Fuzzing for Software Security Testing and Quality Assurance

Ari Takanen
Jared DeMott
Charlie Miller

ARTECH HOUSE
BOSTON | LONDON
artechhouse.com

Library of Congress Cataloging-in-Publication Data
A catalog record for this book is available from the U.S. Library of Congress.

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library.

ISBN 13: 978-1-59693-214-2

Cover design by Igor Valdman

685 Canton Street
Norwood, MA 02062

10 9 8 7 6 5 4 3 2 1

Contents

CHAPTER 1

Introduction

Welcome to the world of fuzzing! In a nutshell, the purpose of fuzzing is to send anomalous data to a system in order to crash it, therefore revealing reliability problems. Fuzzing is widely used by both security and by quality assurance (QA) experts, although some people still suffer from misconceptions regarding its capabilities, effectiveness, and practical implementation. Fuzzing can be defined as

> A highly automated testing technique that covers numerous boundary cases using invalid data (from files, network protocols, API calls, and other targets) as application input to better ensure the absence of exploitable vulnerabilities. The name comes from modem applications' tendency to fail due to random input caused by line noise on "fuzzy" telephone lines.[1]

But before you explore fuzzing further, we ask you to try to understand why you are interested in fuzzing. If you are reading this, one thing is clear: You would like to find bugs in software, preferably bugs that have security implications. Why do you want to find those flaws? Generally, there are three different purposes for looking for these types of defects:

1. Quality Assurance (QA): Testing and securing your internally developed software.
2. System Administration (SA): Testing and securing software that you depend on in your own usage environment.
3. Vulnerability Assessment (VA): Testing and trying to break into someone else's software or system.

We have seen a number of books about fuzzing that are written by security experts for security experts. There are also a handful of books that cover topics relevant to fuzzing, written by quality assurance engineers for other quality assurance engineers. Another set of books consists of "hacking cookbooks," which teach you to use a set of tools without providing much original content about the topic at hand. In real life, testing practices are not that different from security assessment practices, or from hacking. All three require some theoretical knowledge, and some information on the available tools, as it is not efficient to continuously reinvent the wheel and develop our own tools for every task that arises.

[1]Peter Oehlert, "Violating Assumptions with Fuzzing," *IEEE Security & Privacy* (March/April 2005): 58–62.

In this book, we will look at fuzzing from all of these perspectives. No matter what your purpose and motivation for fuzzing, this is the book for you. We will view fuzzing from a developer's perspective, as well as through the eyes of an enterprise end user. We will also consider the requirements of a third-party assessment team, whether that is a testing consultant or a black-hat hacker. One goal of this book is to level the playing field between software companies (testers) and vulnerability analysts (hackers). Software testers can learn from the talents of hackers, and vice versa. Also, as end users of software, we should all have access to the same bug-hunting knowledge and experience through easy-to-use tools.

Why did we choose fuzzing as the topic for this book? We think that fuzzing is the most powerful test automation tool for discovering security-critical problems in software. One could argue that code auditing tools find more flaws in code, but after comparing the findings from a test using intelligent fuzzing and a thorough code audit, the result is clear. Most findings from code auditing tools are false positives, alerts that have no security implications. Fuzzing has no such problem. There are no false positives. A crash is a crash. A bug is a bug. And almost every bug found with fuzzing is exploitable at some level, at minimum resulting in denial of service. As fuzzing is generally black-box testing, every flaw is, by definition, remotely exploitable, depending, of course, on the interface you are fuzzing and to some extent on your definition of exploitation. Fuzzing is especially useful in analyzing closed-source, off-the-shelf software and proprietary systems, because in most cases it does not require any access to source code. Doesn't that sound almost too good to be true?

In this chapter we will present an overview of fuzzing and related technologies. We will look at why security mistakes happen and why current security measures fail to protect us from security compromises that exploit these mistakes. We will explore how fuzzing can help by introducing proactive tools that anyone can use to find and eliminate security holes. We will go on to look where fuzzing is currently used, and why. Finally, we will get a bit more technical and review the history of fuzzing, with focus on understanding how various techniques in fuzzing came into existence. Still, remember that the purpose of this chapter is only to provide an overview that will prepare you for what is coming later in the book. Subsequent chapters will provide more details on each of these topics.

1.1 Software Security

As stated before, fuzzing is a great technique for finding security-critical flaws in any software rapidly and cost effectively. Unfortunately, fuzzing is not always used where it should be used, and therefore many systems we depend on are immature from a security perspective. One fact has emerged from the security field: *Software will always have security problems*. Almost all software can be hacked easily. But if you become familiar with the topic of software security and the related techniques, you might be able to make a difference on how many of those parasitic security mistakes eventually remain in the software. This is what software security is about.

Very few people today know what software security really is, even if they are so-called "security experts." Like the maps in ancient history used to warn, the dangerous area just outside the map is sometimes best left alone. The uncharted territory just read, "Here be dragons," meaning that you should not venture there. It is too scary or too challenging. Fortunately for software security, the age of darkness is over because the first explorers risked their souls and delved into the mystic lands of hacking, trying to explain security to ordinary software developers. First, they were feared for their new skills, and later they were blamed for many of the dangerous findings they encountered. Even today they are thought to possess some secret arts that make them special. But what they found was not that complex after all.

Well, in our defense we have to say that we have explored the wilderness for more than ten years; so one could say that we are familiar with the things beyond the normal scope of software engineering and that we know the field well. We have looked at software security from an academic and commercial fuzzer developer perspective, but also through the eyes of a security consultant, a contract hacker, and an independent analyst. Grab a book on secure programming, or one of the few good books on security, and you will be amazed how simple it is to improve software security through secure programming practices. Software security is a highly desirable topic to learn. The only thing you need is motivation and a will to learn. Young children could master those skills (and they continue to do so). Anyone can find a unique new vulnerability in almost any piece of commercial software and write an exploit for it. Anyone can even write worms or viruses with some focus on the topic. If you do not know what a vulnerability is, or what an exploit is, or even how to use one, you might be better off reading some other security-related book first. Don't have patience for that? Well, we will do our best to explain along the way.

But first, let us look at what solutions are available out there. Software security can be introduced at various phases and places, starting from research and development (R&D), then entering the test-lab environment, and finally in the operations phase (Figure 1.1).

In R&D, fuzzing can be used both in the early prototyping phase and in the implementation phase, where the majority of programming takes place. In fact, immediately when the first operational prototype is ready, it can be fuzzed. But a

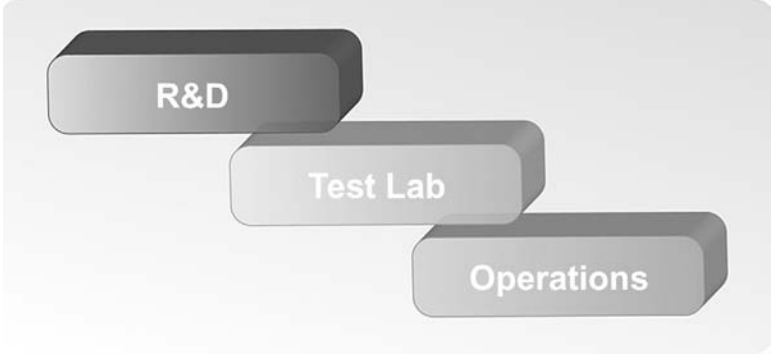

Figure 1.1 Phases in the software life cycle, and the resulting places for using fuzzing.

more natural tool for software security in this phase is a source code auditing tool, which we will discuss later. One thing common in R&D environments is that developers themselves use the tools at this stage.

Testing as an activity takes place throughout all phases of the software life cycle. Although most programmers conduct the first unit tests as part of R&D, a dedicated team typically performs most of the remaining testing efforts.[2] A test lab environment can be quite different from an R&D environment. In a test lab, a system can be tested with any available tools, and the test results can be analyzed with all possible metrics selected for use. A test lab may contain expensive, dedicated tools for load and performance testing, as well as fuzzing. Indeed, some commercial fuzzers have been implemented as fixed test appliances for traditional test lab environments.

In operations, various post-deployment techniques are used to increase software security. Vulnerability scanners or security scanners such as Nessus[3] are most commonly used in a live environment. An important criterion for test tools in an operational environment is that they should not disrupt the operation of critical services. Still, penetration testing services are often conducted against live systems, as they need to validate the real environment from real threats. This is because not all problems can be caught in the controlled confines of a test lab. Similarly, fuzzing should be carefully considered for operational environments. It may be able to find more flaws than when compared to a test lab, but it will most probably also disrupt critical services.

Never forget that there is no silver bullet solution for security: Not even fuzzing can guarantee that there are no flaws left in the software. The fast-paced world of technology is changing, growing, and constantly evolving, for better or worse. This is good news for testers: People will be writing new software for the foreseeable future. And new software inevitably brings new bugs with it. Your future careers are secured. Software is becoming more complex, and the number of bugs is thought to be directly proportional to lines of code. The security testing tools you have will also improve, and as you will see, fuzzing tools certainly have evolved during the past 20 years.

1.1.1 Security Incident

The main motivation for software security is to avoid security incidents: events where someone can compromise the security of a system through active attacks, but also events where data can be disclosed or destroyed through mistakes made by people, or due to natural disasters such as floods or tornadoes. Active compromises are the more significant factor for discussions related to fuzzing. "Accidental" incidents may arise when software is misconfigured or a single bit among massive

[2]The exact proportion of testing intermixed with actual development and testing performed by dedicated testers depends on the software development methodology used and the organizational structure of the development team.

[3]Nessus Security scanner is provided by Tenable Security and is available at www.nessus.org

amounts of data flips due to the infamous alpha-particles, cosmic rays, or other mysterious reasons and result in the crash of a critical service. Accidental incidents are still quite rare events, and probably only concernservice providers handling massive amounts of data such as telecommunication. The related threat is minimal, as the probability of such an incident is insignificant. The threat related to active attacks is much more severe.

Software security boasts a mixture of terms related to security incidents. Threats are typically related to risks of loss for an asset (money, data, reputation). In a security compromise, this threat becomes realized. The means of conducting the compromise is typically done through an attack, a script or malicious code that misuses the system, causing the failure, or potentially even resulting in the attacker's taking control of the system. An attack is used to exploit a weakness in the system. These weaknesses are called software vulnerabilities, defects, or flaws in the system.

Example threats to assets include

- Availability of critical infrastructure components;
- Data theft using various technical means.

Example attacks are the actual exploitation tools or means:

- Viruses and worms that exploit zero-day flaws;
- Distributed Denial of Service (DDoS) attacks.

