steps in the process. Security flaws are a good example of such a flaw category. If buffer overflow vulnerabilities are consistently found in products ready to deploy, the best solution is to radically improve the developer practices.

Note that many security flaws go by different names in different phases of the software development process. During unit testing, a tester might presume that a boundary value flaw is not critical and will label the bug as such. But the same boundary value flaw is not critical and will not label the bug as such. Understanding these links is critical, so that people use the same terminology and have the same understanding of severity of bugs when discussing flaws.

3.2.4 Quality Brings Visibility to the Development Process

Quality assurance is a metric of the software development process. With good quality assurance processes, we are able to get visibility into the software development process and the current status of the software. Integration of system units and software modules is one measurement of the software process.

When a module is ready and tested, it can be labeled as completed. The software industry is full of experiences in which the software has been 90% ready for half of the development time. Security testing should also be an integral part of the software development life cycle and not a delay at the end that adds to this misconception of “almost ready.” Knowing the place and time for security testing enables product managers to understand the requirements of security testing from a time (and money) perspective.

3.2.5 End Users’ Perspective

Quality assurance is a broad topic and we need to narrow it down to be able to explain the selected parts in enough detail. Defining quality is a challenging task, and different definitions apply to different categories of quality. For example, tests that validate security properties can be very complex, and trust in their verdicts is sometimes limited. The definition of quality depends on who is measuring it.

For many testers, the challenge is how to measure and explain the efficiency of quality assurance so that the end customer will understand it. Quality assurance needs to be measurable, but the customer of the quality assurance process has to be able to define and validate the metrics used. In some cases, the customer has to also be able to rerun and validate the actual tests.

Our purpose in this book is to look at quality from the security testing perspective, and also to look at quality assurance definitions mainly from the third-party perspective. This, in most cases, means we are limited to black-box testing approaches.

3.3 Testing for Quality

Testing does not equal quality assurance. The main goal of testing is to minimize the number of flaws in released products. Testing is part of a typical quality assurance process, but there are many other steps before we get to testing. Understanding different quality assurance methods requires us to understand the different steps in the software development life cycle (SDLC). There have been many attempts to describe software development processes, such as the waterfall approach, iterative development, and component-based development.

3.3.1 V-Model

V-model is not necessarily the most modern approach to describing a software development process. In real life, software development rarely follows such a straightforward process. For us, the V-model still offers an interesting view of the testing side of things in the SDLC. Analyzing the software development from simple models is useful no matter what software development process is used. The same functional methods and tools are used in all software development including agile methods and spiral software development processes.

The traditional V-model is a very simplified graphical view of typical software development practices. It maps the traditional waterfall development model into various steps of testing. Note that we are not promoting the V-model over any other software development model. You should not use the V-model in your real-life software development without careful consideration. Let's analyze the steps in typical V-model system development, shown in Figure 3.1.

The phases on the left-hand side are very similar to the overly simplified schoolbook waterfall model of software development. It goes through the different steps, from gathering requirements to the various steps of design and finally to the programming phase. To us, the goal of the V-model is to enforce natural system boundaries at various steps and to enforce test-driven development at different levels of integration. The requirements step results in creation of the acceptance criteria used in acceptance testing. The first set of specifications describes the system at a high level and sets the functional criteria for system testing. Architectural design makes decisions on high-level integration of components that will be used to test against in integration testing. Finally, detailed design defines the most detailed testable units and the test criteria of unit testing. The V-model does not consider the different

