assessed and matured, which allows the organization to decide what maturity level makes sense for each practice.

NOTE You can find more information on SAMM at https://owaspsamm.org/model/.

Chapter Review

While there is no expectation that you, as a CISSP, will necessarily be involved in software development, you will almost certainly lead organizations that either produce software or consume it. Therefore, it is important that you understand how secure software is developed. Knowing this enables you to see what an organization is doing, software development-wise, and quickly get a sense of the maturity of its processes. If processes are ad hoc, this chapter should have given you some pointers on how to formalize the processes. After all, without formal processes and trained programmers in place, you have almost no hope of producing software that is not immediately vulnerable as soon as it is put into production. On the other hand, if the organization seems more mature, you can delve deeper into the specifics of building security into the software, which is the topic of the next chapter.

Quick Review

- The software development life cycle (SDLC) comprises five phases: requirements gathering, design, development, testing, and operations and maintenance (O&M).
- Computer-aided software engineering (CASE) refers to any type of software that allows for the automated development of software, which can come in the form of program editors, debuggers, code analyzers, version-control mechanisms, and more. The goals are to increase development speed and productivity and reduce errors.
- Various levels of testing should be carried out during development: unit (testing individual components), integration (verifying components work together in the production environment), acceptance (ensuring code meets customer requirements), and regression (testing after changes take place).
- Change management is a systematic approach to deliberately regulating the changing nature of projects. Change control, which is a subpart of change management, deals with controlling specific changes to a system.
- Security should be addressed in each phase of software development. It should not be addressed only at the end of development because of the added cost, time, and effort and the lack of functionality.

- The attack surface is the collection of possible entry points for an attacker. The reduction of this surface reduces the possible ways that an attacker can exploit a system.
- Threat modeling is a systematic approach used to understand how different threats could be realized and how a successful compromise could take place.
- The waterfall software development methodology follows a sequential approach that requires each phase to complete before the next one can begin.
- The prototyping methodology involves creating a sample of the code for proof-of-concept purposes.
- Incremental software development entails multiple development cycles that are carried out on a piece of software throughout its development stages.
- The spiral methodology is an iterative approach that emphasizes risk analysis per iteration.
- Rapid Application Development (RAD) combines prototyping and iterative development procedures with the goal of accelerating the software development process.
- Agile methodologies are characterized by iterative and incremental development processes that encourage team-based collaboration, where flexibility and adaptability are used instead of a strict process structure.
- Some organizations improve internal coordination and reduce friction by integrating the development and operations (DevOps) teams or the development, operations, and security (DevSecOps) teams when developing software.
- An integrated product team (IPT) is a multidisciplinary development team with representatives from many or all the stakeholder populations.
- Capability Maturity Model Integration (CMMI) is a process improvement approach that provides organizations with the essential elements of effective processes, which will improve their performance.
- The CMMI model uses six maturity levels designated by the numbers 0 through 5. Each level represents the maturity level of the process quality and optimization. The levels are organized as follows: 0 = Incomplete, 1 = Initial, 2 = Managed, 3 = Defined, 4 = Quantitatively Managed, 5 = Optimizing.
- The OWASP Software Assurance Maturity Model (SAMM) is specifically focused on secure software development and allows organizations to decide their target maturity levels within each of five critical business functions: Governance, Design, Implementation, Verification, and Operations.

Questions

Please remember that these questions are formatted and asked in a certain way for a reason. Keep in mind that the CISSP exam is asking questions at a conceptual level. Questions may not always have the perfect answer, and the candidate is advised against always looking for the perfect answer. Instead, the candidate should look for the best answer in the list.

1. The software development life cycle has several phases. Which of the following lists these phases in the correct order?

- **A.** Requirements gathering, design, development, maintenance, testing, release
- **B.** Requirements gathering, design, development, testing, operations and maintenance
- **C.** Prototyping, build and fix, increment, test, maintenance
- **D.** Prototyping, testing, requirements gathering, integration, testing

2. John is a manager of the application development department within his company. He needs to make sure his team is carrying out all of the correct testing types and at the right times of the development stages. Which of the following accurately describe types of software testing that should be carried out?

- **i.** **Unit testing** Testing individual components in a controlled environment where programmers validate data structure, logic, and boundary conditions
- **ii.** **Integration testing** Verifying that components work together as outlined in design specifications
- **iii.** **Acceptance testing** Ensuring that the code meets customer requirements
- **iv.** **Regression testing** After a change to a system takes place, retesting to ensure functionality, performance, and protection

- **A.** i, ii
- **B.** ii, iii
- **C.** i, ii, iv
- **D.** i, ii, iii, iv

3. Marge has to choose a software development methodology that her team should follow. The application that her team is responsible for developing is a critical application that can have few to no errors. Which of the following best describes the type of methodology her team should follow?

- **A.** Cleanroom
- **B.** Joint Application Development (JAD)
- **C.** Rapid Application Development (RAD)
- **D.** Reuse methodology

4. Which level of Capability Maturity Model Integration allows organizations to manage all projects across the organization and be proactive?
   - **A.** Defined
   - **B.** Incomplete
   - **C.** Managed
   - **D.** Optimizing
5. Mohammed is in charge of a large software development project with rigid requirements and phases that will probably be completed by different contractors. Which methodology would be best?
   - **A.** Waterfall
   - **B.** Spiral
   - **C.** Prototyping
   - **D.** Agile

Use the following scenario to answer Questions 6–9. You're in charge of IT and security at a midsize organization going through a growth stage. You decided to stand up your own software development team and are about to start your first project: a knowledge base for your customers. You think it can eventually grow to become the focal point of interaction with your customers, offering a multitude of features. You've heard a lot about the Scrum methodology and decide to try it for this project.

6. How would you go about documenting the requirements for this software system?
   - **A.** User stories
   - **B.** Use cases
   - **C.** System Requirements Specification (SRS)
   - **D.** Informally, since it's your first project
7. You are halfway through your first Scrum sprint and get a call from a senior vice president insisting that you add a new feature immediately. How do you handle this request?
   - **A.** Add the feature to the next sprint
   - **B.** Change the current sprint to include the feature
   - **C.** Reset the project to the requirements gathering phase
   - **D.** Delay the new feature until the end of the project

8. Your software development team, being new to the organization, is struggling to work smoothly with other teams within the organization as needed to get the software into production securely. Which approach can help mitigate this internal friction?
   A. DevSecOps
   B. DevOps
   C. Integrated Product Teams (IPT)
   D. Joint Analysis Design (JAD) sessions
9. What would be the best approach to selectively mature your software development practices with a view to improving cybersecurity?
   A. Software Assurance Maturity Model (SAMM)
   B. Capability Maturity Model Integration (CMMI)
   C. Kanban
   D. Integrated product teams (IPTs)

Answers

1. **B.** The following outlines the common phases of the software development life cycle:
   i. Requirements gathering
   ii. Design
   iii. Development
   iv. Testing
   v. Operations and maintenance
2. **D.** There are different types of tests the software should go through because there are different potential flaws to look for. The following are some of the most common testing approaches:
   - **Unit testing** Testing individual components in a controlled environment where programmers validate data structure, logic, and boundary conditions
   - **Integration testing** Verifying that components work together as outlined in design specifications
   - **Acceptance testing** Ensuring that the code meets customer requirements
   - **Regression testing** After a change to a system takes place, retesting to ensure functionality, performance, and protection
3. **A.** The listed software development methodologies and their definitions are as follows:
   - **Joint Application Development (JAD)** A methodology that uses a team approach in application development in a workshop-oriented environment.

- **Rapid Application Development (RAD)** A methodology that combines the use of prototyping and iterative development procedures with the goal of accelerating the software development process.
- **Reuse methodology** A methodology that approaches software development by using progressively developed code. Reusable programs are evolved by gradually modifying preexisting prototypes to customer specifications. Since the reuse methodology does not require programs to be built from scratch, it drastically reduces both development cost and time.
- **Cleanroom** An approach that attempts to prevent errors or mistakes by following structured and formal methods of developing and testing. This approach is used for high-quality and critical applications that will be put through a strict certification process.

4. A. The six levels of Capability Maturity Integration Model are

- **Incomplete** Development process is ad hoc or even chaotic. Tasks are not always completed at all, so projects are regularly cancelled or abandoned.
- **Initial** The organization does not use effective management procedures and plans. There is no assurance of consistency, and quality is unpredictable. Success is usually the result of individual heroics.
- **Managed** A formal management structure, change control, and quality assurance are in place for individual projects. The organization can properly repeat processes throughout each project.
- **Defined** Formal procedures are in place that outline and define processes carried out in all projects across the organization. This allows the organization to be proactive rather than reactive.
- **Quantitatively Managed** The organization has formal processes in place to collect and analyze quantitative data, and metrics are defined and fed into the process-improvement program.
- **Optimizing** The organization has budgeted and integrated plans for continuous process improvement, which allow it to quickly respond to opportunities and changes.

5. D. The Waterfall methodology is a very rigid approach that could be useful for projects in which all the requirements are fully understood up front or projects for which different organizations will perform the work at each phase. The Spiral, prototyping, and Agile methodologies are well suited for situations in which the requirements are not well understood, and don't lend themselves well to switching contractors midstream.

6. A. Any answer except "informally" would be a reasonable one, but since you are using an Agile methodology (Scrum), user stories is the best answer. The important point is that you document the requirements formally, so you can design a solution that meets all your users' needs.

7. **A.** The Scrum methodology allows the project to be reset by allowing product features to be added, changed, or removed at clearly defined points that typically happen at the conclusion of each sprint.
8. **A.** DevSecOps is the integration of development, security, and operations professionals into a software development team. This is a good way to solve the friction between developers and members of the security and operations staff.
9. **A.** CMMI and SAMM are the only maturity models among the possible answers. SAMM is the best answer because it allows for more granular maturity goals than CMMI does, and it is focused on security.

CHAPTER 25

Secure Software

This chapter presents the following:

- Programming languages
- Secure coding
- Security controls for software development
- Software security assessments
- Assessing the security of acquired software

A good programmer is someone who always looks both ways before crossing a one-way street.

—Doug Linder

Quality can be defined as fitness for purpose. In other words, quality refers to how good or bad something is for its intended purpose. A high-quality car is good for transportation. We don't have to worry about it breaking down, failing to protect its occupants in a crash, or being easy for a thief to steal. When we need to go somewhere, we can count on a high-quality car to get us to wherever we need to go. Similarly, we don't have to worry about high-quality software crashing, corrupting our data under unforeseen circumstances, or being easy for someone to subvert. Sadly, many developers still think of functionality first (or only) when thinking about quality. When we look at it holistically, we see that quality is the most important concept in developing secure software.

Every successful compromise of a software system relies on the exploitation of one or more vulnerabilities in it. Software vulnerabilities, in turn, are caused by defects in the design or implementation of code. The goal, then, is to develop software that is as free from defects or, in other words, as high quality as we can make it. In this chapter, we will discuss how secure software is quality software. We can't have one without the other. By applying the right processes, controls, and assessments, the outcome will be software that is more reliable and more difficult to exploit or subvert. Of course, these principles apply equally to software we develop in our own organizations and software that is developed for us by others.

Programming Languages and Concepts

All software is written in some type of programming language. Programming languages have gone through several generations over time, each generation building on the next, providing richer functionality and giving programmers more powerful tools as they evolve.

The main categories of languages are machine, assembly, high-level, very high-level, and natural languages. *Machine language* is in a format that the computer's processor can understand and work with directly. Every processor family has its own machine code instruction set, which is represented in a binary format (1 and 0) and is the most fundamental form of programming language. Since this was pretty much the only way to program the very first computers in the early 1950s, machine languages are the first generation of programming languages. Early computers used only basic binary instructions because compilers and interpreters were nonexistent at the time. Programmers had to manually calculate and allot memory addresses and sequentially feed instructions, as there was no concept of abstraction. Not only was programming in binary extremely time consuming, it was also highly prone to errors. (If you think about writing out thousands of 1's and 0's to represent what you want a computer to do, this puts this approach into perspective.) This forced programmers to keep a tight rein on their program lengths, resulting in programs that were very rudimentary.