Vulnerabilities can be, for example,

- Openness of wireless networks;
- Processing of untrusted data received over the network;
- Mishandling of malicious content received over the network.

Even the casual security hackers are typically one step ahead of the system administrators who try to defend their critical networks. One reason for that is the easy availability of vulnerability details and attack tools. You do not need to keep track of all available hacking tools if you know where you can find them when you need them. However, now that computer crime has gone professional, all tools might not be available to the good guys anymore. Securing assets is becoming more challenging, as white-hat hacking is becoming more difficult.

1.1.2 Disclosure Processes

There are hundreds of software flaws just waiting to be found, and given enough time they will be found. It is just a matter of who finds them and what they do with the findings. In the worst case, each found security issue could result in a "patch and penetrate" race: A malicious user tries to infiltrate the security of the services before the administrators can close the gaping holes, or a security researcher keeps reporting issues to a product vendor one by one over the course of many months or years, forcing the vendor to undergo a resource-intensive patch testing, potential recertification, and worldwide rollout process for each new reported issue.

Three different models are commonly used in vulnerability disclosure processes:

1. No disclosure: No details of the vulnerability, nor even the existence of the vulnerability, are disclosed publicly. This is often the case when vulnerabilities are found internally, and they can be fixed with adequate time and prioritization. The same can also happen if the disclosing organization is a trusted customer or a security expert who is not interested in gaining fame for the findings. People who do not like the no-disclosure model often argue that it is difficult for the end users to prioritize the deployment of updates if they do not know whether they are security-related and that companies may not bother to fix even critical issues quickly unless there is direct pressure from customers to do so.
2. Partial disclosure: This is the most common means of disclosure in the industry. The vendor can disclose the nature of the correction and even a workaround when a proper correction is not yet available. The problem with partial disclosure is that hackers can reverse-engineer the corrections even when limited information is given. Most partial disclosures end up becoming fully known by those who are interested in the details and have the expertise to understand them.
3. Full disclosure: All details of the vulnerability, including possible exploitation techniques, are disclosed publicly. In this model, each reader with enough skill can analyze the problem and prioritize it accordingly. Sometimes users decide to deploy the vendor-provided patches, but they can also build other means of protecting against attacks targeting the vulnerability, including deploying IDS/IPS systems or firewalls.

From an end-user perspective, there are several worrying questions: Will an update from the vendor appear on time, before attackers start exploiting a reported vulnerability? Can we deploy that update immediately when it becomes available? Will the update break some other functionality? What is the total cost of the repair process for our organization?

As a person conducting fuzzing, you may discover a lot of critical vulnerabilities that can affect both vendors and end users. You may want to consider the consequences before deciding what to do with the vulnerabilities you find. Before blowing the whistle, we suggest you familiarize yourself with the works done on vulnerability disclosure at Oulu University Secure Programming Group (OUSPG).[4]

1.1.3 Attack Surfaces and Attack Vectors

Now that we have explained how software vulnerabilities affect us and how they are announced, let's take a close look at the nature of vulnerabilities in software. Vulnerabilities have many interesting aspects that could be studied, such as the level of exploitability and the mode of exploitability. But one categorization is the most important—the accessibility of the vulnerability. Software vulnerabilities have secu-

[4]Vulnerability disclosure publications and discussion tracking, maintained by University of Oulu since 2001. Available at www.ee.oulu.fi/research/ouspg/sage/disclosure-tracking

rity implications only when they are accessible through external interfaces, as well as when triggers can be identified and are repeatable.

Fuzzing enables software testers, developers, and researchers to easily find vulnerabilities that can be triggered by malformed or malicious inputs via standard interfaces. This means that fuzzing is able to cover the most exposed and critical attack surfaces in a system relatively well. *Attack surface* has several meanings, depending on what is analyzed. To some, attack surface means the source code fingerprint that can be accessed through externally accessible interfaces. This is where either remote or local users interact with applications, like loading a document into a word processor, or checking email from a remote mail server. From a system testing standpoint, the total attack surface of a system can comprise all of the individual external interfaces it provides. It can consist of various network components, various operating systems and platforms running on those devices, and finally, all client applications and services.

Interfaces where privilege changes occur are of particular interest. For example, network data is unprivileged, but the code that parses the network traffic on a server always runs with some privilege on its host computer. If an attack is possible through that network-enabled interface—for example, due to a vulnerability in message parsing code—an unprivileged remote user could gain access to the computer doing the parsing. As a result, the attacker will elevate its privileges into those of the compromised process. Privilege elevation can also happen from lower privileges into higher privileges on the same host without involving any network interfaces.

An example of fuzzing remote network-enabled attack surfaces would be to send malformed web requests to a web server, or to create malformed video files for viewing in a media player application. Currently, dozens of commercial and free fuzz testing frameworks and fuzz-data generators of highly varying testing capability exist. Some are oriented toward testing only one or a few interfaces with a specialized and predefined rule set, while some are open frameworks for creating fuzz tests for any structured data. The quality of tests can vary depending on the complexity of the interface and the fuzzing algorithms used. Simple tools can prove very good at testing simple interfaces, for which complex tools could be too time-consuming or expensive. On the other hand, a complex interface can only be tested thoroughly with a more capable fuzzing system.

The various routes into a system, whether they are remote or local, are called attack vectors. A local vector means that an attacker already has access to the system to be able to launch the attack. For instance, the attacker may possess a username and password, with which he or she can log into the system locally or remotely. Another option is to have access to the local user interface of the system. Note that some user interfaces are realized over the network, meaning that they are not local. The attacker can also have access to a physical device interface such as a USB port or floppy drive. As an example, a local attack vector can consist of any of the following:

1. Graphical User Interface (GUI);
2. Command Line User Interface (CLI);
3. Programming Interfaces (API);
4. Files;

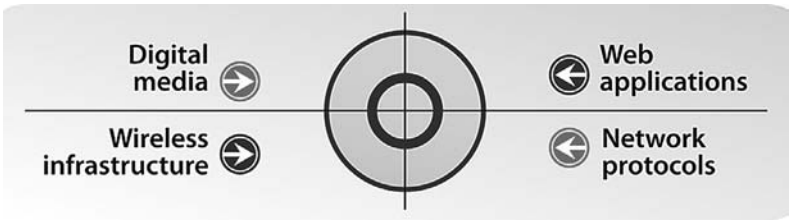

Figure 1.2 Categories of remote attack vectors in most network-enabled systems.

5. Local network loops (RPC, Unix sockets or pipes, etc.);
6. Physical hardware access.

Much more interesting interfaces are those that are accessible remotely. Those are what fuzzing has traditionally focused on. Note that many local interfaces can also be accessed remotely through active content (ActiveX, JavaScript, Flash) and by fooling people into activating malicious content (media files, executables).

Most common categories of remote interfaces that fuzzers test are displayed in Figure 1.2.

1. Web applications: Web forms are still the most common attack vector. Almost 50% of all publicly reported vulnerabilities are related to various packaged or tailored web applications. Almost all of those vulnerabilities have been discovered using various forms of fuzzing.
2. Digital media: File formats are transferred as payloads over the network, e.g., downloaded over the web or sent via email. There are both open source and commercial fuzzers available for almost any imaginable file format. Many fuzzers include simple web servers or other tools to automatically send the malicious payloads over the network, whereas other file fuzzers are completely local. Interesting targets for file fuzzers are various media gateways that cache, proxy, and/or convert various media formats, antivirus software that has to be able to separate valid file contents from malware, and, of course, various widely used operating system components such as graphics libraries or metadata indexing services.
3. Network protocols: Standardization organizations such as IETF and 3GPP have specified hundreds of communication protocols that are used everywhere in the world. A network protocol fuzzer can be made to test both client- and server-side implementations of the protocol. A simple router on the network can depend on dozens of publicly open protocol interfaces, all of which are extremely security critical due to the requirement of the router being available for any remote requests or responses.
4. Wireless infrastructure: All wireless networks are always open. Fuzzing has been used to discover critical flaws in Bluetooth and 802.11 WLAN (WiFi) implementations, for example, with these discoveries later emerging as sophisticated attack tools capable of exploiting wireless devices several miles away. Wireless devices are almost always embedded, and a flaw found in a wireless device has the potential of resulting in a total corrup-

tion of the device. For example, a flaw in an embedded device such as a Bluetooth-enabled phone can totally corrupt the memory of the device with no means of recovery.

Fuzzers are already available for well over one hundred different attack vectors, and more are emerging constantly. The hottest trends in fuzzing seem to be related to communication interfaces that have just recently been developed. One reason for that could be that those technologies are most immature, and therefore security flaws are easy to find in them. Some very interesting technologies for fuzzers include

- Next Generation Networks (Triple-Play) such as VoIP and IPTV;
- IPv6 and related protocols;
- Wireless technologies such as WiFi, WiMAX, Bluetooth, and RFID;
- Industrial networks (SCADA);
- Vehicle Area Networks such as CAN and MOST.

We will not list all the various protocols related to these technologies here, but if you are interested in finding out which protocols are the most critical ones for you, we recommend running a port scanner[5] against your systems, and using network analyzers[6] to monitor the actual data being sent between various devices in your network.

1.1.4 Reasons Behind Security Mistakes

Clearly, having security vulnerabilities in software is a "bad thing." If we can define the attack surface and places where privilege can be elevated, why can't we simply make the code secure? The core reason is that the fast evolution of communications technologies has made software overwhelmingly complex. Even the developers of the software may not understand all of the dependencies in the communication infrastructure. Integration into off-the-shelf platforms and operating systems brings along with it unnecessary network services that should be shut down or secured before deploying the products and services. Past experience has shown that all complex communication software is vulnerable to security incidents: The more complex a device is, the more vulnerable it usually proves in practice. For example, some security solutions are brought in to increase security, but they may instead enable new vulnerabilities due to their complexity. If a thousand lines of code have on average between two to ten critical defects, a product with millions of lines of code can easily contain thousands of flaws just waiting to be found. For secure solutions, look for simple solutions over complex ones, and minimize the feature sets instead of adding anything unnecessary. Everything that is not used by majority of the users can probably be removed completely or shut down by default. If you cannot do that or do not want to do that, you have to test (fuzz) that particular feature very thoroughly.