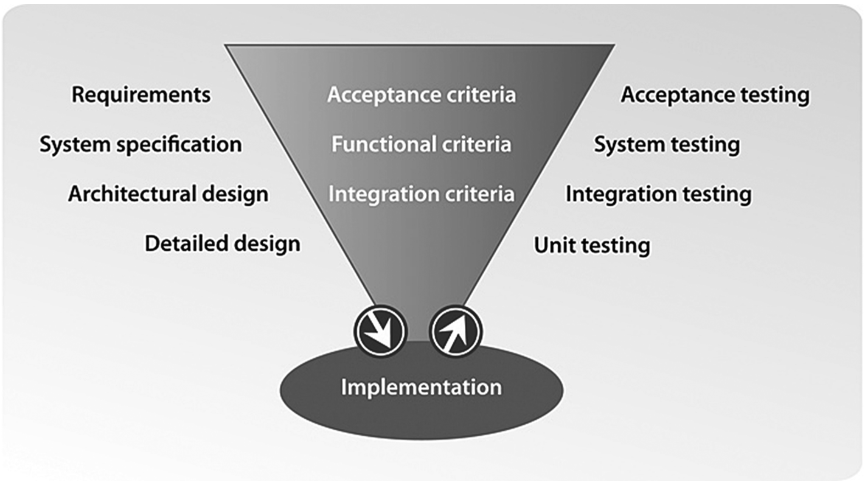

Figure 3.1 V-model for the system development life cycle.

purposes of testing; it only looks at the different levels of integration. There are many specifics missing from the V-model when viewed from the security testing perspective. Nevertheless, it is a good starting point when used in black-box testing processes.

3.3.2 Testing on the Developer's Desktop

Another set of quality assurance practices takes place even before we enter the testing phase inside a typical waterfall model of software development. A majority of bugs are caught in the programming phase. All the tools in the developer's desktop are tuned to catch human errors made during the programming and building phases. When code is submitted to building, it typically goes through rigorous code auditing. This can be either manual or automated. Also, manual programmers use fuzzing as part of unit testing.

3.3.3 Testing the Design

Software inspections and peer reviews are static analysis approaches to assessing various attributes in software development documentation and code. Verification of the design phase requires a formal review and complex automation tools. The formal methods employed can include mathematical proofs of encryption algorithms and analyses of the message flows used. For example, when a new protocol specification is being designed, the dynamic operation of the protocol message exchange needs to be carefully analyzed from the security perspective.

State machines in complex interfaces can also act as a source of information for black-box testing. Test automation can help in trying out a complex state machine to ensure that there are no deadlocks or unhandled exceptional situations.[4]

3.4 Main Categories of Testing

Software quality assurance techniques such as testing can be based on either static analysis or dynamic analysis. Static analysis is off-line analysis that is done to the source code without any requirement to run the code. Dynamic analysis is a runtime method that is performed while the software is executing. A good test process can combine both of these approaches. For example, code optimization tools can augment the code when the code is executed with information that can later be used in a static analysis. There are many different ways that the methods of testing can be partitioned.

3.4.1 Validation Testing Versus Defect Testing

Sommerville[5] (2004) divides testing into validation testing and defect testing. The purpose of validation testing is to show that the software functions according to

[4]An example of test automation framework that generates test cases from a state chart is the Conformiq Test Generator. www.conformiq.com/

[5]Ian Sommerville. *Software Engineering*, 8th ed. New York: Addison Wesley, 2006.

user requirements. On the other hand, defect testing intends to uncover flaws in the software rather than simulate its operational use. Defect testing aims at finding inconsistencies between the system and its specification.

3.4.2 Structural Versus Functional Testing

Another division of testing is based on access to the source code. These two categories are structural testing and functional testing.

Structural testing, or white-box testing, uses access to the source code to reveal flaws in the software. Structural testing techniques can also be used to test the object code. Structural testing can be based on either static or dynamic analysis, or their combination. The focus is on covering the internals of the product in detail. Various source code coverage techniques are used to analyze the depth of structural testing. One example of white-box testing is called unit testing, which concentrates on testing each of the functions as you see them in the code.

Functional testing, or black-box testing, tests the software through external interfaces. Functional testing is always dynamic and is designed on the basis of various specification documents produced in different phases of the software development process. A functional tester does not necessarily need to know the internals of the software. Access to the code is unnecessary, although it can be helpful in designing the tests.

Finally, gray-box testing is a combination of both the white-box and black-box approaches, and it uses the internals of the software to assist in the design of the tests of the external interfaces.