An *assembly language* is considered a low-level programming language and is the symbolic representation of machine-level instructions. It is "one step above" machine language. It uses symbols (called mnemonics) to represent complicated binary codes. Programmers using assembly language could use commands like ADD, PUSH, POP, etc., instead of the binary codes (1001011010, etc.). Assembly languages use programs called *assemblers*, which automatically convert these assembly codes into the necessary machine-compatible binary language. To their credit, assembly languages drastically reduced programming and debugging times, introduced the concept of variables, and freed programmers from manually calculating memory addresses. But like machine code, programming in an assembly language requires extensive knowledge of a computer's architecture. It is easier than programming in binary format, but more challenging compared to the high-level languages most programmers use today.

Programs written in assembly language are also hardware specific, so a program written for an ARM-based processor would be incompatible with Intel-based systems; thus, these types of languages are not portable. Once the program is written, it is fed to an assembler, which translates the assembly language into machine language. The assembler also replaces variable names in the assembly language program with actual addresses at which their values will be stored in memory.

NOTE Assembly language allows for direct control of very basic activities within a computer system, as in pushing data on a memory stack and popping data off a stack. Attackers commonly use assembly language to tightly control how malicious instructions are carried out on victim systems.

The third generation of programming languages started to emerge in the early 1960s. They are known as *high-level languages* because of their refined programming structures.

High-level languages use abstract statements. Abstraction naturalizes multiple assembly language instructions into a single high-level statement, such as IF – THEN – ELSE. This allows programmers to leave low-level (system architecture) intricacies to the programming language and focus on their programming objectives. In addition, high-level languages are easier to work with compared to machine and assembly languages, as their syntax is similar to human languages. The use of mathematical operators also simplifies arithmetic and logical operations. This drastically reduces program development time and allows for more simplified debugging. This means the programs are easier to write and mistakes (bugs) are easier to identify. High-level languages are processor independent. Code written in a high-level language can be converted to machine language for different processor architectures using compilers and interpreters. When code is independent of a specific processor type, the programs are portable and can be used on many different system types.

Fourth-generation languages *(very high-level languages)* were designed to further enhance the natural language approach instigated within the third-generation languages. They focus on highly abstract algorithms that allow straightforward programming implementation in specific environments. The most remarkable aspect of fourth-generation languages is that the amount of manual coding required to perform a specific task may be ten times less than for the same task on a third-generation language. This is an especially important feature because these languages have been developed to be used by inexpert users and not just professional programmers.

As an analogy, let's say that you need to pass a calculus exam. You need to be very focused on memorizing the necessary formulas and applying the formulas to the correct word problems on the test. Your focus is on how calculus works, not on how the calculator you use as a tool works. If you had to understand how your calculator is moving data from one transistor to the other, how the circuitry works, and how the calculator stores and carries out its processing activities just to use it for your test, this would be overwhelming. The same is true for computer programmers. If they had to worry about how the operating system carries out memory management functions, input/output activities, and how processor-based registers are being used, it would be difficult for them to also focus on real-world problems they are trying to solve with their software. Very high-level languages hide all of this background complexity and take care of it for the programmer.

The early 1990s saw the conception of the fifth generation of programming languages *(natural languages)*. These languages approach programming from a completely different perspective. Program creation does not happen through defining algorithms and function statements, but rather by defining the constraints for achieving a specified result. The goal is to create software that can solve problems by itself instead of a programmer having to develop code to deal with individual and specific problems. The applications work more like a black box—a problem goes in and a solution comes out. Just as the introduction of assembly language eliminated the need for binary-based programming, the full impact of fifth-generation programming techniques may bring to an end the traditional programming approach. The ultimate target of fifth-generation languages is to eliminate the need for programming expertise and instead use advanced knowledge-based processing and artificial intelligence.

Language Levels

The "higher" the language, the more abstraction that is involved, which means the language hides details of how it performs its tasks from the software developer. A programming language that provides a high level of abstraction frees the programmer from the need to worry about the intricate details of the computer system itself, as in registers, memory addresses, complex Boolean expressions, thread management, and so forth. The programmer can use simple statements such as "print" and does not need to worry about how the computer will actually get the data over to the printer. Instead, the programmer can focus on the core functionality that the application is supposed to provide and not be bothered with the complex things taking place in the belly of the operating system and motherboard components.

As an analogy, you do not need to understand how your engine or brakes work in your car—there is a level of abstraction. You just turn the steering wheel and step on the pedal when necessary, and you can focus on getting to your destination.

There are so many different programming languages today, it is hard to fit them neatly in the five generations described in this chapter. These generations are the classical way of describing the differences in software programming approaches and what you will see on the CISSP exam.

The industry has not been able to fully achieve all the goals set out for these fifth-generation languages. The human insight of programmers is still necessary to figure out the problems that need to be solved, and the restrictions of the structure of a current computer system do not allow software to "think for itself" yet. We are getting closer to achieving artificial intelligence within our software, but we still have a long way to go.

The following lists the basic software programming language generations:

- **Generation one** Machine language
- **Generation two** Assembly language
- **Generation three** High-level language
- **Generation four** Very high-level language
- **Generation five** Natural language

Assemblers, Compilers, Interpreters

No matter what type or generation of programming language is used, all of the instructions and data have to end up in a binary format for the processor to understand and work with. Just like our food has to be broken down into specific kinds of molecules for our body to be able to use it, all code must end up in a format that is consumable by specific systems. Each programming language type goes through this transformation through the use of assemblers, compilers, or interpreters.

Assemblers are tools that convert assembly language source code into machine language code. Assembly language consists of mnemonics, which are incomprehensible to processors and therefore need to be translated into operation instructions.

Compilers are tools that convert high-level language statements into the necessary machine-level format (.exe, .dll, etc.) for specific processors to understand. The compiler transforms instructions from a source language (high-level) to a target language (machine), sometimes using an external assembler along the way. This transformation allows the code to be executable. A programmer may develop an application in the C language, but when you purchase this application, you do not receive the source code; instead, you receive the executable code that runs on your type of computer. The source code was put through a compiler, which resulted in an executable file that can run on your specific processor type.

Compilers allow developers to create software code that can be developed once in a high-level language and compiled for various platforms. So, you could develop one piece of software, which is then compiled by five different compilers to allow it to be able to run on five different systems.

Figure 25-1 shows the process by which a high-level language is gradually transformed into machine language, which is the only language a processor can understand natively. In this example, we have a statement that assigns the value 42 to the variable *x*. Once we feed the program containing this statement to a compiler, we end up with assembly language, which is shown in the middle of the figure. The way to set the value of a variable in assembly language is to literally move that value into wherever the variable is being stored. In this example, we are moving the hexadecimal value for 42 (which is 2a in hexadecimal, or 2ah) into the ax register in the processor. In order for the processor to execute this command, however, we still have to convert it into machine language, which is the job of the assembler. Note that it is way easier for a human coder to write *x* = 42 than it is to represent the same operation in either assembly or (worse yet) machine language.

If a programming language is considered "interpreted," then a tool called an *interpreter* takes care of transforming high-level code to machine-level code. For example, applications that are developed in JavaScript, Python, or Perl can be run directly by an interpreter, without having to be compiled. The goal is to improve portability. The greatest advantage of executing a program in an interpreted environment is that the platform independence and memory management functions are part of an interpreter. The major disadvantage

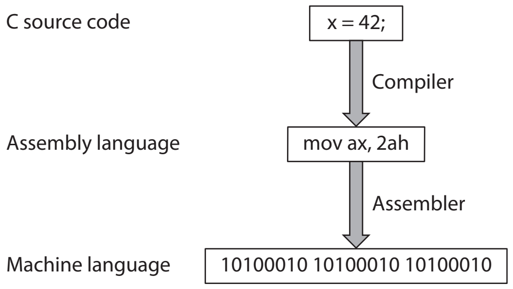


Figure 25-1 Converting a high-level language statement into machine language code

with this approach is that the program cannot run as a stand-alone application, requiring the interpreter to be installed on the local machine.

NOTE Some languages, such as Java and Python, blur the lines between interpreted and compiled languages by supporting both approaches. We'll talk more about how Java does this in the next section.

From a security point of view, it is important to understand vulnerabilities that are inherent in specific programming languages. For example, programs written in the C language could be vulnerable to buffer overrun and format string errors. The issue is that some of the C standard software libraries do not check the length of the strings of data they manipulate by default. Consequently, if a string is obtained from an untrusted source (i.e., the Internet) and is passed to one of these library routines, parts of memory may be unintentionally overwritten with untrustworthy data—this vulnerability can potentially be used to execute arbitrary and malicious software. Some programming languages, such as Java, perform automatic memory allocation as more space is needed; others, such as C, require the developer to do this manually, thus leaving opportunities for error.

Garbage collection is an automated way for software to carry out part of its memory management tasks. A *garbage collector* identifies blocks of memory that were once allocated but are no longer in use and deallocates the blocks and marks them as free. It also gathers scattered blocks of free memory and combines them into larger blocks. It helps provide a more stable environment and does not waste precious memory. If garbage collection does not take place properly, not only can memory be used in an inefficient manner, an attacker could carry out a denial-of-service attack specifically to artificially commit all of a system's memory, rendering the system unable to function.

Nothing in technology seems to be getting any simpler, which makes learning this stuff much harder as the years go by. Ten years ago assembly, compiled, and interpreted languages were more clear-cut and their definitions straightforward. For the most part, only scripting languages required interpreters, but as languages have evolved they have become extremely flexible to allow for greater functionality, efficiency, and portability. Many languages can have their source code compiled or interpreted depending upon the environment and user requirements.

Runtime Environments

What if you wanted to develop software that could run on many different environments without having to recompile it? This is known as *portable code* and it needs something that can sort of "translate" it to each different environment. That "translator" could be tuned to a particular type of computer but be able to run any of the portable code it understands. This is the role of *runtime environments (RTEs)*, which function as miniature operating systems for the program and provide all the resources portable code needs. One of the best examples of RTE usage is the Java programming language.

Java is platform independent because it creates intermediate code, *bytecode*, which is not processor-specific. The *Java Virtual Machine (JVM)* converts the bytecode to the machine

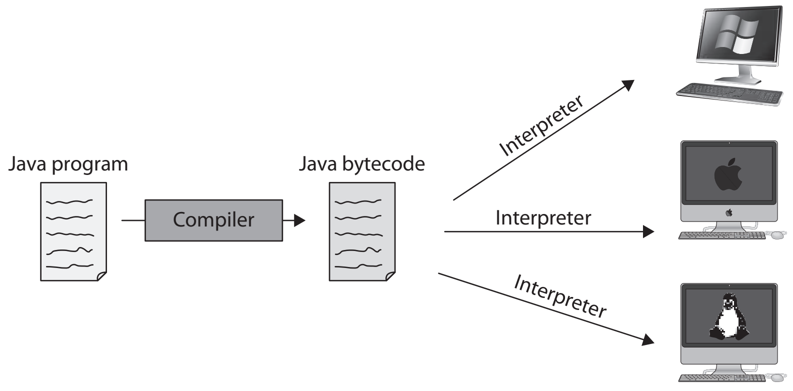


Figure 25-2 The JVM interprets bytecode to machine code for that specific platform.

code that the processor on that particular system can understand (see Figure 25-2). Despite its name, the JVM is not a full-fledged VM (as defined in Chapter 7). Instead, it is a component of the Java RTE, together with a bunch of supporting files like class libraries.

Let's quickly walk through these steps:

1. A programmer creates a Java applet and runs it through a compiler.
2. The Java compiler converts the source code into bytecode (not processor-specific).
3. The user downloads the Java applet.
4. The JVM converts the bytecode into machine-level code (processor-specific).
5. The applet runs when called upon.

When an applet is executed, the JVM creates a unique RTE for it called a *sandbox*. This sandbox is an enclosed environment in which the applet carries out its activities. Applets are commonly sent over within a requested web page, which means the applet executes as soon as it arrives. It can carry out malicious activity on purpose or accidentally if the developer of the applet did not do his part correctly. So the sandbox strictly limits the applet's access to any system resources. The JVM mediates access to system resources to ensure the applet code behaves and stays within its own sandbox. These components are illustrated in Figure 25-3.

NOTE The Java language itself provides protection mechanisms, such as garbage collection, memory management, validating address usage, and a component that verifies adherence to predetermined rules.

