debugging and fixing vulnerabilities. These artifacts, generated during testing, might be sufficient for developers to understand and eliminate vulnerabilities.

Fuzzing is likely to uncover large numbers of vulnerabilities in software that have not been fuzzed previously. A risk analysis that considers the probability of attack (more likely on a port exposed to the internet) and hazard level (patient safety versus temporary inconvenience) may be used to triage the vulnerabilities.

How Much Fuzzing Is Enough?

The rule of thumb with fuzzers is that more is almost always better. Fuzzing is an infinite space problem; different fuzzers are likely to find different vulnerabilities. Running more fuzzers increases your chances of finding more vulnerabilities.

In 2008, Miller[5] performed fascinating research in which he intentionally introduced vulnerabilities into a piece of software, then ran multiple fuzzers to see how well the fuzzers located the vulnerabilities. Among his findings is a correlation between using more fuzzers and finding more vulnerabilities.

How do you know if you've done enough fuzzing? You want to be confident that the target software will not fail either accidentally or through deliberate attack. This is a surprisingly difficult subject and will be covered in an upcoming article about fuzzing metrics. In general, two approaches are practical:

1. Compare yourself to your peers. If you are keeping pace with others in your community or industry, then your software should be roughly comparable in terms of robustness and security.
2. Use the FTMM,[3] a tool-agnostic standard that maps fuzz testing metrics to specific maturity levels. This gives builder and buyer organizations a standard scale for communicating about fuzz testing.

Conclusion

Generational fuzzing is a highly effective method for locating vulnerabilities in software. Fuzzing has been embraced by fields as diverse as the telecommunication and automotive industries. More recently, the U.S. Food and Drug Administration adopted a generational fuzzer as the foundation of its newly created cybersecurity laboratory, in which medical devices will be tested with the aim of improving safety and reliability.[6]

Fuzzing works well as part of a software development life cycle or a product validation and verification process. Applying the principles described in this article can generate insight into areas of higher risk. Generational fuzzing can provide concrete testing results and the ability to improve the safety and security of your final product.

By knowing what vulnerabilities are present in their products, medical device manufacturers can better understand their risk profile, determine the most effective course of action to manage the inherent risk, and bring that risk to a level that is acceptable to all stakeholders. ■

References

1. **Baker S.** Fuzzing: A Solution Chosen by the FDA to Investigate Detection of Software Vulnerabilities. *Horizons*. 2014;Spring:42–47.
2. **Knudsen J.** Make Software Better with Fuzzing. Available at: www.codenomicon.com/news/editorial/Make%20Software%20Better%20with%20Fuzzing.pdf. Accessed Feb. 21, 2014.
3. **Codenomicon.** Fuzz Testing Maturity Model. Available at: www.codenomicon.com/resources/ftmm.shtml. Accessed Feb. 21, 2014.
4. **Linux.** Beginner's Guide to Nmap. Available at: www.linux.com/learn/tutorials/290879-beginners-guide-to-nmap. Accessed Feb. 21, 2014.
5. **Miller C.** Fuzz by Number: More Data About Fuzzing Than You Ever Wanted to Know. Available at: cansecwest.com/csw08/csw08-miller.pdf. Accessed Feb. 21, 2014.
6. **AAMI.** FDA to Develop Cybersecurity Laboratory. Available at: www.aami.org/news/2013/072413_FDA_Cybersecurity_Lab.html. Accessed Feb. 21, 2014.

By knowing what vulnerabilities are present in their products, medical device manufacturers can better understand their risk profile, determine the most effective course of action to manage the inherent risk, and bring that risk to a level that is acceptable to all stakeholders.