3.5 White-Box Testing

White-box testing has the benefit of having access to the code. In principle, the only method of reaching 100% coverage (of some sort) in testing is with white-box techniques. Different white-box testing techniques can be used to catch suspicious code during the programming phase and also while the code is being executed. We will next look at some relevant aspects of white-box testing techniques.

3.5.1 Making the Code Readable

A prerequisite for catching problems is to make the code more readable and thereby easier to understand and debug. Good programming practices and coding conventions can help in standardizing the code, and they will also help in implementing various automated tools in the validation process. An example of such quality improvements is compile-time checks, which will detect use of insecure function calls and structures.

3.5.2 Inspections and Reviews

Static analysis methods, such as various types of inspections and reviews, are widely used, and they are critical to the development of good-quality software. Inspections can focus on software development documents or the actual code. A requirement for successful inspections and reviews is agreeing on a policy on how the code should be implemented. Several industry standards from bodies like IEEE have defined guidelines on how and where inspections and reviews should be implemented.

3.5.3 Code Auditing

The simplest form of white-box testing is code auditing. Some people are more skilled at noticing flaws in code, including security mistakes, than others. From the security perspective, the most simple code auditing tools systematically search the code looking for vulnerable functions, such as sprintf(), strcpy(), gets(), memcpy(), scanf(), system(), and popen(), because they are often responsible for overflow problems. Such a simplistic approach will necessarily reveal many false positives because these functions can be used safely. More complex auditing tools analyze the entire program's structure, have models that represent common programming errors, and compare the structure of the program to these models. Such tools will greatly reduce the number of false positives as instead of just reporting the use of 'strcpy,' it analyzes whether the input has been limited to it.

A code review can take place either off-line or during compilation. Some static analysis tools check the compiled result of the code, analyzing weaknesses in the assembly code generated during compilation of the software module. Compilers themselves are also integrated with various quality-aware functionalities, issuing warnings when something suspicious is seen in the code or in an intermediate representation. As mentioned above, the most common problem encountered with code auditing tools is the number of false-positive issues, which are security warnings that do not pose a security risk. Another problem with all code-auditing practices is that they can only find problems they are taught to find. For example, the exploitable security flaw in the following code snippet from an X11 bitmap-handling routine might easily be missed by even the most skilled code auditing people and tools:

```
/ *Copyright 1987, 1998 The Open Group – Shortened for presentation!
  * Code to read bitmaps from disk files. Interprets
  * data from X10 and X11 bitmap files and creates
  * Pixmap representations of files.
  * Modified for speedup by Jim Becker, changed image
  * data parsing logic (removed some fscanf()s). Aug 5, 1988 */

int XReadBitmapFileData (_Xconst char *filename,
  unsigned int *width,          / *RETURNED */
  unsigned int *height,         / *RETURNED */
  unsigned char **data,         / *RETURNED */
  int *x_hot,                   / *RETURNED */
  int *y_hot)                   / *RETURNED */

  unsigned char *bits = NULL;   / *working variable */
  int size;                     / *number of data bytes */
  int padding;                  / *to handle alignment */
int bytes_per_line;          / *per scanline of data */
unsigned int ww = 0;         / *width */
unsigned int hh = 0;         / *height */

while (fgets(line, MAX_SIZE, file)) {
if (strlen(line) == MAX_SIZE-1) {
  RETURN (BitmapFileInvalid);
}
if (sscanf(line,"#define %s %d",name_and_type,&value) == 2) {
  if (!(type = strrchr(name_and_type, '_')))
    type = name_and_type;
  else
    type++;

  if (!strcmp("width", type))
    ww = (unsigned int) value;
  if (!strcmp("height", type))
    hh = (unsigned int) value;
  continue;
}

if (sscanf(line, "static short %s = {", name_and_type) == 1)
  version10p = 1;
else if (sscanf(line,"static unsigned char %s = {",name_and_type)
== 1)
  version10p = 0;
else if (sscanf(line, "static char %s = {", name_and_type) == 1)
  version10p = 0;
else
  continue;

if (!(type = strrchr(name_and_type, ‘_’)))
  type = name_and_type;
else
  type++;

if (strcmp("bits[]", type))
  continue;

if (!ww __ !hh)
  RETURN (BitmapFileInvalid);

if ((ww % 16) && ((ww % 16) < 9) && version10p)
    padding = 1;
else
    padding = 0;

64   bytes_per_line = (ww+7)/8 + padding;
65
66   size = bytes_per_line * hh;
67   bits = (unsigned char *) Xmalloc ((unsigned int) size);
68   if (!bits)
69     RETURN (BitmapNoMemory);
n    /* ... */
n+1  *data = bits;
n+2  *width = ww;
n+3  *height = hh;
n+4
n+5  return (BitmapSuccess);
n+6  }
```