However, as with many other things in the computing world, the bad guys have figured out how to escape the confines and restrictions of the sandbox. Programmers have figured out how to write applets that enable the code to access hard drives and

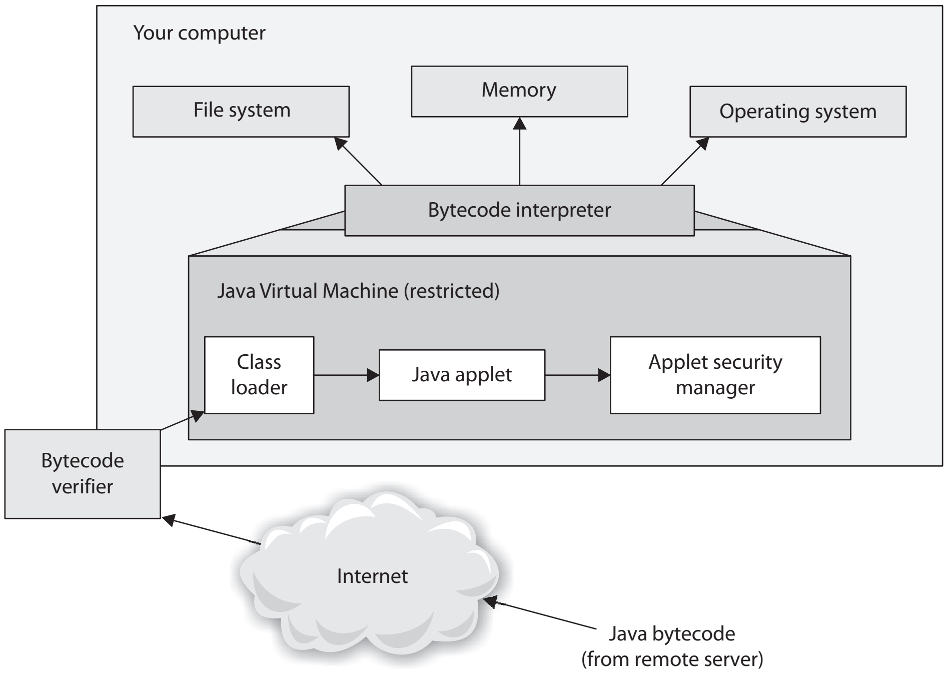
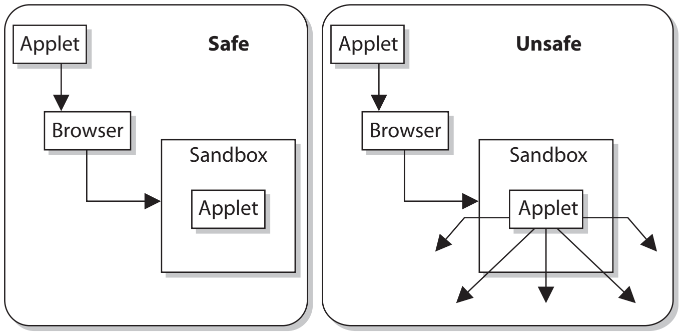


Figure 25-3 Java's security model

resources that are supposed to be protected by the Java security scheme. This code can be malicious in nature and cause destruction and mayhem to the user and her system.



Object-Oriented Programming Concepts

Software development used to be done by classic input–processing–output methods. This development used an information flow model from hierarchical information structures. Data was input into a program, and the program passed the data from the beginning to end, performed logical procedures, and returned a result. *Object-oriented programming (OOP)* methods perform the same functionality, but with different techniques that work in a more efficient manner. First, you need to understand the basic concepts of OOP.

OOP works with classes and objects. A real-world object, such as a table, is a member (or an instance) of a larger class of objects called "furniture." The furniture class has a set of attributes associated with it, and when an object is generated, it inherits these attributes. The attributes may be color, dimensions, weight, style, and cost. These attributes apply if a chair, table, or loveseat object is generated, also referred to as *instantiated*. Because the table is a member of the class furniture, the table inherits all attributes defined for the class (see Figure 25-4).

The programmer develops the class and all of its characteristics and attributes. The programmer does not develop each and every object, which is the beauty of this approach. As an analogy, let's say you developed an advanced coffee maker with the goal of putting Starbucks out of business. A customer punches the available buttons on your coffee maker interface, ordering a large latte, with skim milk, vanilla and raspberry flavoring, and an extra shot of espresso, where the coffee is served at 250 degrees. Your coffee maker does all of this through automation and provides the customer with a lovely cup of coffee exactly to her liking. The next customer wants a mocha Frothy Frappé, with whole milk and extra foam. So the goal is to make something once (coffee maker, class), allow it to accept requests through an interface, and create various results (cups of coffee, objects) depending upon the requests submitted.

But how does the class create objects based on requests? A piece of software that is written in OOP will have a request sent to it, usually from another object. The requesting object wants a new object to carry out some type of functionality. Let's say that object A wants object B to carry out subtraction on the numbers sent from A to B. When this request comes in, an object is built (instantiated) with all of the necessary programming code. Object B carries out the subtraction task and sends the result back to object A.

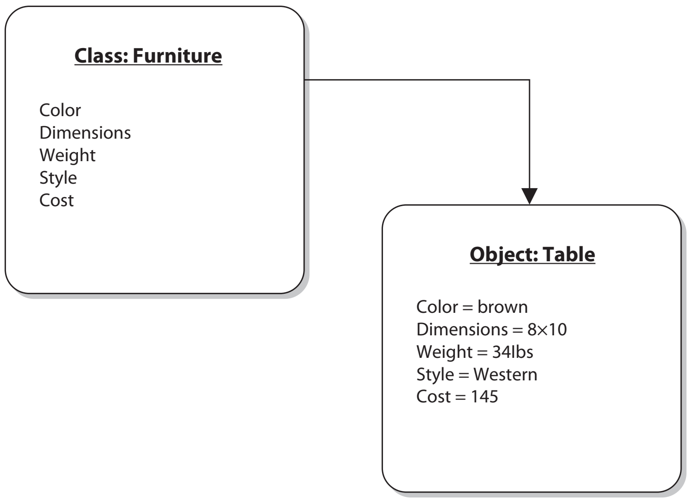
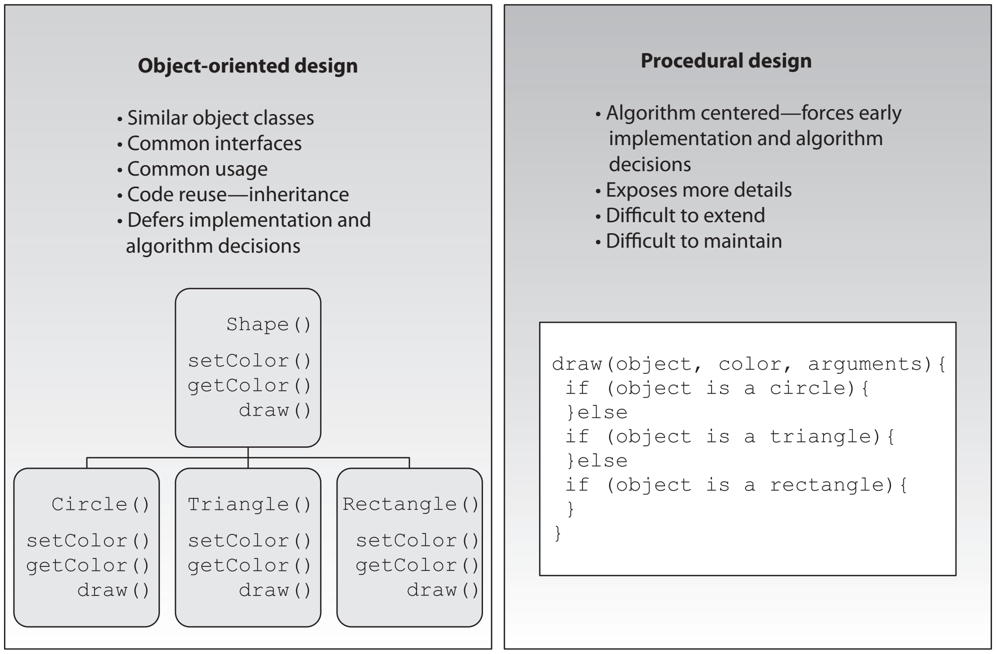


Figure 25-4
In object-oriented inheritance, each object belongs to a class and takes on the attributes of that class

It does not matter what programming language the two objects are written in; what matters is if they know how to communicate with each other. One object can communicate with another object if it knows the application programming interface (API) communication requirements. An API is the mechanism that allows objects to talk to each other (as described in depth in the forthcoming section "Application Programming Interfaces"). Let's say you want to talk to Jorge, but can only do so by speaking French and can only use three phrases or less, because that is all Jorge understands. As long as you follow these rules, you can talk to Jorge. If you don't follow these rules, you can't talk to Jorge.

TIP An object is an instance of a class.

What's so great about OOP? Figure 25-5 shows the difference between OOP and procedural programming, which is a non-OOP technique. Procedural programming is built on the concept of dividing a task into procedures that, when executed, accomplish the task. This means that large applications can quickly become one big pile of code (sometimes called *spaghetti code*). If you want to change something in this pile, you have to go through all the program's procedures to figure out what your one change is going to break. If the program contains hundreds or thousands of lines of code, this is not an easy or enjoyable task. Now, if you choose to write your program in an object-oriented


Object-oriented design
• Similar object classes
• Common interfaces
• Common usage
• Code reuse—inheritance
• Defers implementation and algorithm decisions
Shape() setColor() getColor() draw()
Circle() setColor() getColor() draw()
Triangle() setColor() getColor() draw()
Rectangle() setColor() getColor() draw()
Procedural design
• Algorithm centered—forces early implementation and algorithm decisions
• Exposes more details
• Difficult to extend
• Difficult to maintain

```
draw(object, color, arguments){
 if (object is a circle){
 }else
 if (object is a triangle){
 }else
 if (object is a rectangle){
 }
}
```

Figure 25-5 Procedural vs. object-oriented programming

language, you don't have one monolithic application, but an application that is made up of smaller components (objects). If you need to make changes or updates to some functionality in your application, you can just change the code within the class that creates the object carrying out that functionality, and you don't have to worry about everything else the program actually carries out. The following breaks down the benefits of OOP:

- **Modularity** The building blocks of software are autonomous objects, cooperating through the exchange of messages.
- **Deferred commitment** The internal components of an object can be redefined without changing other parts of the system.
- **Reusability** Classes are reused by other programs, though they may be refined through inheritance.
- **Naturalness** Object-oriented analysis, design, and modeling map to business needs and solutions.

Most applications have some type of functionality in common. Instead of developing the same code to carry out the same functionality for ten different applications, using OOP allows you to create the object only once and reuse it in other applications. This reduces development time and saves money.

Now that we've covered the concepts of OOP, let's clarify the terminology. A *method* is the functionality or procedure an object can carry out. An object may be constructed to accept data from a user and to reformat the request so a back-end server can understand and process it. Another object may perform a method that extracts data from a database and populates a web page with this information. Or an object may carry out a withdrawal procedure to allow the user of an ATM to extract money from her account.

The objects *encapsulate* the attribute values, which means this information is packaged under one name and can be reused as one entity by other objects. Objects need to be able to communicate with each other, and this happens by using *messages* that are sent to the receiving object's API. If object A needs to tell object B that a user's checking account must be reduced by $40, it sends object B a message. The message is made up of the destination, the method that needs to be performed, and the corresponding arguments. Figure 25-6 shows this example.

Messaging can happen in several ways. A given object can have a single connection (one-to-one) or multiple connections (one-to-many). It is important to map these communication paths to identify if information can flow in a way that is not intended. This helps to ensure that sensitive data cannot be passed to objects of a lower security level.