[5]One good free port scanner is NMAP. Available at http://insecure.org/nmap

[6]A popular network analyzer is Wireshark. Available at www.wireshark.org

Standardization and harmonization of communications have their benefits, but there is also a downside to them. A standardized homogeneous infrastructure can be secure if all possible best practices are in place. But when security deployment is lacking in any way, such an environment can be disrupted with one single flaw in one single standard communication interface. Viruses and worms often target widely deployed homogeneous infrastructures. Examples of such environments include e-mail, web, and VoIP. A unified infrastructure is good from a security point of view only if it is deployed and maintained correctly. Most people never update the software in their mobile phones. Think about it! Some people do not even know if they can update the software on their VoIP phones. If you do not want to update all those devices every week, it might be beneficial to try to fuzz them, and maybe fix several flaws in one fell swoop.

Open, interconnected wireless networks pose new opportunities for vulnerabilities. In a wireless environment, anyone can attack anyone or anything. Wireless is by definition always open, no matter what authentication and encryption mechanisms are in place. For most flaws that are found with fuzzing in the wireless domain, authentication is only done after the attack has already succeeded. This is because in order to attempt authentication or encryption, input from an untrusted source must be processed. In most cases the first message being sent or received in wireless communications is completely anonymous and unauthenticated. If you do not need wireless, do not use it. If you need to have it open, review the real operation of that wireless device and at minimum test the handling of pre-authentication messages using fuzzing.

Mobility will increase the probability of vulnerabilities, but also will make it easier to attack those devices. Mobile devices with complex communication software are everywhere, and they often exist in an untrusted environment. Mobility will also enable anonymity of users and devices. Persons behind security attacks cannot be tracked reliably. If you have a critical service, it might be safer to ask everyone to authenticate himself or herself in a secure fashion. At minimum, anything that can be accessed anonymously has to be thoroughly tested for vulnerabilities.

1.1.5 Proactive Security

For an enterprise user, there are typically two different measures available for protecting against security incidents: reactive and proactive. The main difference between reactive security measures and proactive security measures is who is in control of the process. In reactive measures, you react to external events and keep running around putting fires out. In proactive security measures, you take a step back and start looking at the system through the eyes of a hacker. Again, a hacker in this sense is not necessarily a criminal. A hacker is a person who, upon hearing about a new technology or a product, will start having doubts on the marketing terms and specifications and will automatically take a proactive mindset into analyzing various technologies. Why? How? What if? These are just some of the questions someone with this mindset will start to pose to the system.

Let us take a look at the marketing terms for proactive security from various commercial fuzzer companies:

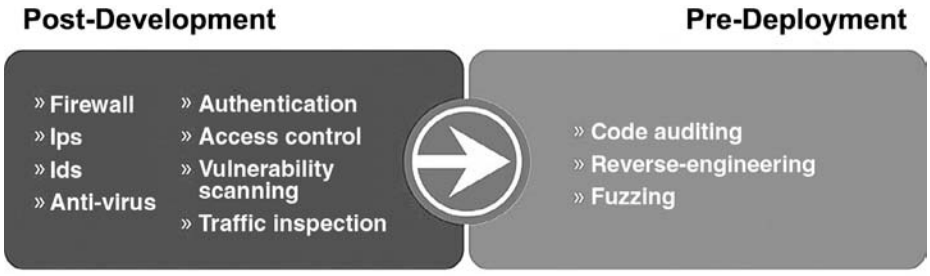

Figure 1.3 Reactive post-deployment versus proactive pre-deployment security measures.

> XXX enables companies to preemptively mitigate unknown and published threats in products and services prior to release or deployment—before systems are exposed, outages occur, and zero-day attacks strike.
>
> By using YYY, both product developers and end users can proactively verify security readiness before products are purchased, upgraded, and certainly before they are deployed into production environments.

In short, proactive, pre-deployment, or pre-emptive software security equals to catching vulnerabilities earlier in the software development life cycle (SDLC), and catching also those flaws that have not been disclosed publicly, which traditional reactive security measures cannot detect. Most traditional security solutions attempt to detect and block attacks, as opposed to discovering the underlying vulnerabilities that these attacks target (Figure 1.3).

A post-deployment, reactive solution depends on other people to ask the questions critical for security. An example of a reactive solution is a signature based anti-virus system. After a new virus emerges, researchers at the security vendor start poking at it, trying to figure out what makes it tick. After they have analyzed the new virus, they will make protective measures, trying to detect and eliminate the virus in the network or at a host computer. Another example of a reactive solution is an Intrusion Detection System (IDS) or an Intrusion Prevention System (IPS). These systems look for known exploits and block them. They do not attempt to identify which systems are vulnerable or what vulnerabilities exist in software. One could argue that even a firewall is reactive solution, although the pace of development is much slower. A firewall protects against known attacks by filtering communication attempts at the perimeter. The common thread with most post-deployment security solutions is that they all target "attacks," and not "vulnerabilities." They are doomed to fail because of that significant difference. Every time there is a unique vulnerability discovered, there will be hundreds, if not thousands of attack variants trying to exploit that worm-sized hole. Each time a new attack emerges, the retroactive security vendors will rush to analyze it and deploy new fingerprints to their detection engines. But based on studies, unfortunately they only detect less than 70% of attacks, and are often between 30 and 60 days late.[7]

[7]"Anti-Virus Is Dead; Long Live Anti-Malware." Published by Yankee Group. Jan. 17, 2007. www.marketresearch.com/map/prod/1424773.html

One could argue that security scanners (or vulnerability scanners) are also proactive security tools. However, security scanners are still mostly based on known attacks and exhibit the same problems as other reactive security solutions. A security scanner cannot find a specific vulnerability unless the vulnerability is publicly known. And when a vulnerability becomes known, attacks usually already exist in the wild exploiting it. That does not sound very proactive, does it? You still depend on someone else making the decisions for you, and in their analyzing and protecting your assets. Security scanners also look for known issues in standard operating systems and widely used hosts, as data on known vulnerabilities is only available for those platforms. Most tests in security scanners are based on passive probing and fingerprinting, although they can contain active hostile tests (real exploits) for selected known issues. Security scanners cannot find any unknown issues, and they need regular updating of threats. Security scanners also rarely support scanning anything but very popular operating systems and selected network equipment.

An additional recent problem that is becoming more and more challenging for reactive security solutions that depend on public disclosure is that they do not know the problems (vulnerabilities) anymore. This is because the "public disclosure movement" has finally died down. However, raising awareness about security mistakes can only be a good thing. Public disclosure is fading because very few people actually benefit from it. Manufacturers and software developers do not want to publish the details of vulnerabilities, and therefore today we may not know if they have fixed the problems or not. Hackers do not want to publish the details, as they can sell them for profit. Corporate enterprise customers definitely do not want to publish any vulnerability details, as they are the ones who will get damaged by any attacks leading to compromises. The only ones who publish details are security companies trying to make you believe they actually have something valuable to offer. Usually, this is just bad and irresponsible marketing, because they are getting mostly second-hand, used vulnerabilities that have already been discovered by various other parties and exploited in the wild.

A proactive solution will look for vulnerabilities and try to resolve them before anyone else learns of them and before any attacks take place. As we said, many enterprise security measures fail because they are focused on known attacks. A truly proactive approach should focus on fixing the actual flaws (unknown zero-day vulnerabilities) that enable these attacks. An attack will not work if there is no underlying vulnerability to exploit. Vulnerability databases indicate that programming errors cause 80% of vulnerabilities, so the main focus of security solutions should probably be in that category of flaws. Based on research conducted at the PROTOS project and also according to our experience at commercial fuzzing companies, 80% of software will crash when tested via fuzzing. That means we can find and eliminate many of those flaws with fuzzing, if we spend the effort in deploying fuzzing.

1.1.6 Security Requirements

We have discussed fuzzing and its uses, but the truth is not all software is security-critical, and not all software needs fuzzing. Just as is the case with all security measures, introducing fuzzing into development or deployment processes needs to be

based on the requirement set for the system. Unfortunately, traditional security requirements are feature-driven and do not really strike a chord with fuzzing.

Typical and perhaps the most common subset of security requirements or security goals consists of the following: confidentiality, integrity, and availability. Fuzzing directly focuses on only one of these, namely availability, although many vulnerabilities found using fuzzing can also compromise confidentiality and integrity by allowing an attacker to execute malicious code on the system. Furthermore, the tests used in fuzzing can result in corrupted databases, or even in parts of the memory being sent back to the fuzzer, which also constitute attacks against confidentiality and integrity.

Fuzzing is much closer to the practices seen in quality assurance than those related to traditional security requirements. This may have been one of the main reasons why fuzzing has not been widely adopted so far in software engineering processes: Security people have mostly driven its deployment. Without solid requirements to fulfill, you only end up with a new tool with no apparent need for it. The result is that your expensive fuzzer equipment ends up just collecting dust in some far-away test lab.

1.2 Software Quality

> *Thrill to the excitement of the chase! Stalk bugs with care, methodology, and reason. Build traps for them. . . . Testers! Break that software (as you must) and drive it to the ultimate—but don't enjoy the programmer's pain.*
>
> Boris Beizer[8]

People who are not familiar with testing processes might think that the purpose of testing is to find flaws. And the more flaws found, the better the testing process is. Maybe this was the case a long time ago, but today things are different. Modern testing is mostly focused on two things: verification and validation (V&V). Although both terms are used ambiguously, there is an intended difference.

Verification attempts to answer the question: "Did we build the product right?" Verification is more focused on the methods (and in the existence of these methods), such as checklists, general process guidelines, industry best practices, and regulations. Techniques in verification ensure that the development, and especially the quality assurance process, is correct and will result in reduction of common mistakes.

Validation, on the other hand, asks: "Did we build the right product?" The focus is on ensuring and documenting the essential requirements for the product and in building the tests that will check those requirements. For any successful validation testing, one needs to proactively define and document clear pass/fail criteria for all functionality so that eventually when the tests are done, the test verdicts can be issued based on something that has been agreed upon beforehand.

Unfortunately, fuzzing does not fit well into this V&V model, as we will see here, and later in more detail in Chapter 3.

[8]Quote is from *Software Testing Techniques*, 2nd ed., Boris Beizer, International Thomson Computer Press. 1990. Abbreviated for brevity.

Testing is a time-consuming process that has been optimized over time at the same time that software has become more complex. With increasing complexity, devising a completely thorough set of tests has become practically impossible. Software development with a typical waterfall model and its variants—such as the iterative development process—proceed in phases from initial requirements through specification, design, and implementation, finally reaching the testing and post-deployment phases. These phases are rarely completely sequential in real-life development, but run in parallel and can revisit earlier steps. They can also run in cycles, such as in the spiral model. Due to this, the requirements that drive testing are drafted very early in the development process and change constantly. This is extremely true for various agile processes, where test requirements may be only rarely written down due to the fast change process.