The simple integer overflow flaw in this example is on line 64:

```
bytes_per_line = (ww+7)/8 + padding;
```

This integer overflow bug does not have an adverse effect on the library routine itself. However, when returned dimensions of width and height do not agree with actual data available, this may cause havoc among downstream consumers of data provided by the library. This indeed took place in most popular web browsers on the market and was demonstrated with PROTOS file fuzzers in July 2002 to be exploitable beyond "denial of service."[6] Full control over the victim's browser was gained over a remote connection. The enabling factor for full exploitability was conversion of library routine provided image data from row-first format into column-first format. When width of the image (ww) is in the range of (MAX_UINT-6). .MAX_UINT, i.e., 4294967289 . . . 4294967295 on 32-bit platforms, the calculation overflows back into a small integer.

This example is modified from X11 and X10 Bitmap handling routines that were written as part of the X Windowing System library in 1987 and 1988, over 20 years ago. Since then, this image format has refused to go away, and code to handle it is widely deployed in open source software and even on proprietary commercial platforms. Most implementations have directly adopted the original implementation, and the code has been read, reviewed, and integrated by thousands of skilled programmers. Very persistently, this flaw keeps reappearing in modern software.

3.6 Black-Box Testing

Testing will always be the main software verification and validation technique, although static analysis methods are useful for improving the overall quality of documentation and source code. When testing a live system or its prototype, real data is sent to the target and the responses are compared with various test criteria to assess the test verdict. In black-box testing, access to the source code is not necessary,

[6]The PROTOS file fuzzers are one of the many tools that were never released by University of Oulu, but the same functionality was included in the Codenomicon Images suite of fuzzing tools, released in early 2003.

although it will help in improving the tests. Black-box testing is sometimes referred to as functional testing, but for the scope of this book this definition can be misleading. Black-box testing can test more than just the functionality of the software.

3.6.1 Software Interfaces

In black-box testing, the system under test is tested through its interfaces. Black-box testing is built on the expected (or nonexpected) responses to a set of inputs fed to the software through selected interfaces. As mentioned in Chapter 1, the interfaces to a system can consist of, for example,

- User interfaces: GUI, command line;
- Network protocols;
- Data structures such as files;
- System APIs such as system calls and device drivers.

These interfaces can be further broken down into actual protocols or data structures.

3.6.2 Test Targets

Black-box testing can have different targets. The various names of test targets include, for example,

- Implementation under test (IUT);
- System under test (SUT);
- Device under test (DUT).

The test target can also be a subset of a system, such as:

- Function or class;
- Software module or component;
- Client or server implementation;
- Protocol stack or parser;
- Hardware such as network interface card (NIC);
- Operating system.

The target of testing can vary depending on the phase in the software development life cycle. In the earlier phases, the tests can be first targeted to smaller units such as parsers and modules, whereas in later phases the target can be a complete network-enabled server farm augmented with other infrastructure components.

3.6.3 Fuzz Testing as a Profession

We have had discussions with various fuzzing specialists with both QA and VA background, and this section is based on the analysis of those interviews. We will look at the various tasks from the perspectives of both security and testing professions. Let's start with security.