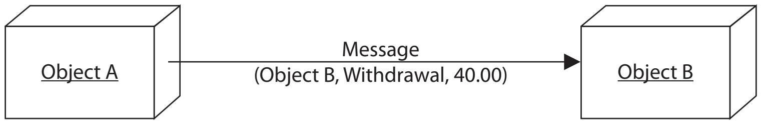


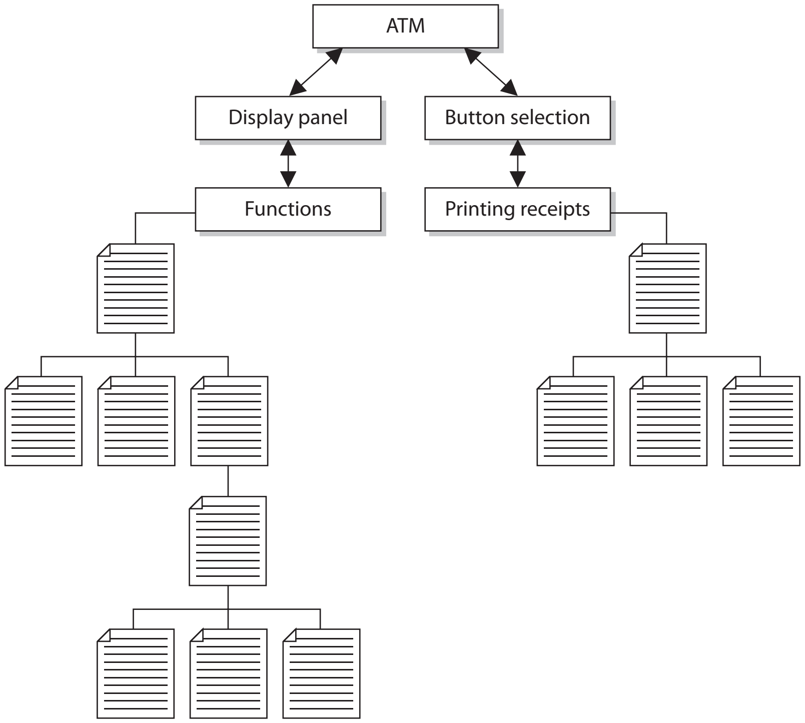
Figure 25-6 Objects communicate via messages.

An object can have a shared portion and a private portion. The *shared* portion is the interface (API) that enables it to interact with other components. Messages enter through the interface to specify the requested operation, or method, to be performed. The *private* portion of an object is how it actually works and performs the requested operations. Other components need not know how each object works internally—only that it does the job requested of it. This is how *data hiding* is possible. The details of the processing are hidden from all other program elements outside the object. Objects communicate through well-defined interfaces; therefore, they do not need to know how each other works internally.

NOTE Data hiding is provided by encapsulation, which protects an object's private data from outside access. No object should be allowed to, or have the need to, access another object's internal data or processes.

These objects can grow to great numbers, so the complexity of understanding, tracking, and analyzing can get a bit overwhelming. Many times, the objects are shown in connection to a reference or pointer in documentation. Figure 25-7 shows how related objects are represented as a specific piece, or reference, in a bank ATM system. This enables analysts and developers to look at a higher level of operation and procedures without having to view each individual object and its code. Thus, this modularity provides for a more easily understood model.

Figure 25-7 Object relationships within a program



Abstraction, as discussed earlier, is the capability to suppress unnecessary details so the important, inherent properties can be examined and reviewed. It enables the separation of conceptual aspects of a system. For example, if a software architect needs to understand how data flows through the program, she would want to understand the big pieces of the program and trace the steps the data takes from first being input into the program all the way until it exits the program as output. It would be difficult to understand this concept if the small details of every piece of the program were presented. Instead, through abstraction, all the details are suppressed so the software architect can understand a crucial part of the product. It is like being able to see a forest without having to look at each and every tree.

Each object should have specifications it adheres to. This discipline provides cleaner programming and reduces programming errors and omissions. The following list is an example of what should be developed for each object:

- Object name
- Attribute descriptions
- Attribute name
- Attribute content
- Attribute data type
- External input to object
- External output from object
- Operation descriptions
- Operation name
- Operation interface description
- Operation processing description
- Performance issues
- Restrictions and limitations
- Instance connections
- Message connections

The developer creates a class that outlines these specifications. When objects are instantiated, they inherit these attributes.

Each object can be reused as stated previously, which is the beauty of OOP. This enables a more efficient use of resources and the programmer's time. Different applications can use the same objects, which reduces redundant work, and as an application grows in functionality, objects can be easily added and integrated into the original structure.

The objects can be catalogued in a library, which provides an economical way for more than one application to call upon the objects (see Figure 25-8). The library provides an index and pointers to where the objects actually live within the system or on another system.

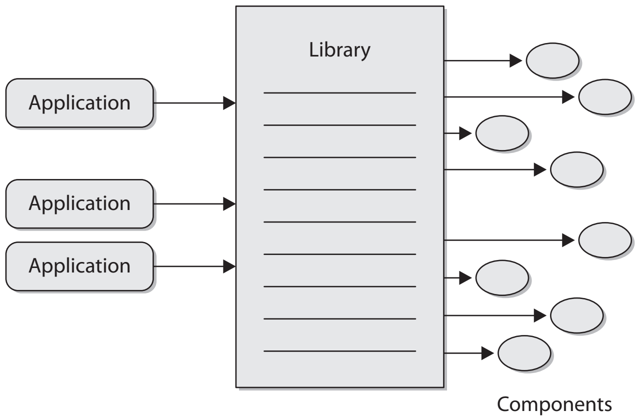


Figure 25-8 Applications locate the necessary objects through a library index.

When applications are developed in a modular approach, like object-oriented methods, components can be reused, complexity is reduced, and parallel development can be done. These characteristics allow for fewer mistakes, easier modification, resource efficiency, and more timely coding than the classic programming languages. OOP also provides functional independence, which means each module addresses a specific subfunction of requirements and has an interface that is easily understood by other parts of the application.

An object is *encapsulated*, meaning the data structure (the operation's functionality) and the acceptable ways of accessing it are grouped into one entity. Other objects, subjects, and applications can use this object and its functionality by accessing it through controlled and standardized interfaces and sending it messages (see Figure 25-9).

Cohesion and Coupling

Cohesion reflects how many different types of tasks a module can carry out. If a module carries out only one task (i.e., subtraction) or tasks that are very similar (i.e., subtract, add, multiply), it is described as having high cohesion, which is a good thing. The higher the cohesion, the easier it is to update or modify the module and not affect other modules that interact with it. This also means the module is easier to reuse and maintain because it is more straightforward when compared to a module with low cohesion. An object with low cohesion carries out multiple *different* tasks and increases the complexity of the module, which makes it harder to maintain and reuse. So, you want your objects to be focused, manageable, and understandable. Each object should carry out a single function or similar functions. One object should not carry out mathematical operations, graphic rendering, and cryptographic functions—these are separate functionality types, and keeping track of this level of complexity would be confusing. If you are attempting to create complex multifunction objects, you are trying to shove too much into one object. Objects should carry out modular, simplistic functions—that is the whole point of OOP.

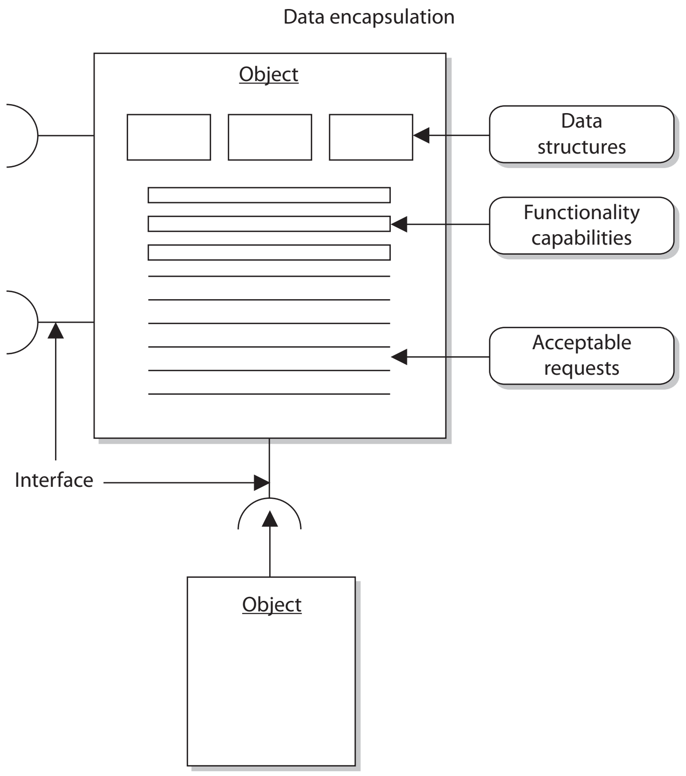


Figure 25-9 The different components of an object and the way it works are hidden from other objects.

Coupling is a measurement that indicates how much interaction one module requires to carry out its tasks. If a module has low (loose) coupling, this means the module does not need to communicate with many other modules to carry out its job. High (tight) coupling means a module depends upon many other modules to carry out its tasks. Low coupling is more desirable because the module is easier to understand and easier to reuse, and it can be changed without affecting many modules around it. Low coupling indicates that the programmer created a well-structured module. As an analogy, a company would want its employees to be able to carry out their individual jobs with the least amount of dependencies on other workers. If Joe has to talk with five other people just to get one task done, too much complexity exists, the task is too time-consuming, and the potential for errors increases with every interaction.

If modules are tightly coupled, the ripple effect of changing just one module can drastically affect the other modules. If they are loosely coupled, this level of complexity decreases.

An example of *low coupling* would be one module passing a variable value to another module. As an example of *high coupling*, Module A would pass a value to Module B, another value to Module C, and yet another value to Module D. Module A could not complete its tasks until Modules B, C, and D completed their tasks and returned results to Module A.

EXAM TIP Objects should be self-contained and perform a single logical function, which is high cohesion. Objects should not drastically affect each other, which is low coupling.

The level of complexity involved with coupling and cohesion can directly impact the security level of a program. The more complex something is, the harder it is to secure. Developing "tight code" not only allows for efficiencies and effectiveness but also reduces the software's attack surface. Decreasing complexity where possible reduces the number of potential holes a bad guy can sneak through. As an analogy, if you were responsible for protecting a facility, your job would be easier if the facility had a small number of doors, windows, and people coming in and out of it. The smaller number of variables and moving pieces would help you keep track of things and secure them.

Application Programming Interfaces

When we discussed some of the attributes of object-oriented development, we spent a bit of time on the concept of abstraction. Essentially, abstraction is all about defining *what* a class or object does and ignoring *how* it accomplishes it internally. An *application programming interface (API)* specifies the manner in which a software component interacts with other software components. We already saw in Chapter 9 how this can come in handy in the context of Trusted Platform Modules (TPMs) and how they constrain communications with untrusted modules. APIs create checkpoints where security controls can be easily implemented. Furthermore, they encourage software reuse and also make the software more maintainable by localizing the changes that need to be made while eliminating (or at least reducing) cascading effects of fixes or changes.

Besides the advantages of reduced effort and improved maintainability, APIs often are required to employ the underlying operating system's functionality. Apple macOS and iOS, Google Android, and Microsoft Windows all require developers to use standard APIs for access to operating system functionality such as opening and closing files and network connections, among many others. All these major vendors restrict the way in which their APIs are used, most notably by ensuring that any parameter that is provided to them is first checked to ensure it is not malformed, invalid, or malicious, which is something we should all do when we are dealing with APIs.

Parameter validation refers to confirming that the parameter values being received by an application are within defined limits before they are processed by the system. In a client/server architecture, validation controls may be placed on the client side prior to submitting requests to the server. Even when these controls are employed, the server should perform parallel validation of inputs prior to processing them because a client has fewer controls than a server and may have been compromised or bypassed.

Software Libraries

APIs are perhaps most familiar to us in the context of software libraries. A *software library* is a collection of components that do specific tasks that are useful to many other components. For example, there are software libraries for various encryption algorithms, managing network connections, and displaying graphics. Libraries allow software developers to work on whatever makes their program unique, while leveraging known-good code for the tasks that similar programs routinely do. The programmer simply needs to understand the API for the libraries she intends to use. This reduces the amount of new code that the programmer needs to develop, which in turn makes the code easier to secure and maintain.

Using software libraries has potential risks, and these risks must be mitigated as part of secure software development practices. The main risk is that, because the libraries are reused across multiple projects in multiple organizations, any defect in these libraries propagates through every program that uses them. In fact, according to Veracode's 2020 report "State of Software Security: Open Source Edition," seven in ten applications use at least one open-source library with a security flaw, which makes those applications vulnerable. Keep in mind that these are open-source libraries, which (as we will discuss later in this chapter) are subject to examination by any number of security researchers looking for bugs. If you use proprietary libraries (including your own), it may be much harder to find these vulnerabilities before the threat actors do.