If we look at fuzzing from a quality assurance perspective, fuzzing is a branch of testing; testing is a branch of quality control; quality control is a branch of quality assurance. Fuzzing differs from other testing methods in that it

- Tends to focus on input validation errors;
- Tends to focus on actual applications and dynamic testing of a finished product;
- Tends to ignore the responses, or valid behavior;
- Concentrates mostly on testing interfaces that have security implications.

In this section, we'll look at different kinds of testing and auditing of software from a tester's perspective. We will start with identifying how much you need to test (and fuzz) based on your needs. We will then define what a testing target is and follow that up with some descriptions of different kinds of testing as well as where fuzzing fits in with these definitions. Finally, we will contrast fuzzing with more traditional security measures in software development such as code auditing.

1.2.1 Cost-Benefit of Quality

From a quality assurance standpoint, it is vital to understand the benefits from defect elimination and test automation. One useful study was released in January 2001, when Boehm and Basili reviewed and updated their list of metrics on the benefits of proactive defect elimination. Their software defect reduction "Top 10" list includes the following items:[9]

1. Finding and fixing a software problem after delivery is often 100 times more expensive than finding and fixing it during the requirements and design phase.
2. Current software projects spend about 40% to 50% of their effort on avoidable rework.
3. About 80% of avoidable rework comes from 20% of the defects.

[9]Victor R. Basili, Barry Boehm. "Software defect reduction top 10 list." *Computer* (January 2001): 135–137.

4. About 80% of the defects come from 20% of the modules, and about half of the modules are defect free.
5. About 90% of the downtime comes from, at most, 10% of the defects.
6. Peer reviews catch 60% of the defects.
7. Perspective-based reviews catch 35% more defects than nondirected reviews.
8. Disciplined personal practices can reduce defect introduction rates by up to 70%.
9. All other things being equal, it costs 50% more per source instruction to develop high-dependability software products than to develop low-dependability software products. However, the investment is more than worth it if the project involves significant operations and maintenance costs.
10. About 40% to 50% of users' programs contain nontrivial defects.

Although this list was built from the perspective of code auditing and peer review (we all know that those are necessary), the same applies to security testing. If you review each point above from a security perspective, you can see that all of them apply to vulnerability analysis, and to some extent also to fuzzing. This is because every individual security vulnerability is also a critical quality issue, because any crash-level flaws that are known by people outside the development organization have to be fixed immediately. The defects found by fuzzers lurk in an area that current development methods such as peer reviews fail to find. These defects almost always are found only after the product is released and someone (a third party) conducts fuzz tests. Security is a subset of software quality and reliability, and the methodologies that can find flaws later in the software life-cycle should be integrated to earlier phases to reduce the total cost of software development.

The key questions to ask when considering the cost of fuzzing are the following.

1. **What is the cost per defect with fuzzing?** Some people argue that this metric is irrelevant, because the cost per defect is always less than the cost of a security compromise. These people recognize that there are always benefits in fuzzing. Still, standard business calculations such as ROI (return on investment) and TCO (total cost of ownership) are needed in most cases also to justify investing in fuzzing.
2. **What is the test coverage?** Somehow you have to be able to gauge how well your software is being tested and what proportion of all latent problems are being discovered by introducing fuzzing into testing or auditing processes. Bad tests done with a bad fuzzer can be counterproductive, because they waste valuable testing time without yielding any useful results. At worst case, such tests will result in over-confidence in your product and arrogance against techniques that would improve your product.[10] A solid fuzzer with good recommendations and a publicly validated track record will likely prove to be a better investment coverage-wise.

[10]We often hear comments like: "We do not need fuzzing because we do source code auditing" or "We do not need this tool because we already use this tool," without any consideration if they are complementary products or not.

3. **How much should you invest in fuzzing?** The motivation for discussing the price of fuzzing derives from the various options and solutions available in the market. How can you compare different tools based on their price, overall cost of usage, and testing efficiency? How can you compare the total cost of purchasing an off-the-shelf commercial fuzzer to that of adopting a free fuzzing framework and hiring people to design and implement effective tests from the ground up? Our experience in the market has shown that the price of fuzzing tools is not usually the biggest issue in comparisons. In commercial fuzzing, the cheapest tools usually prove to be the simplest ones—and also without exception the worst ones from a testing coverage, efficiency, and professional testing support standpoint. Commercial companies looking for fuzz testing typically want a fuzzer that (a) supports the interfaces they need to test, (b) can find as many issues as possible in the systems they test, and (c) are able to provide good results within a reasonable timeframe.

There will always be a place for both internally built tools and commercial tools. A quick Python[11] script might be better suited to fuzz a single isolated custom application. But if you are testing a complex communication protocol implementation or a complete system with lots of different interfaces, you might be better off buying a fuzzing tool from a commercial test vendor to save yourself a lot of time and pain in implementation. Each option can also be used at different phases of an assessment. A sample practice to analyze fuzzing needs is to

1. Conduct a QA risk analysis, and as part of that, possibly conduct necessary ad-hoc tests;
2. Test your product thoroughly with a commercial testing tool;
3. Hire a professional security auditing firm to do a second check of the results and methods.

1.2.2 Target of Test

In some forms of testing, the target of testing can be any "black box." All various types of functional tests can be directed at different kinds of test targets. The same applies for fuzzing. A fuzzer can test any applications, whether they are running on top of web, mobile, or VoIP infrastructure, or even when they are just standalone software applications. The target of a test can be one single network service, or it can be an entire network architecture. Common names used for test targets include

- SUT (system under test). An SUT can consist of several subsystems, or it can represent an entire network architecture with various services running on top of it. An SUT can be anything from banking infrastructure to a complex telephony system. SUT is the most abstract definition of a test target, because it can encompass any number of individual destinations for the tests.

[11]We mention Python as an example script language due to the availability of PyDBG by Pedram Amini. See PaiMei documentation for more details: http://pedram.redhive.com/PaiMei/docs

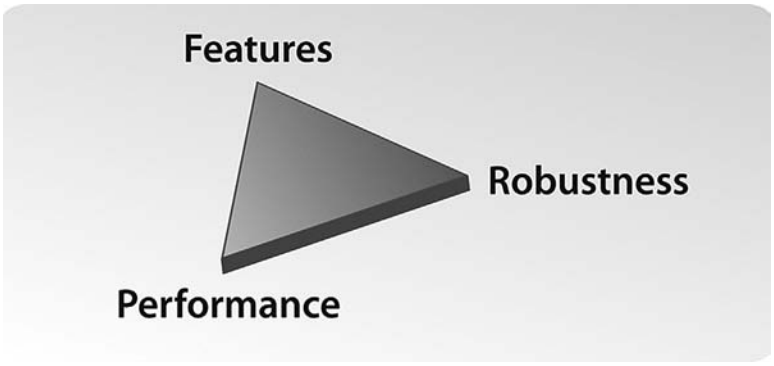

Figure 1.4 Testing purposes: features, performance, and robustness.

- DUT (device under test). A DUT is typically one single service or a piece of equipment, possibly connected to a larger system. Device manufacturers mainly use the term *DUT*. Some examples of DUTs include routers, WLAN access points, VPN gateways, DSL modems, VoIP phones, web servers, or mobile handsets.
- IUT (implementation under test). An IUT is one specific software implementation, typically the binary representation of the software. It can be a process running on a standard operating system, or a web application or script running on an application server.

In this book, we will most often refer to a test target as an SUT, because this term is applicable to all forms of test setups.

1.2.3 Testing Purposes

The main focus of fuzzing is on functional security assessment. As fuzzing is essentially functional testing, it can be conducted in various steps during the overall development and testing process. To a QA person, a test has to have a purpose, or otherwise it is meaningless.[12] Without a test purpose, it is difficult to assign a test verdict—i.e., Did the test pass or fail? Various types of testing have different purposes. Black-box testing today can be generalized to focus on three different purposes (Figure 1.4). Positive testing can be divided into feature tests and performance tests. Test requirements for feature testing consist of a set of valid use cases, which may consist of only few dozens or at most hundreds of tests. Performance testing repeats one of the use cases using various means of test automation such as record-and-playback. Negative testing tries to test the robustness of the system through exploring the infinite amount of possible anomalous inputs to find the tests that cause invalid behavior. An "anomaly" can be defined as any unexpected input

[12]Note that this strict attitude has changed lately with the increasing appreciation to agile testing techniques. Agile testing can sometimes appear to outsiders as ad-hoc testing. Fuzzing has many similarities to agile testing processes.

that deviates from the expected norm, ranging from simple field-level modifications to completely broken message structures or alterations in message sequences. Let us explore these testing categories in more detail.

Feature testing, or conformance testing, verifies that the software functions according to predefined specifications. The features can have relevance to security—for example, implementing security mechanisms such as encryption and data verification. The test specification can be internal, or it can be based on industry standards such as protocol specifications. A pass criterion simply means that according to the test results, the software conforms to the specification. A fail criterion means that a specific functionality was missing or the software operated against the specification. Interoperability testing is a special type of feature test. In interoperability testing, various products are tested against one another to see how the features map to the generally accepted criteria. Interoperability testing is especially important if the industry standards are not detailed enough to provide adequate guidance for achieving interoperability. Most industry standards always leave some features open to interpretation. Interoperability testing can be conducted at special events sometimes termed plug-fests (or unplug-fests in the case of wireless protocols such as Bluetooth).

Performance testing tests the performance limitations of the system, typically consisting of positive testing only, meaning it will send large amounts of legal traffic to the SUT. Performance is not only related to network performance, but can also test local interfaces such as file systems or API calls. The security implications are obvious: A system can exhibit denial-of-service when subjected to peak loads. An example of this is distributed denial of service (DDoS) attacks. Another example from the field of telephony is related to the "mothers' day effect," meaning that a system should tolerate the unfortunate event when everyone tries to utilize it simultaneously. Performance testing will measure the limits that result in denial of service. Performance testing is often called load testing or stress testing, although some make the distinction that performance testing attempts to prove that a system can handle a specific amount of load (traffic, sessions, transactions, etc.), and that stress testing investigates how the system behaves when it is taken over that limit. In any case, the load used for performance testing can either be sequential or parallel—e.g., a number of requests can be handled in parallel, or within a specified time frame. The acceptance criteria are predefined and can vary depending on the deployment. Whereas another user can be happy with a performance result of 10 requests per second, another user could demand millions of processed requests per minute. In failure situations, the system can crash, or there can be a degradation of service where the service is denied for a subset of customers.