Typically, fuzzing first belongs to the security team. At a software development organization, the name of this team can be, for example, Product Security Team (PST for short). Risk assessment is one of the tools in deciding where to fuzz and what to fuzz, or if to fuzz at all. Security teams are often very small and very rarely have any budget for tool purchases. They depend on the funding from product development organizations. Although fuzzing has been known in QA for decades, the push to introduce it into development has almost always come from the security team, perhaps inspired by the increasing security alerts in its own products or perhaps by new knowledge from books like this. Initially, most security organizations depend on consultative fuzzing, but very fast most interviewed security experts claimed that they have turned almost completely toward in-house fuzzing. The primary reason usually is that buying fuzzing from consultative sources almost always results in unmaintained proprietary fuzzers and enormous bills for services that seem to be repeating themselves each time. Most security people will happily promote fuzzing tools into the development organization, but many of them want to maintain control on the chosen tools and veto right on consultative services bought by the development groups. This brings us to taking a closer look at the testing organization.

The example testing organization we will explore here is divided into three segments. One-fourth of the people are focused on tools and techniques, which we will call T&T. And one-fourth is focused on quality assurance processes, which we will call QAP. The remaining 50% of testers work for various projects in the product lines, with varying size teams depending on the project sizes. These will be referred to as product line testing (PLT) in this text.

The test specialists from the tools and techniques (T&T) division each have focus on one or more specific testing domains. For example, one dedicated team can be responsible for performance testing and another on the automated regression runs. One of the teams is responsible for fuzz testing and in supporting the projects with their fuzzing needs. The same people who are responsible for fuzzing can also take care of the white-box security tools. The test specialist can also be a person in the security auditing team outside the traditional QA organization.

But before any fuzzing tools are integrated into the quality assurance processes, the requirement needs to come from product management, and the integration of the new technique has to happen in cooperation with the QAP people. The first position in the QA process often is not the most optimal one, and therefore the QAP people need to closely monitor and improve the tactics in testing. The relationship with security auditors is also a very important task to the QAP people, as every single auditing or certification process will immediately become very expensive unless the flaw categories discovered in third-party auditing are already sought after in the QA process.

The main responsibility for fuzzing is on each individual project manager from PLT who is responsible for both involving fuzzing into his or her project, and in reporting the fuzzing results to the customer. PLT is almost always also responsible for the budget and will need to authorize all product purchases.

When a new fuzzing tool is being introduced to the organization, the main responsibility for tool selection should still be on the shoulders of the lead test specialist responsible for fuzzing. If the tool is the first in some category of test automation, a new person is appointed as the specialist. Without this type of assignment, the purchases of fuzzing tools will go astray very fast, with the decisions being made not on the actual quality and efficiency of the tools but on some other criteria such as vendor relations or marketing gimmicks. And this is not beneficial to the testing organization. Whereas it does not matter much which performance testing suite is used, fuzzing tools are very different from their efficiency perspective. A bad fuzzer is simply just money and time thrown away.

Let's next review some job descriptions in testing:

- QA Leader: Works for PLT in individual QA projects and selects the used processes and tools based on company policies and guidelines. The QA leader does the test planning, resourcing, staffing, and budget and is typically also responsible for the QA continuity, including transition of test plans between various versions and releases. One goal can include integration of test automation and review of the best practices between QA teams.
- QA Technical Leader: Works for T&T-related tasks. He or she is responsible for researching new tools and best practices of test automation, and doing tool recommendations. That can include test product comparisons either with third parties or together with customers. The QA technical leader can also be responsible for building in-house tools and test script that pilot or enable integration of innovative new ideas and assisting the PLT teams in understanding the test technologies, including training the testers in the tools. The QA technical leader can assist QA leader in performing ROI analysis of new tools and techniques and help with test automation integration either directly or through guidelines and step-by-step instructions. He or she can either perform the risk assessments with the product-related QA teams, or can recommend outsourced contractors that can perform those.
- Test Automation Engineer: Builds the test automation harnesses, which can involve setting up the test tools, building scripts for nightly and weekly tests, and keeping the regression tests up-to-date. In some complex environments, the setting up of the target system can be assigned to the actual developers or to the IT staff. The test automation engineer will see that automated test executions are progressing as planned, and that the failures are handled and that the test execution does not stop for any reason. All monitors and instruments are also critical in those tasks.
- Test Engineer/Designer: These are sometimes also called manual testers, although that is becoming more rare. The job of a test engineer can vary from building test cases for use in conformance and performance testing to selecting the "templates" that are used in fuzzing, if a large number of templates is required. Sometimes when manual tasks are required, the test engineer/designer babysits the test execution to see that it progresses as planned—for example, by pressing a key every hour. Most test automation tools are designed to eliminate manual testing.