Secure Software Development

So far in this chapter (and the previous one), we've discussed software development in general terms, pointing out potential security pitfalls along the way. We now turn our attention to how we can bake security into our software from the ground up. To do so, however, we have to come from the top down, meaning we need an organizational policy document that clearly identifies the strategic goals, responsibilities, and authorities for mitigating risks associated with building or acquiring software. If the executive leadership doesn't push this, it just won't happen, and the policy document puts everyone on notice that secure coding is an organizational priority.

Secure coding is a set of practices that reduces (to acceptable levels) the risk of vulnerabilities in our software. No software will ever be 100 percent secure, but we can sure make it hard for threat actors to find and exploit any remaining vulnerabilities if we apply secure coding guidelines and standards to our projects.

Source Code Vulnerabilities

A *source code vulnerability* is a defect in code that provides a threat actor an opportunity to compromise the security of a software system. All code has defects (or bugs), but a vulnerability is a particularly dangerous one. Source code vulnerabilities are typically caused by two types of flaws: design and implementation. A *design flaw* is one that, even if the programmer did everything perfectly right, would still cause the vulnerability. An *implementation flaw* is one that stems from a programmer who incorrectly implemented a part of a good design. For example, suppose you are building an e-commerce application that collects payment card information from your customers and stores it for their future purchases. If you design the system to store the card numbers unencrypted, that would be a design flaw. If, on the other hand, you design the system to encrypt the data as soon as it is captured but a programmer incorrectly calls the encryption function, resulting in the card number being stored in plaintext, that would be an implementation vulnerability.

Source code vulnerabilities are particularly problematic when they exist in externally facing systems such as web applications. These accounted for 39 percent of the external attacks carried out, according to Forrester's report "The State of Application Security, 2021." Web applications deserve particular attention because of their exposure.

The *Open Web Application Security Project (OWASP)* is an organization that deals specifically with web security issues. OWASP offers numerous tools, articles, and resources that developers can utilize to create secure software, and it also has individual member meetings (chapters) throughout the world. OWASP provides development guidelines, testing procedures, and code review steps, but is probably best known for its OWASP Top 10 list of web application security risks. The following is the most recent Top 10 list as of this writing, from 2017 (the 2021 version should be published by the time you're reading this):

- Injection
- Broken Authentication
- Sensitive Data Exposure
- XML External Entities (XEE)
- Broken Access Control
- Security Misconfiguration
- Cross-Site Scripting (XSS)
- Insecure Deserialization
- Using Components with Known Vulnerabilities
- Insufficient Logging & Monitoring

This list represents the most common vulnerabilities that reside in web-based software and are exploited most often. You can find out more information pertaining to these vulnerabilities at https://owasp.org/www-project-top-ten/.

Secure Coding Practices

So, we've talked about secure coding practices, but what exactly are they? Although the specific practices vary from one organization to the next, generally, they break down into two categories: standards and guidelines. Recall from Chapter 1 that *standards* are mandatory activities, actions, or rules, whereas *guidelines* are recommended actions and operational guides that provide the necessary flexibility for unforeseen circumstances. By enforcing secure coding standards and maintaining coding guidelines that reflect best practices, software development organizations dramatically reduce their source code vulnerabilities. Let's see how this works.

Coding Standards

Standards are the strongest form of secure coding practices because, to be considered a standard, a practice must meet the following requirements:

- Demonstrably reduce the risk of a particular type of vulnerability
- Be enforceable across the breadth of an organization's software development efforts
- Be verifiable in its implementation

EXAM TIP The rigorous application of secure coding standards is the best way to reduce source code vulnerabilities.

A very good reference for developing coding standards is the OWASP Top 10 list referenced in the previous section. Though it's focused on web applications, most of the vulnerabilities apply to any kind of software. Another good source of information is the organization's own past experience in developing code with vulnerabilities that later had to be patched.

Once the vulnerabilities are identified, even if at a fairly high level, coding standards can be developed to reduce the risk of building code that contains them. This is where things get a bit sticky, because the standards vary from one programming language to the next. If your organization develops web applications in Ruby (a common language for web apps), the way in which you reduce the risk of, say, broken authentication will be different than if you use PHP (another popular web app language). Still, there are plenty of opportunities to build standards that apply to all languages when we take a step back and consider the processes by which we develop, operationalize, and maintain that code. We'll cover this in more detail when we discuss security controls for software development later in this chapter.

Finally, a standard is only good if we can verify that we are complying with it. (Otherwise, why bother?) So, for instance, if we have a standard that reduces the risk of injection by validating inputs and parameters, then we should have a way to verify that none of our code fails to validate them. An excellent way to verify compliance with secure coding standards is the practice of code reviews, as discussed in Chapter 18. Ideally, though, we can verify at least some of our standards automatedly.

Coding standards enable secure coding by ensuring programmers always do certain things and never do others. For example, a standard could require use of a particular library for encryption functions because it's been analyzed and determined to be sound and free from vulnerabilities. Another example of a standard could be forbidding programmers from using specific unsafe functions, such as the notorious `strcpy()` function in the C programming language. This function copies a string from one memory location to another, but doesn't check the length of the string being copied compared to the destination. If the string is longer than the destination, it will overwrite other areas of memory, which can result in a buffer overflow condition.

Software-Defined Security

A promising new area of security builds on the idea of software-defined networking (SDN), which we covered in Chapter 13. Recall that, in SDN, the control plane (i.e., the routing and switching decisions) is separate from the data plane (i.e., the packets and frames moving around). This allows centralized control of the network, which in turn improves performance, flexibility, and security. SDN also enables the separation of security functions from more traditional network appliance approaches. *Software-defined security (SDS or SDSec)* is a security model in which security functions such as firewalling, intrusion detection and prevention (IDS/IPS), and network segmentation are implemented in software within an SDN environment. One of the advantages of this approach is that sensors (for functions like IDS/IPS) can be dynamically repositioned depending on the threat environment.

SDS is a new technology but promises significant security advantages. Because of its dependence on SDN, SDS is best used in cloud and virtualized network environments.

NOTE Coding standards are required in certain regulated sectors such as automobile and railroad control software, among others.

Coding Guidelines

Secure coding guidelines are recommended practices that tend to be less specific than standards. For example, coding guidelines might encourage programmers to use variable names that are self-explanatory and not reused anywhere else in the program because this makes the code easier to understand. Applied to secure coding, these standards can help by ensuring code is consistently formatted and commented, which makes the code easier to read during code reviews. Guidelines may also recommend that coders keep functions short (without specifying how short) because this reduces the chance of errors. These practices may not sound like much, but they make it easier to spot errors early in the development process, thus improving quality, while decreasing vulnerabilities and costs.

Security Controls for Software Development

We tend to think of security controls as something to be added to an environment in order to reduce risks to it. While this is certainly true of software development environments, secure coding adds another layer, which consists of the security controls we build into the code itself. Regardless of whether we are protecting the development subnetwork or the software that is produced therein, we should implement security controls only after conducting deliberate threat modeling tied to a risk analysis process.

Keep in mind, however, that the threat models for an internal subnet are different from the threat models for software you're deploying throughout your organization or even selling to your customers. Either way, the goals are to reduce vulnerabilities and the possibility of system compromise, but the manner in which we do so will be very different.

Let's zoom in on just software you're developing. Which specific software controls you should use depends on the software itself, its objectives, the security goals of its associated security policy, the type of data it will process, the functionality it is to carry out, and the environment in which it will be placed. If an application is purely proprietary and will run only in closed, trusted environments, it may need fewer security controls than those required for applications that will connect businesses over the Internet and provide financial transactions. The trick is to understand the security needs of a piece of software, implement the right controls and mechanisms, thoroughly test the mechanisms and how they integrate into the application, follow structured development methodologies, and provide secure and reliable distribution methods.

In the sections that follow, we'll identify and describe the application of security controls for the major aspects of software development. These include aspects of the software itself, of course, but also the tools used to develop it, the manner in which we test it, and even how to integrate the software development environment into the broader security architecture.

Development Platforms

Software is normally developed by a team of software engineers who may or may not use the same tools. The most important tool in their tool set is an *integrated development environment (IDE)*, which enables each engineer to pull code from a repository (more on that later), edit it, test it, and then push it into the repository so the rest of the team can build on it. Depending on the programming language, target environments, and a host of other considerations, your developers may use Eclipse, Microsoft Visual Studio, Xcode, or various other applications. The software they develop will likely be tested (formally or otherwise) using development clients and servers that are supposed to represent the production platforms on which the finished software product will run. When we talk about security of the development platforms, therefore, we mean both the development endpoints and the "fake" clients and servers on which the software gets tested.

It may seem obvious, but the first step in ensuring the security of development platforms is to secure the devices on which our software engineers practice their craft. The challenge that many organizations face is that their engineers tend to be more sophisticated than the average user and will make changes to their computers that may or may not be authorized. Their principal incentive, after all, is to develop code quickly and correctly. If the configuration of their workstation gets in the way, it may find itself being modified. To avoid this, you should resist the temptation of giving your software engineers unfettered privileged access to their own devices. Enforcing good change management practices is critical to securing these development endpoints.

Even harder than ensuring change controls on your developers' workstations is securely provisioning the development clients and servers that they will need for testing.

Many organizations allow their developers to stand up and maintain their own development environment, which may be fine provided that these devices are isolated from the production environments. It may sound like common sense, but the problem is that some organizations don't do a good enough job of isolating development and production systems. In principle, doing so simply requires putting the development nodes in an isolated VLAN. In practice, the demarcation is not that cut and dry. This gets even more challenging when the team is distributed, which requires your developers (or perhaps their external collaborators) to remotely access the development hosts.

The best solution is to require use of a VPN to connect to the isolated development network. This may create a bit of work for the operations staff but is the only way to ensure that development and production code remains separate. Another good approach is to create firewall rules that prevent any unauthorized external connections (and even then only the bare minimum) to or from development servers. It should be clear by now that the provisioning of hosts on the development network should not be left to the software development team.

Tool Sets

As the old saying goes, you can't make everyone happy. Your IDE may be awesome, but invariably your software developers will need (or just want) additional tool sets. This is particularly true for developers that have a favorite tool that they've grown used to over the years, or if there is new work to be done for which the existing tools are not ideal. There are two approaches we've seen adopted by many organizations, and neither is ultimately good. The first is to force strict compliance with the approved tool sets that the organization provides. On the surface, this makes sense from a security and operations perspective. Having fewer tools means more standardization, allows for more thorough security assessments, and streamlines provisioning. However, it can also lead to a loss in productivity and typically leads the best coders to give up and move on to another organization where they're allowed more freedom.

The other (not good) approach is to let the developers run amuck in their own playground. The thinking goes something like this: we let them use whatever tools they feel are good, we set up and maintain whatever infrastructure they need, and we just fence the whole thing off from the outside so nothing bad can get in. The end of that sentence should make you shake your head in disagreement because keeping all the bad stuff out obviously is not possible, as you've learned throughout this book. Still, this is the approach of many small and mid-sized development shops.

A better approach is to treat the software development department the same way we treat any other. If they need a new tool, they simply put in a request that goes through the change management process, discussed in Chapter 20. The change advisory board (CAB) validates the requirement, assesses the risk, reviews the implementation plan, and so on. Assuming everything checks out and the CAB approves, the IT operations team integrates the tool into the inventory, updating and provisioning processes; the security team implements and monitors the appropriate controls, and the developers get the new tool they need.

Application Security Testing

Despite our best efforts, we (and all our programmers) are human and will make mistakes. Some of those mistakes will end up being source code vulnerabilities. Wouldn't it be nice to find them before our adversaries do? That's the role of application security testing, which comes in three flavors that you should know for the CISSP exam: static analysis, dynamic analysis, and fuzzing.

Static Application Security Testing

Static application security testing (SAST), also called *static analysis*, is a technique meant to help identify software defects or security policy violations and is carried out by examining the code without executing the program, and therefore is carried out before the program is compiled. The term SAST is generally reserved for automated tools that assist analysts and developers, whereas manual inspection by humans is generally referred to as *code review* (covered in Chapter 18).