Robustness testing (including fuzzing) is complementary to both feature and performance tests. Robustness can be defined as an ability to tolerate exceptional inputs and stressful environmental conditions. Software is not robust if it fails when facing such circumstances. Attackers can take advantage of robustness problems and compromise the system running the software. Most security vulnerabilities reported in the public are caused by robustness weaknesses.

Whereas both feature testing and performance testing are still positive tests, based on real-life use cases, robustness testing is strictly negative testing with tests that

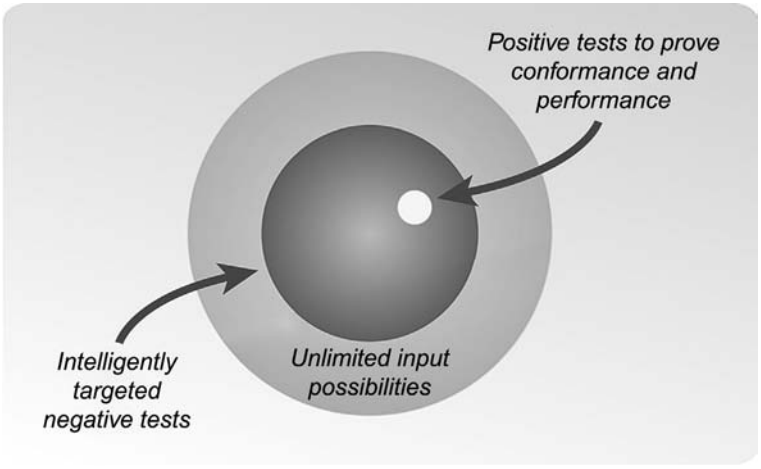

Figure 1.5 Limited input space in positive tests and the infinity of tests in negative testing.

should never occur in a well-behaving, friendly environment. For every use case in feature testing, you can create a performance test by running that use case in parallel or in rapid succession. Similarly, for every use case in feature testing, you can create "misuse cases" by systematically or randomly breaking the legal and valid stimuli.

With negative testing, the pass-fail criteria are challenging to define. A fail criterion is easier to define than a pass criterion. In robustness testing, you can define that a test fails if the software crashes, becomes unstable, or does other unacceptable things. If nothing apparent seems to be at fault, the test has passed. Still, adding more instrumentation and monitoring the system more closely can reveal uncaught failures with exactly the same set of tests, thus revealing the vagueness of the used pass-fail criteria. Fuzzing is one form of robustness testing, and it tries to fulfill the testing requirements in negative testing with random or semi-random inputs (often millions of test cases). But more often robustness testing is model-based and optimized, resulting in better test results and shorter test execution time due to optimized and intelligently targeted tests selected from the infinity of inputs needed in negative testing (Figure 1.5).

1.2.4 Structural Testing

Software rarely comes out as it was originally planned (Figure 1.6).[13] The differences between the specification and the implementation are faults (defects, bugs, vulnerabilities) of various types. A specification defines both positive and negative requirements. A positive requirement says what the software should do, and a negative requirement defines what it must not do. The gray area in between leaves some functionality undefined, open for interpretation. The implementation very

[13]J. Eronen, and M.Laakso. (2005) "A Case for Protocol Dependency." In *Proceedings of the First IEEE International Workshop on Critical Infrastructure Protection*. Darmstadt, Germany. November 3–4, 2005.

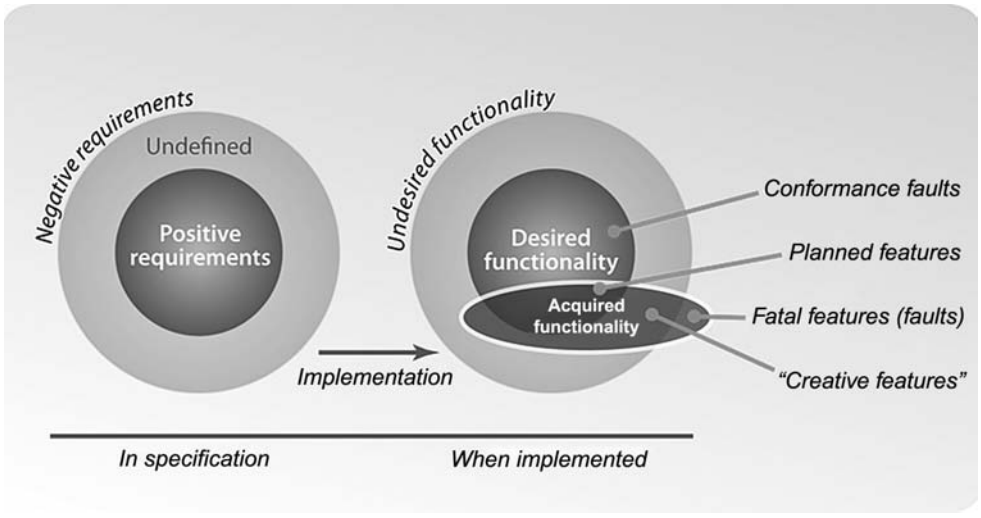

Figure 1.6 Specification versus implementation.

rarely represents the specification. The final product implements the acquired functionality, with some of the planned features present and some of them missing (conformance faults). In addition to implementing (or not implementing) the positive requirements, the final software typically implements some features that were defined as negative requirements (often fatal or critical faults). Creative features implemented during the software life cycle can either be desired or nondesired in the final product.

Whereas all critical flaws can be considered security-critical, many security problems also exist inside the set of creative features. One reason for this is that those features very rarely will be tested even if fuzzing is part of the software development life cycle. Testing plans are typically built based on a requirements specification. The reason for a vulnerability is typically a programming mistake or a design flaw.

Typical security-related programming mistakes are very similar in all communication devices. Some examples include

- Inability to handle invalid lengths and indices;
- Inability to handle out-of-sequence or out-of-state messages;
- Inability to tolerate overflows (large packets or elements);
- Inability to tolerate missing elements or underflows.

Try to think of implementation mistakes as undesired features. Whereas a username of eight characters has a feature of identifying users, nine characters can be used to shut the service down. Not very applicable, is it? Implementation flaws are often created due to vague definitions of how things should be implemented. Security-related flaws are often created when a programmer is left with too much choice when implementing a complex feature such as a security mechanism. If the requirements specification does not define how authentication must exactly be

implemented, or what type of encryption should be used, the programmers become innovative. The result is almost always devastating.

1.2.5 Functional Testing

In contrast to structural testing disciplines, fuzzing falls into the category of functional testing, which is more interested in how a system behaves in practice rather than in the components or specifications from which it is built. The system under test during functional testing can be viewed as a "black box," with one or more external interfaces available for injecting test cases, but without any other information available on the internals of the tested system. Having access to information such as source code, design or implementation specifications, debugging or profiling hooks, logging output, or details on the state of the system under test or its operational environment will help in root cause analysis of any problems that are found, but none of this is strictly necessary. Having any of the above information available turns the testing process into "gray-box testing," which has the potential to benefit from the best of both worlds in structural as well as functional testing and can sometimes be recommended for organizations that have access to source code or any other details of the systems under test. Access to the internals can also be a distraction.

A few good ideas that can be used in conjunction with fuzz testing when source code is available include focusing code auditing efforts on components or subsystems in which fuzzing has already revealed some initial flaws (implying that the whole component or portions of the code around the flaws might be also of similarly poor quality) or using debuggers and profilers to catch more obscure issues such as memory leaks during fuzz testing.

1.2.6 Code Auditing

"Use the source, Luke—if you have it!"

Anonymous security expert

Fuzzing is sometimes compared to code auditing and other white-box testing methods. Code auditing looks at the source code of a system in an attempt to discover defective programming constructs or expressions. This falls into the category of structural testing, looking at specifications or descriptions of a system in order to detect errors. While code auditing is another valuable technique in the software tester's toolbox, code auditing and fuzzing are really complementary to each other. Fuzzing focuses on finding some critical defects quickly, and the found errors are usually very real. Fuzzing can also be performed without understanding the inner workings of the tested system in detail. Code auditing is usually able to find more problems, but it also finds more false positives that need to be manually verified by an expert in the source code before they can be declared real, critical errors. The choice of which technique fits your purposes and testing goals best is up to you. With unlimited time and resources, both can be recommended. Neither fuzzing nor code auditing is able to provably find all possible bugs and defects in a tested system or program, but both of them are essential parts in building security into your product development processes.

1.3 Fuzzing

So far we have discussed vulnerabilities and testing. It is time to finally look at the real topic of this book, fuzzing.

1.3.1 Brief History of Fuzzing

Fuzzing is one technique for negative testing, and negative testing is nothing new in the quality assurance field. Hardware testing decades ago already contained negative testing in many forms. The most traditional form of negative testing in hardware is called *fault injection*. The term *fault injection* can actually refer to two different things. Faults can be injected into the actual product, through mutation testing, i.e., intentionally breaking the product to test the efficiency of the tests. Or the faults can be injected to data, with the purpose of testing the data-processing capability. Faults in hardware communication buses typically happen either through random inputs—i.e., white-noise testing—or by systematically modifying the data—e.g., by bit-flipping. In hardware, the tests are typically injected through data busses or directly to the various pins on the chip. Most modern chips contain a test channel, which will enable modification of not only the external interfaces but injection of anomalies in the data channels inside the chip.

Some software engineers used fuzzing-like test methods already in the 1980s. One proof of that is a tool called The Monkey: "The Monkey was a small desk accessory that used the journaling hooks to feed random events to the current application, so the Macintosh seemed to be operated by an incredibly fast, somewhat angry monkey, banging away at the mouse and keyboard, generating clicks and drags at random positions with wild abandon."[14] However, in practice, software testing for security and reliability was in its infancy until the late 1990s. It appeared as if nobody cared about software quality, as crashes were acceptable and software could be updated "easily." One potential reason for this was that before the availability of public networks, or the Internet, there was no concept of an "attacker." The birth of software security as a research topic was created by widely deployed buffer overflow attacks such as the Morris Internet Worm in 1988. In parallel to the development in the software security field, syntax testing was introduced around 1990 by the quality assurance industry.[15] Syntax testing basically consists of model-based testing of protocol interfaces with a grammar. We will explain syntax testing in more detail in Chapter 3.