3.7 Purposes of Black-Box Testing

Black-box testing can have the following general purposes:

- Feature or conformance testing;
- Interoperability testing;

- Performance testing;
- Robustness testing.

We will next examine each of these in more detail.

3.7.1 Conformance Testing

The first category of black-box testing is feature testing or conformance testing. The earliest description of the software being produced is typically contained in the requirements specification. The requirements specification of a specific software project can also be linked to third-party specifications, such as interface definitions and other industry standards. In quality assurance processes, the people responsible for validation evaluate the resulting software product against these specifications and standards. Such a process aims at validating the conformity of the product against the specifications. In this book we use the term *conformance testing* for all testing that validates features or functional requirements, no matter when or where that testing actually takes place.

3.7.2 Interoperability Testing

The second testing category is interoperability testing. Interoperability is basically a subset of conformance testing. Interoperability testing is a practical task in which the final product or its prototype is tried against other industry products. In real life, true conformance is very difficult if not impossible to reach. But, the product must at least be able to communicate with a large number of devices or systems. This type of testing can take place at various interoperability events, where software developers fight it out, in a friendly manner, to see who has the most conformant product. In some cases, if a dominant player decides to do it his or her own way, a method that complies with standards may not necessarily be the "correct" method. An industry standard can be defined by an industry forum or by an industry player who controls a major share of the industry. For example, if your web application does not work on the most widely used web browser, even if it is completely standards compliant, you will probably end up fixing the application instead of the browser vendor fixing the client software.

3.7.3 Performance Testing

The third type of testing, performance testing, comes from real-use scenarios of the software. When the software works according to the feature set and with other vendors' products, testers need to assess if the software is efficient enough in real-life use scenarios. There are different categories of tests for this purpose, including stress testing, performance testing, and load testing. In this book we use the term *performance testing* for all of these types of tests, whether they test strain conditions in the host itself or through a load over communication interfaces, and even if the test is done by profiling the efficiency of the application itself. All these tests aim at making the software perform "fast enough." The metric for final performance can be given as the number of requests or sessions per given time, the number of parallel users that can be served, or a number of other metrics. There are many types of throughput metrics that are relevant to some applications but not to others. Note that many performance metrics are often confused with quality-of-service metrics. Quality of service is, for the most part, a subset of performance, or at least it can be fixed by improving performance. Attacks that aim for denial of service are also typically exploiting performance issues in software.

3.7.4 Robustness Testing

The fourth black-box testing category is negative testing, or robustness testing. This is often the most important category from the security perspective. In negative testing, the software is tortured with semi-valid requests, and the reliability of the software is assessed. The sources of negative tests can come from the systems specifications, such as

- Requirement specifications: These are typically presented as "shall not" and "must not" requirements.
- System specifications: Physical byte boundaries, memory size, and other resource limits.
- Design specifications: Illegal state transitions and optional features.
- Interface specifications: Boundary values and blacklisted characters.
- Programming limitations: Programming language specific limits.

3.8 Testing Metrics

There is no single metric for black-box testing, but instead various metrics are needed with different testing approaches. At least three different levels of metrics are easily recognized:

- Specification coverage;
- Input space coverage;
- Attack surface coverage.

We will next give a brief overview of these, although they are explained in more detail in Chapter 4.

3.8.1 Specification Coverage

Specification coverage applies to all types of black-box testing. Tests can only be as good as the specification they are built from. For example, in Voice over IP (VoIP) testing, a testing tool has to cover about 10 different protocols with somewhere from one to 30 industry standard specifications for each protocol. A tool that covers only one specification has smaller test coverage than a tool that covers all of them. All tests have a specification, whether it is a text document, a machine-understandable interface model, or a capture of a test session that is then repeated and modified.