SAST allows developers to quickly scavenge their source code for programming flaws and vulnerabilities. Additionally, this testing provides a scalable method of security code review and ensures that developers are following secure coding policies. There are numerous manifestations of SAST tools, ranging from tools that simply consider the behavior of single statements to tools that analyze the entire source code at once. However, you must remember that static code analysis can never reveal logical errors and design flaws, and therefore must be used in conjunction with manual code review to ensure thorough evaluation.

Dynamic Application Security Testing

Dynamic application security testing (DAST), also known as *dynamic analysis*, refers to the evaluation of a program in real time, while it is running. DAST is commonly carried out once a program has cleared the SAST stage and basic programming flaws have been rectified offline. DAST enables developers to trace subtle logical errors in the software that are likely to cause security mayhem later on. The primary advantage of this technique is that it eliminates the need to create artificial error-inducing scenarios. Dynamic analysis is also effective for compatibility testing, detecting memory leakages, identifying dependencies, and analyzing software without having to access the software's actual source code.

EXAM TIP Remember that SAST requires access to the source code, which is not executed during the tests, while DAST requires that you actually run the code but does not require access to the source code.

Fuzzing

Fuzzing is a technique used to discover flaws and vulnerabilities in software by sending large amounts of malformed, unexpected, or random data to the target program in order to trigger failures. Attackers can then manipulate these errors and flaws to inject their own code into the system and compromise its security and stability. Fuzzing tools, aka fuzzers, use complex inputs to attempt to impair program execution. Fuzzing tools

Manual Penetration Testing

Application security testing tools, together with good old-fashioned code reviews, are very good at unearthing most of the vulnerabilities that would otherwise go unnoticed by the software development team. As good as these tools are, however, they lack the creativity and resourcefulness of a determined threat actor. For this reason, many organizations also rely on *manual penetration testing (MPT)* as the final check before code is released into production environments. In this approach, an experienced red team examines the software system in its intended environment and looks for ways to compromise it. It is very common for this testing to uncover additional vulnerabilities that cannot be detected by automated tools.

are commonly successful at identifying buffer overflows, DoS vulnerabilities, injection weaknesses, validation flaws, and other activities that can cause software to freeze, crash, or throw unexpected errors.

Continuous Integration and Delivery

With the advent of Agile methodologies, discussed in Chapter 24, it has become possible to dramatically accelerate the time it takes to develop and release code. This has been taken to an extreme by many of the best software development organizations through processes of continuous integration and continuous delivery.

Continuous integration (CI) means that all new code is integrated into the rest of the system as soon as the developer writes it. For example, suppose Diana is a software engineer working on the user interface of a network detection and response (NDR) system. In traditional development approaches, she would spend a couple of weeks working on UI features, pretty much in isolation from the rest of the development team. There would then be a period of integration in which her code (and that of everyone else who's ready to deliver) gets integrated and tested. Then, Diana (and everyone else) goes back to working alone on her next set of features. The problem with this approach is that Diana gets to find out whether her code integrates properly only every two weeks. Wouldn't it be nice if she could find out instantly (or at least daily) whether any of her work has integration issues?

With continuous integration, Diana works on her code for a few hours and then merges it into a shared repository. This merge triggers a batch of unit tests. If her code fails those tests, the merge is rejected. Otherwise, her code is merged with everyone else's in the repository and a new version of the entire software system is built. If there are any errors in the build, she knows her code was the cause, and she can get to work fixing them right away. If the build goes well, it is immediately subjected to automated integration tests. If anything goes wrong, Diana knows she has to immediately get back to work fixing her code because she "broke the build," meaning nobody else can commit code until she fixes it or reverses her code merge.

Continuous integration dramatically improves software development efficiency by identifying errors early and often. CI also allows the practice of *continuous delivery (CD)*, which is incrementally building a software product that can be released at any time. Because all processes and tests are automated, you could choose to release code to production daily or even hourly. Most organizations that practice CI/CD, however, don't release code that frequently. But they could if they wanted to.

CI/CD sounds wonderful, so what are the security risks we need to mitigate? Because CI/CD relies heavily on automation, most organizations that practice it use commercial or open-source testing platforms. One of those platforms is Codecov, which was compromised in early 2021, allowing the threat actor to modify its bash uploader script. This is the script that would take Diana's code in our earlier example and upload it for testing and integration. As an aside, because the tests are automated and don't involve actual users, developers typically have to provide access credentials, tokens, or keys to enable testing. The threat actor behind the Codecov breach modified the bash uploader so that it would exfiltrate this access data, potentially providing covert access to any of the millions of products worldwide that use Codecov for CI/CD.

The Codecov breach was detected about three months later by an alert customer who noticed unusual behavior in the uploader, investigated it, and alerted the vendor to the problem. Would you be able to tell that one of the components in your CI/CD toolset was leaking sensitive data? You could if you practice the secure design principles we've been highlighting throughout the book, especially threat modeling, least privilege, defense in depth, and zero trust.

Security Orchestration, Automation, and Response

The Codecov breach mentioned in the previous section also highlights the role that a security orchestration, automation, and response (SOAR) platform can play in securing your software development practices. Chapter 21 introduced SOAR in the context of the role of a security information and event management (SIEM) platform in your security operations. Both SOAR and SIEM platforms can help detect and, in the case of SOAR, respond to threats against your software development efforts. If you have sensors in your development subnet (you did segment your network, right?) and a well-tuned SOAR platform, you can detect new traffic flowing from that subnet (which shouldn't be talking much to the outside world) to a new external endpoint. If the traffic is unencrypted (or you use a TLS decryption proxy to do deep packet inspection), you'd notice access tokens and keys flowing out to a new destination. Based on this observation, you could declare an incident and activate the playbook for data breaches in your SOAR platform. Just like that, you would've stopped the bleeding, buying you time to figure out what went wrong and how to fix it for the long term.

One of the challenges with the scenario just described is that many security teams treat their organization's development environment as a bit of a necessary chaos that must be tolerated. Software developers are typically rewarded (or punished) according to their ability to produce quality code quickly. They can be resistant (or even rebel against) anything that gets in the way of their efficiency, and, as we well know, security tends to do just that. This is where DevSecOps (discussed in Chapter 24) can help build the right culture and balance the needs of all teammates. It can also help the security team identify and implement controls that mitigate risks such as data breaches, while minimally affecting productivity. One such control is the placement of sensors such as IDS/IPS, NDR, and data loss prevention (DLP) within the development subnets. These systems, in turn, would report to the SOAR platform, which could detect and contain active threats against the organization.

Software Configuration Management

Not every threat, of course, is external. There are plenty of things our own teammates can do deliberately or otherwise that cause problems for the organization. As we'll see later in this chapter when we discuss cloud services, improper configurations consistently rank among the worst threats to many organizations. This threat, however, is a solved problem in organizations that practice proper configuration management, as we covered in Chapter 20.

Anticipating the inevitable changes that will take place to a software product during its development life cycle, a configuration management system should be put into place that allows for change control processes to take place through automation. Since deploying an insecure configuration to an otherwise secure software product makes the whole thing insecure, these settings are a critical component of securing the software development environment. A product that provides *software configuration management (SCM)* identifies the attributes of software at various points in time and performs a methodical control of changes for the purpose of maintaining software integrity and traceability throughout the software development life cycle. It tracks changes to configurations and provides the ability to verify that the final delivered software has all of the approved changes that are supposed to be included in the release.

During a software development project, the centralized code repositories are often kept in systems that can carry out SCM functionality. These SCM systems manage and track revisions made by multiple people against a single master set and provide concurrency management, versioning, and synchronization. *Concurrency management* deals with the issues that arise when multiple people extract the same file from a central repository and make their own individual changes. If they were permitted to submit their updated files in an uncontrolled manner, the files would just write over each other and changes would be lost. Many SCM systems use algorithms to version, fork, and merge the changes as files are checked back into the repository.

Versioning deals with keeping track of file revisions, which makes it possible to "roll back" to a previous version of the file. An archive copy of every file can be made when it is checked into the repository, or every change made to a file can be saved to a transaction log. Versioning systems should also create log reports of who made changes, when they were made, and what the changes were.

Some SCM systems allow individuals to check out complete or partial copies of the repositories and work on the files as needed. They can then commit their changes back to the master repository as needed and update their own personal copies to stay up to date with changes other people have made. This process is called *synchronization*.

Code Repositories

A *code repository*, which is typically a version control system, is the vault containing the crown jewels of any organization involved in software development. If we put on our adversarial hats for a few minutes, we could come up with all kinds of nefarious scenarios involving these repositories. Perhaps the simplest is that someone could steal our source code, which embodies not only many staff hours of work but, more significantly, our intellectual property. An adversary could also use our source code to look for vulnerabilities to exploit later, once the code is in production. Finally, adversaries could deliberately insert vulnerabilities into our software, perhaps after it has undergone all testing and is trusted, so that they can exploit it later at a time of their choosing. Clearly, securing our source code repositories is critical.

Perhaps the most secure way of managing security for your code repositories is to implement them on an isolated (or "air-gapped") network that includes the development, test, and QA environments. The development team would have to be on this network to do their work, and the code, once verified, could be exported to the production servers using removable storage media. We already presented this best

Software Escrow

If a company pays another company to develop software for it, it should have some type of *software escrow* in place for protection. We covered this topic in Chapter 23 from a business continuity perspective, but since it directly deals with software development, we will mention it here also.

In a software escrow framework, a third party keeps a copy of the source code, and possibly other materials, which it will release to the customer only if specific circumstances arise, mainly if the vendor who developed the code goes out of business or for some reason is not meeting its obligations and responsibilities. This procedure protects the customer, because the customer pays the vendor to develop software code for it, and if the vendor goes out of business, the customer otherwise would no longer have access to the actual code. This means the customer code could never be updated or maintained properly.

A logical question would be, "Why doesn't the vendor just hand over the source code to the customer, since the customer paid for it to be developed in the first place?" It does not always work that way. The code may be the vendor's intellectual property. The vendor employs and pays people with the necessary skills to develop that code, and if the vendor were to just hand it over to the customer, it could be giving away its intellectual property, its secrets. The customer oftentimes gets compiled code instead of source code. *Compiled code* is code that has been put through a compiler and is unreadable to humans. Most software profits are based on licensing, which outlines what customers can do with the compiled code. For an added fee, of course, most custom software developers will also provide the source, which could be useful in sensitive applications.

practice in the preceding section. The challenge with this approach is that it severely limits the manner in which the development team can connect to the code. It also makes it difficult to collaborate with external parties and for developers to work from remote or mobile locations.

A pretty good alternative would be to host the repository on the intranet, which would require developers to either be on the local network or connect to it using a VPN connection. As an added layer of security, the repositories can be configured to require the use of Secure Shell (SSH), which would ensure all traffic is encrypted, even inside the intranet, to mitigate the risk of sniffing. Finally, SSH can be configured to use public key infrastructure (PKI), which allows us to implement not only confidentiality and integrity but also nonrepudiation. If you have to allow remote access to your repository, this would be a good way to go about it.

Finally, if you are operating on a limited budget or have limited security expertise in this area, you can choose one of the many web-based repository service providers and let them take care of the security for you. While this may mitigate the basic risks for small organizations, it is probably not an acceptable course of action for projects with significant investments of intellectual property.

Software Security Assessments

We already discussed the various types of security assessments in Chapter 18, but let's circle back here and see how these apply specifically to software security. Recall from previous sections in this chapter that secure software development practices originate in an organizational policy that is grounded in risk management. That policy is implemented through secure coding standards, guidelines, and procedures that should result in secure software products. We verify this is so through the various testing methods discussed in this chapter (e.g., SAST and DAST) and Chapter 24 (e.g., unit, integration, etc.). The purpose of a software security assessment, then, is to verify that this entire chain, from policy to product, is working as it should.

When conducting an assessment, it is imperative that the team review all applicable documents and develop a plan for how to verify each requirement from the applicable policies and standards. Two areas that merit additional attention are the manner in which the organization manages risks associated with software development and how it audits and logs software changes.

Risk Analysis and Mitigation

Risk management is at the heart of secure software development, particularly the mapping between risks we've identified and the controls we implement to mitigate them. This is probably one of the trickiest challenges in secure software development in general, and in auditing it in particular. When organizations do map risks to controls in software development, they tend to do so in a generic way. For example, the OWASP Top 10 list is a great starting point for analyzing and mitigating vulnerabilities, but how are we doing against specific (and potentially unique) threats faced by our organization?