A much more simpler form of testing gained more reputation, perhaps due to the easiness of its implementation. The first (or at least best known) rudimentary negative testing project and tool was called Fuzz from Barton Miller's research group at the University of Wisconsin, published in 1990.[16] Very simply, it tried ran-

[14]From Folklore.org (1983). www.folklore.org/StoryView.py?story=Monkey_Lives.txt

[15]Syntax testing is introduced in the *Software Testing Techniques* 2nd edition, by Boris Beizer, International Thomson Computer Press. 1990.

[16]More information on "Fuzz Testing of Application Reliability" at University of Wisconsin is available at http://pages.cs.wisc.edu/~bart/fuzz

History of Fuzzing

1983: The Monkey

1988: The Internet Worm

1989–1991:
- Boris Beizer explains Syntax Testing (similar to robustness testing).
- "Fuzz: An Empirical Study of Reliability . . ." by Miller et al. (Univ. of Wisconsin)

1995–1996:
- Fuzz revisited by Miller et al. (Univ. of Wisconsin).
- Fault Injection of Solaris by OUSPG (Oulu University, Finland).

1999–2001:
- PROTOS tests for: SNMP, HTTP, SIP, H.323, LDAP, WAP, . . .

2002:
- Codenomicon launch with GTP, SIP, and TLS robustness testers.
- Click-to-Secure (now Cenzic) Hailstorm web application tester.
- IWL and SimpleSoft SNMP fuzzers (and various other protocol specific tools).

dom inputs for command line options, looking for locally exploitable security holes. The researchers repeated the tests every five years, with same depressing results. Almost all local command-line utilities crashed when provided unexpected inputs, with most of those flaws exploitable. They described their approach as follows:

> There is a rich body of research on program testing and verification. Our approach is not a substitute for a formal verification or testing procedures, but rather an inexpensive mechanism to identify bugs and increase overall system reliability. We are using a coarse notion of correctness in our study. A program is detected as faulty only if it crashes or hangs (loops indefinitely). Our goal is to complement, not replace, existing test procedures. While our testing strategy sounds somewhat naive, its ability to discover fatal program bugs is impressive. If we consider a program to be a complex finite state machine, then our testing strategy can be thought of as a random walk through the state space, searching for undefined states.[17]

Inspired by the research at the University of Wisconsin, and by syntax testing explained by Boris Beizer, Oulu University Secure Programming Group (OUSPG) launched the PROTOS project in 1999.[18] The initial motivation for the work grew out of frustration with the difficulties of traditional vulnerability coordination and disclosure processes, which led the PROTOS team to think what could be done to expedite the process of vulnerability discovery and transfer the majority of the process back toward the software and equipment vendors from the security research community. They came up with the idea of producing fuzzing test suites for various interfaces and releasing them first to the vendors, and ultimately to the general public after the vendors had been able to fix the problems. During the following years,

[17]B. P. Miller, L. Fredriksen, and B. So. "An empirical study of the reliability of Unix utilities." *Communications of the Association for Computing Machinery, 33*(12)(1990):32–44.

[18]OUSPG has conducted research in the security space since 1996. www.ee.oulu.fi/research/ouspg

PROTOS produced free test suites for the following protocols: WAP-WSP, WMLC, HTTP-reply, LDAP, SNMP, SIP, H.323, ISAKMP/IKE, and DNS. The biggest impact occurred with the SNMP test suite, where over 200 vendors were involved in the process of repairing their devices, some more that nine months before the public disclosure. With this test suite the PROTOS researchers were able to identify numerous critical flaws within the ASN.1 parsers of almost all available SNMP implementations. This success really set the stage to alert the security community to this "new" way of testing called fuzzing.

1.3.2 Fuzzing Overview

To begin, we would like to clearly define the type of testing we are discussing in this book. This is somewhat difficult because no one group perfectly agrees on the definitions related to fuzzing. The key concept of this book is that of black-box or grey-box testing: delivering input to the software through different communication interfaces with no or very little knowledge of the internal operations of the system under test. Fuzzing is a black-box testing technique in which the system under test is stressed with unexpected inputs and data structures through external interfaces.

Fuzzing is also all about negative testing, as opposed to feature testing (also called conformance testing) or performance testing (also called load testing). In negative testing, unexpected or semi-valid inputs or sequences of inputs are sent to the tested interfaces, instead of the proper data expected by the processing code. The purpose of fuzzing is to find security-related defects, or any critical flaws leading to denial of service, degradation of service, or other undesired behavior. In short, fuzzing or fuzz testing is a negative software testing method that feeds malformed and unexpected input data to a program, device, or system.

Programs and frameworks that are used to create fuzz tests or perform fuzz testing are commonly called fuzzers. During the last 10 to 15 years, fuzzing has gradually developed from a niche technique toward a full testing discipline with support from both the security research and traditional QA testing communities.

Sometimes other terms are used to describe tests similar to fuzzing. Some of these terms include

- Negative testing;
- Protocol mutation;
- Robustness testing;
- Syntax testing;
- Fault injection;
- Rainy-day testing;
- Dirty testing.

Traditionally, terms such as negative testing or robustness testing have been used mainly by people involved with software development and QA testing, and the word fuzzing was used in the software security field. There has always been some overlap, and today both groups use both terms, although hackers tend to use the testing related terminology a lot less frequently. Testing terms and requirements in relation to fuzzing have always carried a notion of structure, determinism, and

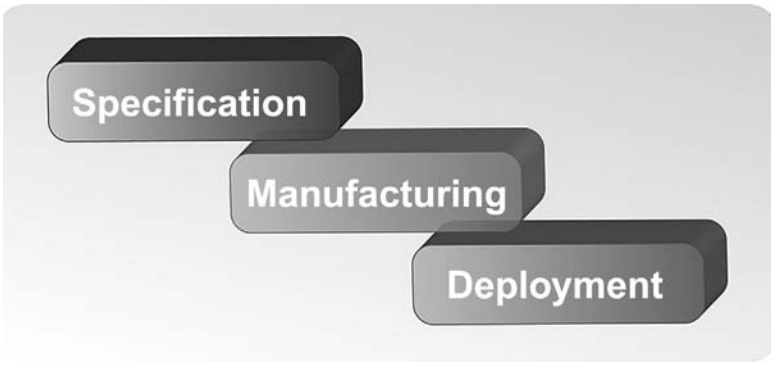

Figure 1.7 Various phases in the SDLC in which vulnerabilities are introduced.

repeatability. The constant flood of zero-day exploits has proved that traditional functional testing is insufficient. Fuzzing was first born out of the more affordable, and curious, world of randomness. Wild test cases tended to find bugs overlooked in the traditional development and testing processes. This is because such randomly chosen test data, or inputs, do not make any assumptions for the operation of the software, for better or worse. Fuzzing has one goal, and one goal only: to crash the system; to stimulate a multitude of inputs aimed to find any reliability or robustness flaws in the software. For the security people, the secondary goal is to analyze those found flaws for exploitability.

1.3.3 Vulnerabilities Found with Fuzzing

Vulnerabilities are created in various phases of the SDLC: specification, manufacturing, and deployment (Figure 1.7). Issues created in the specification or design phase are fundamental flaws that are very difficult to fix. Manufacturing defects are created by bad practices and mistakes in implementing a product. Finally, deployment flaws are caused by default settings and bad documentation on how the product can be deployed securely.

Looking at these phases, and analyzing them from the experience gained with known mistakes, we can see that implementation mistakes prevail. More than 70% of modern security vulnerabilities are programming flaws, with only less than 10% being configuration issues, and about 20% being design issues. Over 80% of communication software implementations today are vulnerable to implementation-level security flaws. For example, 25 out of 30 Bluetooth implementations crashed when they were tested with Bluetooth fuzzing tools.[19] Also, results from the PROTOS research project indicate that over 80% of all tested products failed with fuzz tests around WAP, VoIP, LDAP, and SNMP.[20]

Fuzzing tools used as part of the SDLC are proactive, which makes them the best solution for finding zero-day flaws. Reactive tools fail to do that, because they are based on knowledge of previously found vulnerabilities. Reactive tools only test

[19]Ari Takanen and Sami Petäjäsoja, "Assuring the Robustness and Security of New Wireless Technologies." Paper and presentation. *ISSE* 2007, Sept. 27, 2007. Warsaw, Poland.

[20]PROTOS project. www.ee.oulu.fi/protos

or protect widely used products from major vendors, but fuzzers can test any product for similar problems. With fuzzing you can test the security of any process, service, device, system, or network, no matter what exact interfaces it supports.

1.3.4 Fuzzer Types

Fuzzers can be categorized based on two different criteria:

1. Injection vector or attack vector.
2. Test case complexity.

Fuzzers can be divided based on the application area where they can be used, basically according to the attack vectors that they support. Different fuzzers target different injection vectors, although some fuzzers are more or less general-purpose frameworks. Fuzzing is a black-box testing technique, but there are several doors into each black box (Figure 1.8). Note also that some fuzzers are meant for client-side testing, and others for server-side testing. A client-side test for HTTP or TLS will target browser software; similarly, server-side tests may test a web server. Some fuzzers support testing both servers and clients, or even middleboxes that simply proxy, forward, or analyze passing protocol traffic.

Fuzzers can also be categorized based on test case complexity. The tests generated in fuzzing can target various layers in the target software, and different test cases penetrate different layers in the application logic (Figure 1.9). Fuzzers that change various values in protocol fields will test for flaws like overflows and integer problems. When the message structure is anomalized, the fuzzer will find flaws in message parses (e.g., XML and ASN.1). Finally, when message sequences are fuzzed, the actual state machine can be deadlocked or crashed. Software has sepa-

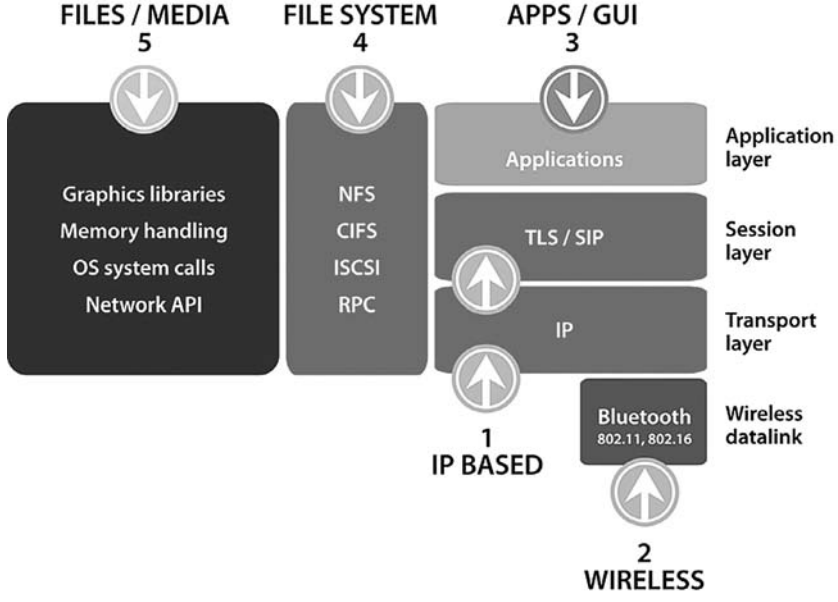

Figure 1.8 Attack vectors at multiple system levels.

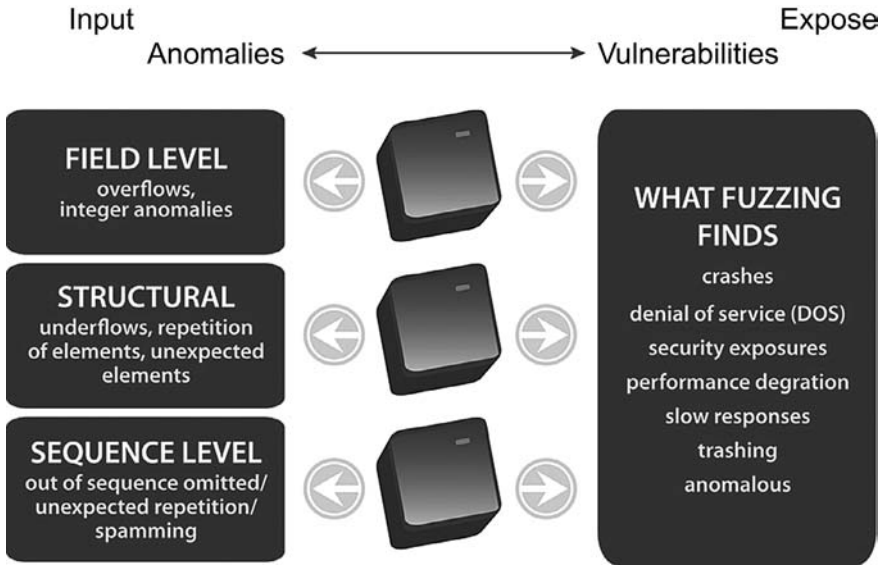

Figure 1.9 Different types of anomalies and different resulting failure modes.

rate layers for decoding, syntax validation, and semantic validation (correctness of field values, state of receiver) and for performing the required state updates and output generation (Figure 1.10). A random test will only scratch the surface, whereas a highly complex protocol model that not only tests the message structures but also message sequences will be able to test deeper into the application.

One example method of categorization is based on the test case complexity in a fuzzer:

- **Static and random template-based fuzzer:** These fuzzers typically only test simple request-response protocols, or file formats. There is no dynamic functionality involved. Protocol awareness is close to zero.
- **Block-based fuzzers:** These fuzzers will implement basic structure for a simple request-response protocol and can contain some rudimentary dynamic functionality such as calculation of checksums and length values.
- **Dynamic generation or evolution based fuzzers:** These fuzzers do not necessarily understand the protocol or file format that is being fuzzed, but they will learn it based on a feedback loop from the target system. They might or might not break the message sequences.
- **Model-based or simulation-based fuzzers:** These fuzzers implement the tested interface either through a model or a simulation, or they can also be full implementations of a protocol. Not only message structures are fuzzed, but also unexpected messages in sequences can be generated.

The effectiveness of fuzzing is based on how well it covers the input space of the tested interface (input space coverage) and how good the representative malicious and malformed inputs are for testing each element or structure within the

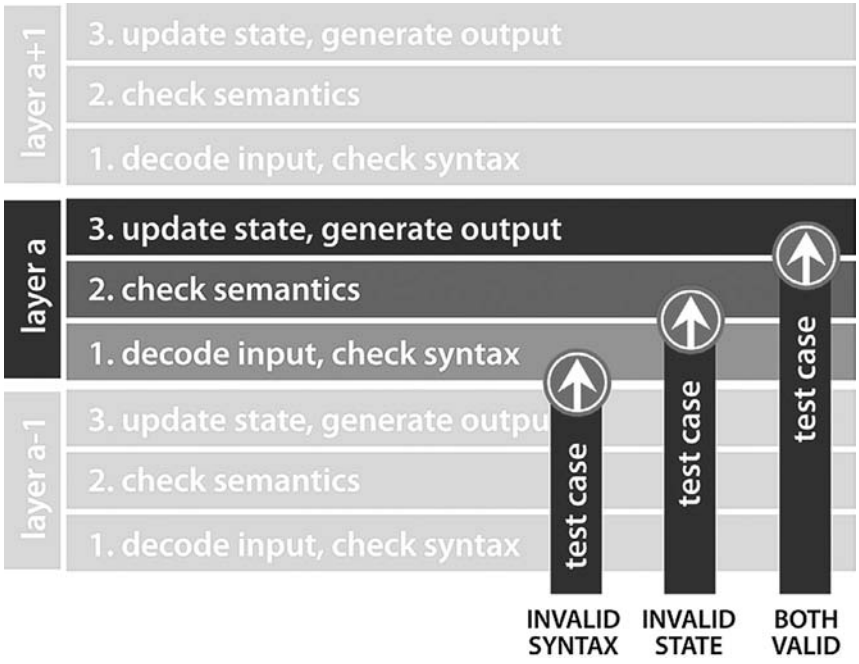

Figure 1.10 Effectivity of a test case to penetrate the application logic.

tested interface definition (quality of test data). Fuzzers that supply totally random characters may yield some fruit but, in general, won't find many bugs. It is generally accepted that fuzzers that generate their inputs with random data are very inefficient and can only find rather naive programming errors. As such, it is necessary for fuzzers to become more complex if they hope to uncover such buried or hard to find bugs. Very obscure bugs have been called "second-generation bugs." They might involve, for example, multipath vulnerabilities such as noninitialized stack or heap bugs.

Another dimension for categorizing fuzzers stems from whether they are model-based. Compared with a static, nonstateful fuzzer that may not be able to simulate any protocol deeper than an initial packet, a fully model-based fuzzer is able to test an interface more completely and thoroughly, usually proving much more effective in discovering flaws in practice. A more simplistic fuzzer is unable to test any interface very thoroughly, providing only limited test results and poor coverage. Static fuzzers may not be able to modify their outputs during runtime, and therefore lack the ability to perform even rudimentary protocol operations such as length or checksum calculations, cryptographic operations, copying structures from incoming messages into outgoing traffic, or adapting to the exact capabilities (protocol extensions, used profiles) of a particular system under test. In contrast, model-based fuzzers can emulate a protocol or file format interface almost completely, allowing them to understand the inner workings of the tested interface and perform any runtime calculations and other dynamic behaviors that are needed to achieve full interoperability with the tested system. For this reason, tests executed by a fully model-based fuzzer are usually able to penetrate much deeper within the system under test, exercising the packet parsing and input handling routines extremely thoroughly, and

reaching all the way into the state machine and even output generation routines, hence uncovering more vulnerabilities.

1.3.5 Logical Structure of a Fuzzer

Modern fuzzers do not just focus solely on test generation. Fuzzers contain different functionalities and features that will help in both test automation and in failure identification. The typical structure of a fuzzer can contain the following functionalities (Figure 1.11).

- Protocol modeler: For enabling the functionality related to various data formats and message sequences. The simplest models are based on message templates, whereas more complex models may use context-free protocol grammars or proprietary description languages to specify the tested interface and add dynamic behavior to the model.
- Anomaly library: Most fuzzers include collections of inputs known to trigger vulnerabilities in software, whereas others just use random data.
- Attack simulation engine: Uses a library of attacks or anomalies, or learns from one. The anomalies collected into the tool, or random modifications, are applied to the model to generate the actual fuzz tests.
- Runtime analysis engine: Monitors the SUT. Various techniques can be used to interact with the SUT and to instrument and control the target and its environment.
- Reporting: The test results need to be prepared in a format that will help the reporting of the found issues to developers or even third parties. Some tools do not do any reporting, whereas others include complex bug reporting engines.

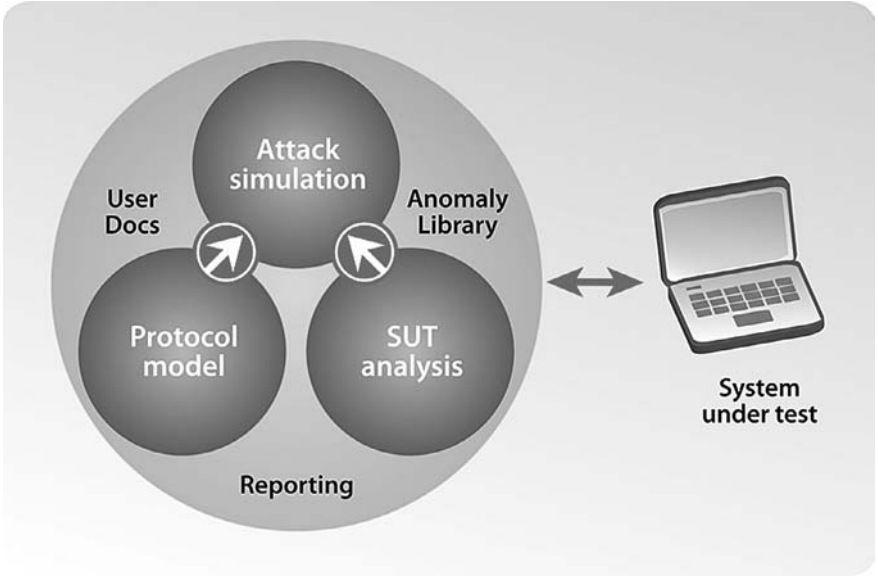

Figure 1.11 Generic structure of a fuzzer.

- Documentation: A tool without user documentation is difficult to use. Especially in QA, there can also be requirement for test case documentation. Test case documentation can sometimes be used in reporting and can be dynamically created instead of a static document.

1.3.6 Fuzzing Process

A simplified process of conducting a fuzz test consists of sequences of messages (requests, responses, or files) being sent to the SUT. The resulting changes and incoming messages can be analyzed, or in some cases they can be completely ignored (Figure 1.12). Typical results of a fuzz test contain the following responses:

- Valid response.
- Error response (may still be valid from a protocol standpoint).
- Anomalous response (unexpected but nonfatal reaction, such as slowdown or responding with a corrupted message).
- Crash or other failure.