Threat modeling is an important activity for any development team, and particularly in DevSecOps. Sadly, however, most organizations don't conduct threat modeling for their software development projects. If they're defending against generic threats, that's good, but sooner or later we all face unique threats that, if we haven't analyzed and mitigated them, have a high probability of ruining our weekend.

Another area of interest for assessors are the linkages between the software development and risk management programs. If software projects are not tracked in the organization's risk matrix, then the development team will probably be working in isolation, disconnected from the broader risk management efforts.

Change Management

Another area in which integration with broader organizational efforts is critical to secure software development is change management. Changes to a software project that may appear inconsequential when considered in isolation could actually pose threats when analyzed within the broader context of the organization. If software development is not integrated into the organization's change management program, auditing changes to software products may be difficult, even if the changes are being logged by the development team. Be that as it may, software changes should not be siloed from overall organizational change management because doing so will likely lead to interoperability or (worse yet) security problems.

Assessing the Security of Acquired Software

Most organizations do not have the in-house capability to develop their own software systems. Their only feasible options are either to acquire standard software or to have a vendor build or customize a software system to their particular environment. In either case, software from an external source will be allowed to execute in a trusted environment. Depending on how trustworthy the source and the code are, this could have some profound implications to the security posture of the organization's systems. As always, we need to ground our response on our risk management process.

In terms of managing the risk associated with acquired software, the essential question to ask is, "How is the organization affected if this software behaves improperly?" Improper behavior could be the consequence of either defects or misconfiguration. The defects can manifest themselves as computing errors (e.g., wrong results) or vulnerability to intentional attack. A related question is, "What is it that we are protecting and this software could compromise?" Is it personally identifiable information (PII), intellectual property, or national security information? The answers to these and other questions will dictate the required thoroughness of our approach.

In many cases, our approach to mitigating the risks of acquired software will begin with an assessment of the software developer. Characteristics that correlate to a lower software risk include the good reputation of the developer and the regularity of its patch pushes. Conversely, developers may be riskier if they have a bad reputation, are small or new organizations, if they have immature or undocumented development processes, or if their products have broad marketplace presence (meaning they are more lucrative targets to exploit developers).

A key element in assessing the security of acquired software is, rather obviously, its performance in an internal assessment. Ideally, we are able to obtain the source code from the vendor so that we can do our own code reviews, vulnerability assessments, and penetration tests. In many cases, however, this will not be possible. Our only possible assessment may be a penetration test. The catch is that we may not have the in-house capability to perform such a test. In such cases, and depending on the potential risk posed by this software, we may be well advised to hire an external party to perform an independent penetration test for us. This is likely a costly affair that would only be justifiable in cases where a successful attack against the software system would likely lead to significant losses for the organization.

Even in the most constrained case, we are still able to mitigate the risk of acquisition. If we don't have the means to do code reviews, vulnerability assessments, or penetration tests, we can still mitigate the risk by deploying the software only in specific subnetworks, with hardened configurations, and with restrictive IDS/IPS rules monitoring its behavior. Though this approach may initially lead to constrained functionality and excessive false positives generated by our IDS/IPS, we can always gradually loosen the controls as we gain assurances that the software is trustworthy.

Commercial Software

It is exceptionally rare for an organization to gain access to the source code of a commercial-off-the-shelf (COTS) product to conduct a security assessment of it. However, depending on the product, we may not have to. The most widely used commercial software products have been around for years and have had their share of security researchers (both benign and malicious) poking at them the whole time. We can simply research what vulnerabilities and exploits have been discovered by others and decide for ourselves whether or not the vendor uses effective secure coding practices.

If the software is not as popular, or serves a small niche community, the risk of undiscovered vulnerabilities is probably higher. In these cases, it pays to look into the certifications of the vendor. A good certification for a software developer is ISO/IEC 27034 Application Security. Unfortunately, you won't find a lot of vendors certified in it. There are also certifications that are very specific to a sector (e.g., ISO 26262 for automotive safety) or a programming language (e.g., ISO/IEC TS 17961:2013 for coding in C) and are a bit less rare to find. Ultimately, however, the security of a vendor's software products is tied to how seriously it takes security in the first place. Absent a secure coding certification, you can look for overall information security management system (ISMS) certifications like ISO/IEC 27001 and FedRAMP, which are difficult to obtain and show that security is taken seriously in an organization.

Open-Source Software

Open-source software is released with a license agreement that allows the user to examine its source code, modify it at will, and even redistribute the modified software (which, per the license, usually requires acknowledgment of the original source and a description of modifications). This may seem perfect, but there are some caveats to keep in mind. First, the software is released as-is, typically without any service or support agreements (though these can be purchased through third parties). This means that your staff may have to figure out how to install, configure, and maintain the software on their own, unless you contract with someone else to do this for you.

Second, part of the allure of open-source software is that we get access to the source code. This means we can apply all the security tests and assessments we covered earlier. Of course, this only helps if we have the in-house capabilities to examine the source code effectively. Even if we don't, however, we can rely on countless developers and researchers around the world who do examine it (at least for the more popular software). The flip side of that coin, however, is that the adversaries also get to examine the code to either identify vulnerabilities quicker than the defenders or gain insights into how they might more effectively attack organizations that use specific software.

Perhaps the greatest risk in using open-source software is relying on outdated versions of it. Many of us are used to having software that automatically checks for updates and applies them automatically (either with or without our explicit permission). This is not all that common in open-source software, however, especially libraries. This means we need to develop processes to ensure that all open-source software is periodically updated, possibly in a way that differs from the way in which COTS software is updated.

Third-Party Software

Third-party software, also known as *outsourced software*, is software made specifically for an organization by a third party. Since the software is custom (or at least customized), it is not considered COTS. Third-party software may rely partly (or even completely) on open-source software, but, having been customized, it may introduce new vulnerabilities. So, we need a way to verify the security of these products that is probably different from how we would do so with COTS or open-source software.

EXAM TIP Third-party software is custom (or at least customized) to an organization and is not considered commercial off-the-shelf (COTS).

The best (and, sadly, most expensive) way to assess the security of third-party software is to leverage the external or third-party audits discussed in Chapter 18. The way this typically works is that we write into the contract a provision for an external auditor to inspect the software (and possibly the practices through which it was developed), and then issue a report, attesting to the security of the product. Passing this audit can be a condition of finalizing the purchase. Obviously, a sticking point in this negotiation can be who pays for this audit.

Another assessment approach is to arrange for a time-limited trial of the third-party software (perhaps at a nominal cost to the organization), and then have a red team perform an assessment. If you don't have a red team, you can probably hire one for less money than a formal application security audit would cost. Still, the cost will be considerable, typically (at least) in the low tens of thousands of dollars. As with any other security control, you'd have to balance the cost of the assessment and the loss you would incur from insecure software.

Managed Services

As our organizations continue to migrate to cloud services (IaaS, PaaS, and SaaS, discussed in depth in Chapter 7), we should also assess the security impact of those services. This is highlighted by a 2020 study by global intelligence firm IDC, which found that nearly 80 percent of the companies surveyed had experienced at least one cloud data breach in the past 18 months. The top three reasons were misconfigurations, lack of visibility into access settings and activities, and improper access control. The major cloud services provide tools to help you avoid these pitfalls, but the bottom line is that, if you don't have the in-house expertise to secure and assess your cloud services, you really should consider contracting an expert to help you out.

Chapter Review

Building secure code requires commitment from many parts of the organization, not just the development and security teams. It starts at the very top with a policy document that is implemented through standards, procedures, and guidelines. A key part of these is the inclusion of the various types of tests that must be run regularly (even continuously) on the software as it is being written, integrated, and prepared for delivery. Software development environments are complex and could require different approaches from those you'd take in a normal network environment. For this reason, teamwork among all stakeholders is absolutely critical. A really good way to facilitate this collaboration is by using the DevSecOps approach introduced in Chapter 24 and highlighted in this one.

Even if your organization doesn't develop software, it most certainly uses applications and services developed by others. That's why the concepts discussed in this chapter are universally applicable to any cybersecurity leader. You must understand how secure code is built, so that you can determine whether the software you're getting from others presents any undue risks to your organization's cybersecurity.

Quick Review

- Machine language, which consists of 1's and 0's, is the only format that a computer's processor can understand directly and is considered a first-generation language.
- Assembly language is considered a second-generation programming language and uses symbols (called mnemonics) to represent complicated binary codes.
- Third-generation programming languages, such as C/C++, Java, and Python, are known as high-level languages due to their refined programming structures, which allow programmers to leave low-level (system architecture) intricacies to the programming language and focus on their programming objectives.
- Fourth-generation languages (aka very high-level languages) use natural language processing to allow inexpert programmers to develop code in less time than it would take an experienced software engineer to do so using a third-generation language.

- Fifth-generation programming languages (aka natural languages) approach programming by defining the constraints for achieving a specified result and allowing the development environment to solve problems by itself instead of a programmer having to develop code to deal with individual and specific problems.
- Assemblers are tools that convert assembly language source code into machine code.
- Compilers transform instructions from a source language (high-level) to a target language (machine), sometimes using an external assembler along the way.
- A garbage collector identifies blocks of memory that were once allocated but are no longer in use and deallocates the blocks and marks them as free.
- A runtime environment (RTE) functions as a miniature operating system for the program and provides all the resources portable code needs.
- In object-oriented programming (OOP), related functions and data are encapsulated together in classes, which may then be instantiated as objects.
- Objects in OOP communicate with each other by using messages that conform to the receiving object's application programming interface (API) definition.
- Cohesion reflects how many different types of tasks a module can carry out, with the goal being to perform only one task (high cohesion), which makes modules easier to maintain.
- Coupling is a measure of how much a module depends on others; the more dependencies it has, the more complex and difficult the module is to maintain, so we want low (or loose) coupling.
- An API specifies the manner in which a software component interacts with other software components.
- Parameter validation refers to confirming that the parameter values being received by an application are within defined limits before they are processed by the system.
- A software library is a collection of components that do specific tasks that are useful to many other components.
- Secure coding is a set of practices that reduce (to acceptable levels) the risk of vulnerabilities in our software.
- A source code vulnerability is a defect in code that provides a threat actor an opportunity to compromise the security of a software system.
- Secure coding standards are verifiable, mandatory practices that reduce the risk of particular types of vulnerabilities in the source code.
- Secure coding guidelines are recommended practices that tend to be less specific than standards.

- Software-defined security (SDS or SDSec) is a security model in which security functions such as firewalling, IDS/IPS, and network segmentation are implemented in software within an SDN environment.
- Software development tools should be authorized, implemented, and maintained just like any other software product through the organization's change management process; developers should not be allowed to install and use arbitrary tools.
- Static application security testing (SAST) is a technique meant to help identify software defects or security policy violations and is carried out by examining the source code without executing the program.
- Dynamic application security testing (DAST) refers to the evaluation of a program in real time, while it is running.
- Fuzzing is a technique used to discover flaws and vulnerabilities in software by sending large amounts of malformed, unexpected, or random data to the target program in order to trigger failures.
- Continuous integration means that all new code is integrated into the rest of the system as soon as the developer writes it.
- Continuous delivery is incrementally building a software product that can be released at any time and requires continuous integration.
- A software configuration management (SCM) platform identifies the attributes of software at various points in time and performs a methodical control of changes for the purpose of maintaining software integrity and traceability throughout the SDLC.
- The purpose of a software security assessment is to verify that this entire development process, from organizational policy to delivered product, is working as it should.
- Security assessments of acquired software are essential to mitigate the risk they could pose to the organization that acquired it.
- The most practical way to assess the security of commercial software is to research what vulnerabilities and exploits have been discovered by others and decide for ourselves whether or not the vendor uses effective secure coding practices.
- The greatest risk in using open-source software is relying on outdated versions of it.
- The best way to assess the security of third-party (i.e., custom or customized) software is to perform external or third-party audits.

Questions

Please remember that these questions are formatted and asked in a certain way for a reason. Keep in mind that the CISSP exam is asking questions at a conceptual level. Questions may not always have the perfect answer, and the candidate is advised against always looking for the perfect answer. Instead, the candidate should look for the best answer in the list.