The process of fuzzing is not only about sending and receiving messages. Tests are first generated and sent to the SUT. Monitoring of the target should be constant

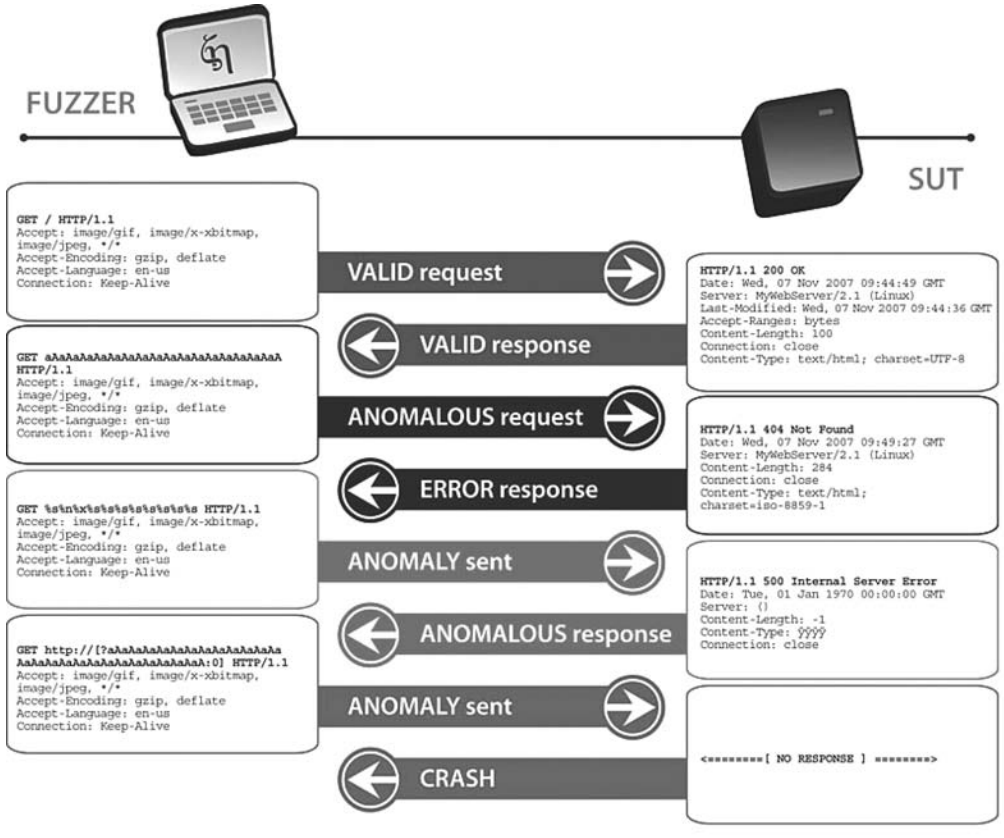

Figure 1.12 Example fuzz test cases and resulting responses from an SUT.

and all failures should be caught and recorded for future evaluation. A critical part of the fuzzing process is to monitor the executing code as it processes the unexpected input (Figure 1.13). Finally, a pass-fail criteria needs to be defined with the ultimate goal being to perceive errors as they occur and store all knowledge for later inspection. If all this can be automated, a fuzzer can have an infinite amount of tests, and only the actual failure events need to be stored, and analyzed manually.

If failures were detected, the reason for the failure is often analyzed manually. That can require a thorough knowledge of the system and the capability to debug the SUT using low-level debuggers. If the bug causing the failure appears to be security-related, a vulnerability can be proved by means of an exploit. This is not always necessary, if the tester understands the failure mode and can forecast the probability and level of exploitability. No matter which post-fuzzing option is taken, the deduction from failure to an individual defect, fixing the flaw, or potential exploit development task can often be equally expensive in terms of man-hours.

1.3.7 Fuzzing Frameworks and Test Suites

As discussed above, fuzzers can have varying levels of protocol knowledge. Going beyond this idea, some fuzzers are implemented as fuzzing frameworks, which means that they provide an end user with a platform for creating fuzz tests for arbitrary protocols. Fuzzer frameworks typically require a considerable investment of time and resources to model tests for a new interface, and if the framework does not offer ready-made inputs for common structures and elements, efficient testing also requires considerable expertise in designing inputs that are able to trigger faults in the tested interface. Some fuzzing frameworks integrate user-contributed test modules back to the platform, bringing new tests within the reach of other users, but for the most part fuzzing frameworks require new tests to always be implemented from scratch. These factors can limit the accessibility, usability, and applicability of fuzzing frameworks.

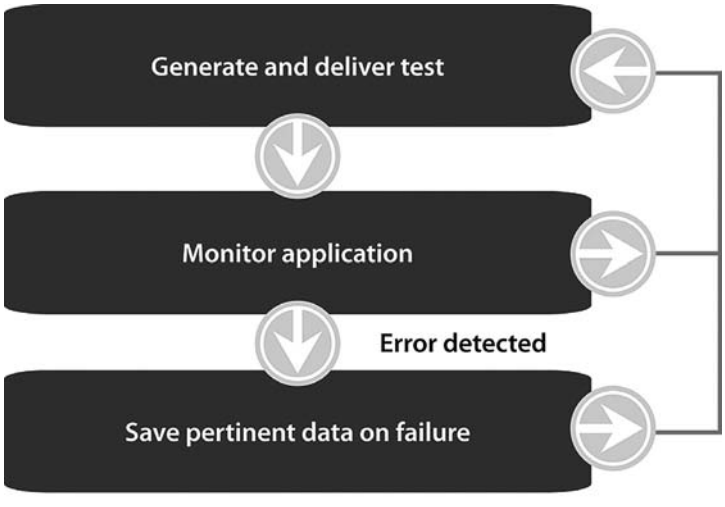

Figure 1.13 Fuzzing process consisting of test case generation and system monitoring.

1.3.8 Fuzzing and the Enterprise

Not all software developer organizations and device manufacturers use any type of fuzzing—although we all agree that they should. For many companies, fuzzing is something that is looked at after all other testing needs have been fulfilled. What should we do to motivate them to embrace fuzzing as part of their software development process? One driving force in changing the testing priorities can be created by using fuzzing tools in the enterprise environment.

The first action that enterprises could take is to require fuzzing in their procurement practices and kick out the vendors who do not use fuzzing in R&D. Several financial organizations and telecommunication service providers are already requiring some proof of negative testing or fuzzing from their vendors. All end customers of communication software have to stress the importance of security to the software developers and to the device manufacturers.

The second step would be to outsource fuzz testing. Fuzzing should be an integral part of penetration testing services offered by both test automation companies and security consultancies. But unfortunately only very few security experts today truly understand fuzzing, and very few quality assurance people understand the importance of negative testing.

The third and final step would be to make fuzzing tools more usable for enterprise users. Fuzzers should be easy to use by people who are not expert hackers. We also need to educate the end users to the available measures to assess the security of their critical system by themselves.

The people opposing the use of fuzzers in the enterprise environment use several statements to discourage their use. For example, these misconceptions can include the following statements.

- "You cannot fuzz in a live environment." This is not true. Attackers will fuzz the systems already, and proactive usage of fuzzing tools by system administrators can prepare an enterprise to withstand or at least understand the risks of such attacks. Even in an enterprise environment, it is still possible to fuzz selected systems and to mitigate its impact on business-critical services.
- "Manufacturers can find all flaws with fuzzing." This is also not true, because the complexity of a total integrated system is always more than the sum of the complexity of each item used. Manufacturers can never test all configurations, nor can they integrate the systems with all possible other systems, required middleware, and proprietary data. The actual environment will always affect test results.
- "Not our responsibility." Many enterprises think vendors should be the only ones testing their products and systems, not the end users or solution integrators. This is definitely not true! Not using negative testing practices at every possible phase when deploying critical systems can be considered negligent. Although some more responsible vendors test their products nowadays with stringent fuzz tests, this is hardly the case for all vendors in the world. Even though we all know vendors should do most of the testing, sadly there is still much to improve when it comes to vendors' attitudes toward preventing quality problems. Since we know that not all vendors will be doing the testing, it

is up to the integrators and users to at least do a "smoke test" for the final systems. If all fails, the systems and software should be sent back to the vendor, with a recommendation to invest in fuzzing and secure development processes.

Despite what has been stated above, you do need to be extremely careful when fuzzing a live system, as the testing process could crash your systems or corrupt data. If you can afford it, build a mirror setup of your critical services for testing purposes. Analyze your services from the attackers' perspective via a thorough analysis of the available attack vectors and by identifying the used protocols. You can test your perimeter defenses separately or together with the service they are trying to protect. You will be surprised at how many of your security solutions actually contain security-related flaws. After testing the perimeter, go on to test the reliability of your critical hosts and services without the protecting perimeter defenses. If all appears to be fine in the test environment, then cross your fingers and shoot the live system. But be prepared for crashes. Even if crashes occur, it is better you cause them to occur, rather than a malicious attacker. You will probably notice that a live system with live data will be more vulnerable to attacks than a white-room test system with default configurations.

1.4 Book Goals and Layout

This book is about fuzzing in all forms. Today all fuzzing-related terms—such as fuzzing, robustness testing, or negative black-box testing—have fused together in such a way that when someone says he or she has created a new RPC fuzzer, DNS robustness test suite, or a framework for creating negative tests against various file formats, we do not know the exact methods that may be in use. Is it random or systematic testing? Is it aimed at finding exploitable vulnerabilities or any robustness flaws? Can it be used as part of software development, or only against deployed systems? Our goal is to shed light on these mysteries. Throughout the book, these terms may be used synonymously, and if a particular connotation is implied, such will be indicated.

The purpose of this chapter was to give you an overview of fuzzing. In Chapter 2 we will look at fuzzing from the software vulnerability analysis (VA) perspective, and later in Chapter 3 we will look at the same issues from the quality assurance (QA) perspective. Chapter 4 will consider the business metrics related to fuzzing, both from cost and effectiveness perspectives. Chapter 5 will attempt to describe how various fuzzers can be categorized, with Chapter 6 identifying how the fuzz-test generators can be augmented with different monitoring and instrumentation techniques. Chapter 7 will provide an overview of current research, potentially providing an indication where future fuzzers are going. Chapter 8 will provide an independent fuzzer comparison, and Chapter 9 will present some sample use cases of where fuzzing can and is being used today.