1. What language is the only one that a computer processor can natively understand and execute?
   - **A.** Machine language
   - **B.** Register language
   - **C.** Assembly language
   - **D.** High-level language
2. To which generation do programming languages such as such as C/C++, Java, and Python belong?
   - **A.** Second generation
   - **B.** Third generation
   - **C.** Fourth generation
   - **D.** Fifth generation
3. Which type of tool is specifically designed to convert assembly language into machine language?
   - **A.** Compiler
   - **B.** Integrated development environment (IDE)
   - **C.** Assembler
   - **D.** Fuzzer
4. Which of the following is not very useful in assessing the security of acquired software?
   - **A.** The reliability and maturity of the vendor
   - **B.** The vendor's software escrow framework
   - **C.** Third-party vulnerability assessments
   - **D.** In-house code reviews if source code is available
5. Cohesion and coupling are characteristics of quality code. Which of the following describes the goals for these two characteristics?
   - **A.** Low cohesion, low coupling
   - **B.** Low cohesion, high coupling
   - **C.** High cohesion, low coupling
   - **D.** High cohesion, high coupling

6. Yichen is a new software engineer at Acme Software, Inc. During his first code review, he is told by his boss that he should use descriptive names for variables in his code. What is this observation an example of?
   - **A.** Secure coding guidelines
   - **B.** Secure coding standards
   - **C.** Secure software development policy
   - **D.** Use of fifth-generation language
7. On what other technology does software-defined security depend?
   - **A.** Software-defined storage (SDS)
   - **B.** Software-defined networking (SDN)
   - **C.** Security orchestration, automation, and response (SOAR)
   - **D.** Continuous integration (CI)
8. If you wanted to test source code for vulnerabilities without running it, which approach would be best?
   - **A.** Static application security testing (SAST)
   - **B.** Fuzzing
   - **C.** Dynamic application security testing (DAST)
   - **D.** Manual penetration testing
9. If you wanted to test software for vulnerabilities by executing it and then exposing it to large amounts of random inputs, which testing technique would you use?
   - **A.** Static application security testing (SAST)
   - **B.** Fuzzing
   - **C.** Dynamic application security testing (DAST)
   - **D.** Manual penetration testing
10. Which of the following is not a common reason for data breaches in managed cloud services?
    - **A.** Misconfigurations
    - **B.** Lack of visibility into access settings and activities
    - **C.** Hardware failures
    - **D.** Improper access control

Answers

1. **A.** Machine language, which consists of 1's and 0's, is the only format that a computer's processor can understand directly and is considered a first-generation language.

2. **B.** Third-generation programming languages, such as C/C++, Java, and Python, are known as high-level languages due to their refined programming structures, which allow programmers to leave low-level (system architecture) intricacies to the programming language and focus on their programming objectives.
3. **C.** Assemblers are tools that convert assembly language source code into machine code. Compilers also generate machine language, but do so by transforming high-level language code, not assembly language.
4. **B.** In a software escrow framework, a third party keeps a copy of the source code, and possibly other materials, which it will release to the customer in specific circumstances such as the developer going out of business. While software escrow is a good business continuity practice, it wouldn't normally tell us anything about the security of the software itself. All three other answers are part of a rigorous assessment of the security of acquired software.
5. **C.** Cohesion reflects how many different types of tasks a module can carry out, with the goal being to perform only one task (high cohesion), which makes modules easier to maintain. Coupling is a measure of how much a module depends on others; the more dependencies it has, the more complex and difficult the module is to maintain, so we want low (or loose) coupling.
6. **A.** Secure coding guidelines are recommended practices that tend to be less specific than standards. They might encourage programmers to use variable names that are self-explanatory and to keep functions short (without specifying how short). Secure coding standards, on the other hand, are verifiable, mandatory practices that reduce the risk of particular types of vulnerabilities in the source code.
7. **B.** Software-defined security (SDS or SDSec) is a security model in which security functions such as firewalling, IDS/IPS, and network segmentation are implemented in software within an SDN environment.
8. **A.** Static application security testing (SAST) is a technique meant to help identify software defects or security policy violations and is carried out by examining the source code without executing the program. All the other answers require that the code be executed.
9. **B.** Fuzzing is a technique used to discover flaws and vulnerabilities in software by sending large amounts of malformed, unexpected, or random data to the target program in order to trigger failures.
10. **C.** The top three reasons for data breaches in cloud services are misconfigurations, lack of visibility into access settings and activities, and improper access control.

This page intentionally left blank

APPENDIX A

Comprehensive Questions

Use the following scenario to answer Questions 1–3. Josh has discovered that an organized hacking ring in China has been targeting his company's research and development department. If these hackers have been able to uncover his company's research findings, this means they probably have access to his company's intellectual property. Josh thinks that an e-mail server in his company's DMZ may have been successfully compromised and a rootkit loaded.

1. Based upon this scenario, what is most likely the biggest risk Josh's company needs to be concerned with?

- **A.** Market share drop if the attackers are able to bring the specific product to market more quickly than Josh's company.
- **B.** Confidentiality of e-mail messages. Attackers may post all captured e-mail messages to the Internet.
- **C.** Impact on reputation if the customer base finds out about the attack.
- **D.** Depth of infiltration of attackers. If attackers have compromised other systems, more confidential data could be at risk.

2. The attackers in this situation would be seen as which of the following?

- **A.** Vulnerability
- **B.** Threat
- **C.** Risk
- **D.** Threat agent

3. If Josh is correct in his assumptions, which of the following best describes the vulnerability, threat, and exposure, respectively?

- **A.** E-mail server is hardened, an entity could exploit programming code flaw, server is compromised and leaking data.
- **B.** E-mail server is not patched, an entity could exploit a vulnerability, server is hardened.
- **C.** E-mail server misconfiguration, an entity could exploit misconfiguration, server is compromised and leaking data.
- **D.** DMZ firewall misconfiguration, an entity could exploit misconfiguration, internal e-mail server is compromised.

4. Aaron is a security manager who needs to develop a solution to allow his company's mobile devices to be authenticated in a standardized and centralized manner using digital certificates. The applications these mobile clients use require a TCP connection. Which of the following is the best solution for Aaron to implement?
   - A. TACACS+
   - B. RADIUS
   - C. Diameter
   - D. Mobile IP
5. Terry is a security manager for a credit card processing company. His company uses internal DNS servers, which are placed within the LAN, and external DNS servers, which are placed in the DMZ. The company also relies on DNS servers provided by its service provider. Terry has found out that attackers have been able to manipulate several DNS server caches to point employee traffic to malicious websites. Which of the following best describes the solution this company should implement?
   - A. IPSec
   - B. PKI
   - C. DNSSEC
   - D. MAC-based security
6. Which of the following is not a key provision of the GDPR?
   - A. Requirement for consent from data subjects
   - B. Right to be informed
   - C. Exclusion for temporary workers
   - D. Right to be forgotten
7. Jane is suspicious that an employee is sending sensitive data to one of the company's competitors but is unable to confirm this. The employee has to use this data for daily activities, thus it is difficult to properly restrict the employee's access rights. In this scenario, which best describes the company's vulnerability, threat, risk, and necessary control?
   - A. Vulnerability is employee access rights, threat is internal entities misusing privileged access, risk is the business impact of data loss, and the necessary control is detailed network traffic monitoring.
   - B. Vulnerability is lack of user monitoring, threat is internal entities misusing privileged access, risk is the business impact of data loss, and the necessary control is detailed user activity logs.
   - C. Vulnerability is employee access rights, threat is internal employees misusing privileged access, risk is the business impact of confidentiality, and the necessary control is multifactor authentication.
   - D. Vulnerability is employee access rights, threat is internal users misusing privileged access, risk is the business impact of confidentiality, and the necessary control is CCTV.

8. Which of the following best describes what role-based access control offers organizations in reducing administrative burdens?

A. It allows entities closer to the resources to make decisions about who can and cannot access resources.

B. It provides a centralized approach for access control, which frees up department managers.

C. User membership in roles can be easily revoked and new ones established as job assignments dictate.

D. It enforces an enterprise-wide security policy, standards, and guidelines.

9. Mark works for a large corporation operating in multiple countries worldwide. He is reviewing his company's policies and procedures dealing with data breaches. Which of the following is an issue that he must take into consideration?

A. Each country may or may not have unique notification requirements.

B. All breaches must be announced to affected parties within 24 hours.

C. Breach notification is a "best effort" process and not a guaranteed process.

D. Breach notifications are avoidable if all PII is removed from data stores.

10. A software development company released a product that committed several errors that were not expected once deployed in their customers' environments. All of the software code went through a long list of tests before being released. The team manager found out that after a small change was made to the code, the program was not tested before it was released. Which of the following tests was most likely not conducted?

A. Unit

B. Compiled

C. Integration

D. Regression

11. Which of the following should not be considered as part of the supply chain risk management process for a smartphone manufacturer?

A. Hardware Trojans inserted by downstream partners

B. ISO/IEC 27001

C. Hardware Trojans inserted by upstream partners

D. NIST Special Publication 800-161

12. Data sovereignty is increasingly becoming an issue that most of us in cybersecurity should address within our organizations. What does the term data sovereignty mean?

A. Certain types of data concerning a country's citizens must be stored and processed in that country.

B. Data on a country's citizens must be stored and processed according to that country's laws, regardless of where the storing/processing takes place.

C. Certain types of data concerning a country's citizens are the sovereign property of that data subject.

D. Data on a country's citizens must never cross the sovereign borders of another country.

Use the following scenario to answer Questions 13–15. Jack has just been hired as the security officer for a large hospital system. The organization develops some of its own proprietary applications. The organization does not have as many layers of controls when it comes to the data processed by these applications, since it is assumed that external entities will not understand the internal logic of the applications. One of the first things that Jack wants to carry out is a risk assessment to determine the organization's current risk profile. He also tells his boss that the hospital should become ISO certified to bolster its customers' and partners' confidence in its risk management processes.

13. Which of the following approaches has been implemented in this scenario?

A. Defense-in-depth

B. Security through obscurity

C. Information security management system

D. ISO/IEC 27001

14. Which ISO/IEC standard would be best for Jack to follow to meet his goals?

A. ISO/IEC 27001

B. ISO/IEC 27004

C. ISO/IEC 27005

D. ISO/IEC 27006

15. Which standard should Jack suggest to his boss for compliance with best practices regarding storing and processing sensitive medical information?

A. ISO/IEC 27004

B. ISO/IEC 27001

C. ISO/IEC 27799

D. ISO/IEC 27006

16. You just received an e-mail from one of your hardware manufacturers notifying you that it will no longer manufacture a certain product and, after the end of the year, you won't be able to send it in for repairs, buy spare parts, or get technical assistance from that manufacturer. What term describes this?

A. End-of-support (EOS)

B. End-of-service-life (EOSL)

C. Deprecation

D. End-of-life (EOL)

17. The confidentiality of sensitive data is protected in different ways depending on the state of the data. Which of the following is the best approach to protecting data in transit?

A. SSL

B. VPN

C. IEEE 802.1X

D. Whole-disk encryption

18. Your boss asks you to put together a report describing probable adverse effects on your assets caused by specific threat sources. What term describes this?

A. Risk analysis

B. Threat modeling

C. Attack trees

D. MITRE ATT&CK

19. A(n) ___________ is the graphical representation of data commonly used on websites. It is a skewed representation of characteristics a person must enter to prove that the subject is a human and not an automated tool, as in a software robot.

A. anti-spoofing symbol

B. CAPTCHA

C. spam anti-spoofing symbol

D. CAPCHAT

20. Mark has been asked to interview individuals to fulfill a new position in his company, chief privacy officer (CPO). What is the function of this type of position?

A. Ensuring that company financial information is correct and secure

B. Ensuring that customer, company, and employee data is protected

C. Ensuring that security policies are defined and enforced

D. Ensuring that partner information is kept safe

21. A risk management program must be developed properly and in the right sequence. Which of the following provides the correct sequence for the steps listed?

i. Develop a risk management team.

ii. Calculate the value of each asset.

iii. Identify the vulnerabilities and threats that can affect the identified assets.

iv. Identify company assets to be assessed.

A. i, iii, ii, iv

B. ii, i, iv, iii

C. iii, i, iv, ii

D. i, iv, ii, iii