Chapter 20
Software Development Security

THE CISSP TOPICS COVERED IN THIS CHAPTER INCLUDE:

- ✓ **Domain 3.0: Security Architecture and Engineering**
  - 3.5 Assess and mitigate the vulnerabilities of security architectures, designs, and solution elements
    - 3.5.3 Database systems
- ✓ **Domain 8.0: Software Development Security**
  - 8.1 Understand and integrate security in the Software Development Life Cycle (SDLC)
    - 8.1.1 Development methodologies (e.g., Agile, Waterfall, DevOps, DevSecOps, Scaled Agile Framework)
    - 8.1.2 Maturity models (e.g., Capability Maturity Model (CMM), Software Assurance Maturity Model (SAMM))
    - 8.1.3 Operation and maintenance
    - 8.1.4 Change management
    - 8.1.5 Integrated Product Team
  - 8.2 Identify and apply security controls in software development ecosystems
    - 8.2.1 Programming languages
    - 8.2.2 Libraries
    - 8.2.3 Tool sets
    - 8.2.4 Integrated Development Environment
    - 8.2.5 Runtime
    - 8.2.6 Continuous Integration and Continuous Delivery (CI/CD)
    - 8.2.7 Software Configuration Management (CM)

- 8.2.8 Code repositories
- 8.3 Assess the effectiveness of software security
  - 8.3.1 Auditing and logging of changes
- 8.4 Assess security impact of acquired software
  - 8.4.1 Commercial-off-the-shelf (COTS)
  - 8.4.2 Open source
  - 8.4.3 Third-party
- 8.5 Define and apply secure coding guidelines and standards
  - 8.5.2 Security of application programming interfaces (API)
  - 8.5.3 Secure coding practices
  - 8.5.4 Software-defined security

Software development is a complex and challenging task undertaken by developers with many different skill levels and varying levels of security awareness. Applications created and modified by these developers often work with sensitive data and interact with members of the general public. That means that applications can present significant risks to enterprise security, and information security professionals must understand these risks, balance them with business requirements, and implement appropriate risk mitigation mechanisms.

Introducing Systems Development Controls

Many organizations use custom-developed software to achieve their unique business objectives. These custom solutions can present great security vulnerabilities as a result of malicious and/or careless developers who create backdoors, buffer overflow vulnerabilities, or other weaknesses that can leave a system open to exploitation by malicious individuals.

To protect against these vulnerabilities, it's vital to introduce security controls into the entire system's development life cycle. An organized, methodical process helps ensure that solutions meet functional requirements as well as security guidelines. The following sections explore the spectrum of systems development activities with an eye toward security concerns that should be foremost on the mind of any information security professional engaged in solutions development.

Software Development

Security should be a consideration at every stage of a system's development, including the software development process. Programmers should strive to build security into every application they develop, with greater levels of security provided to critical applications and those that process sensitive information. It's extremely important to consider the security implications of a software development project from the early stages because it's much easier to build security into a system than it is to add security to an existing system.

Programming Languages

As you probably know, software developers use programming languages to develop software code. You might not know that several types of languages can be used simultaneously by the same system. This section takes a brief look at the different types of programming languages and the security implications of each.

Computers understand binary code. They speak a language of 1s and 0s, and that's it. The instructions that a computer follows consist of a long series of binary digits in a language known as *machine language*. Each central processing unit (CPU) chipset has its own machine language, and it's virtually impossible for a human being to decipher anything but the simplest machine language code without the assistance of specialized software. Assembly language is a higher-level alternative that uses mnemonics to represent the basic instruction set of a CPU, but it still requires hardware-specific knowledge of a relatively obscure language. It also requires a large amount of tedious

programming; a task as simple as adding two numbers together could take five or six lines of assembly code!

Programmers don't want to write their code in either machine language or assembly language. They prefer to use high-level languages, such as Python, C, C#, C++, Ruby, R, Java, and Visual Basic. These languages allow programmers to write instructions that better approximate human communication, decrease the length of time needed to craft an application, possibly decrease the number of programmers needed on a project, and allow some portability between different operating systems and hardware platforms. Once programmers are ready to execute their programs, two options are available to them: compilation and interpretation.

Some languages (such as C, Java, and Fortran) are compiled languages. When using a compiled language, the programmer uses a tool known as a *compiler* to convert source code from a higher-level language into an executable file designed for use on a specific operating system. This executable is then distributed to end users, who may use it as they see fit. Generally speaking, it's not possible to directly view or modify the software instructions in an executable file. However, specialists in the field of reverse engineering may be able to reverse the compilation process with the assistance of tools known as *decompilers* and *disassemblers*. Decompilers attempt to take binary executables and convert them back into source code form, whereas disassemblers convert back into machine-readable assembly language (an intermediate step during the compilation process). These tools are particularly useful when you're performing malware analysis or competitive intelligence and you're attempting to determine how an executable file works without access to the underlying source code. Code protection techniques seek to either prevent or impede the use of decompilers and disassemblers through a variety of techniques. For example, obfuscation techniques seek to modify executables to make it more difficult to retrieve intelligible code from them.

In some cases, languages rely on *runtime environments* to allow the portable execution of code across different operating systems.

The Java virtual machine (JVM) is a well-known example of this type of runtime. Users install the JVM runtime on their systems and may then rely on that runtime to execute compiled Java code.

Other languages (such as Python, R, JavaScript, and VBScript) are interpreted languages. When these languages are used, the programmer distributes the source code, which contains instructions in the higher-level language. When end users execute the program on their systems, that automatically triggers the use of an interpreter to execute the source code stored on the system. If the user opens the source code file, they're able to view the original instructions written by the programmer.

Each approach has security advantages and disadvantages. Compiled code is generally less prone to manipulation by a third party. However, it's also easier for a malicious (or unskilled) programmer to embed backdoors and other security flaws in the code and escape detection because the original instructions can't be viewed by the end user. Interpreted code, however, is less prone to the undetected insertion of malicious code by the original programmer because the end user may view the code and check it for accuracy. On the other hand, everyone who touches the software has the ability to modify the programmer's original instructions and possibly embed malicious code in the interpreted software. You'll learn more about the exploits malicious actors use to undermine software in the section “Application Attacks” in Chapter 21, “Malicious Code and Application Attacks.”

Libraries

Developers often rely on shared *software libraries* that contain reusable code. These libraries perform a variety of functions, ranging from text manipulation to machine learning, and are a common way for developers to improve their efficiency. After all, there's no need to write your own code to sort a list of items when you can just use a standard sorting library to do the work for you.

Many of these libraries are available as open-source projects, whereas others may be commercially sold or maintained

internally by a company. Over the years, the use of shared libraries has resulted in many security issues. One of the most well-known and damaging examples of this is the Heartbleed vulnerability (CVE-2014-0160) that struck the OpenSSL library in 2014. The OpenSSL library is a very widely used implementation of Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols that was incorporated into thousands of other systems. In many cases, users of those systems had no idea that they were also using OpenSSL because of this incorporation. When the Heartbleed bug affected OpenSSL libraries, administrators around the world had to scramble to identify and update OpenSSL installations.

To protect against similar vulnerabilities, developers should be aware of the origins of their shared code and keep abreast of any security vulnerabilities that might be discovered in libraries that they use. This doesn't mean that shared libraries are inherently bad. In fact, it's difficult to imagine a world where shared libraries aren't widely used. It simply calls for vigilance and attention from software developers and cybersecurity professionals.

Development Tool Sets

Developers use a variety of tools to help them in their work. Most important among these is the integrated development environment (IDE). IDEs provide programmers with a single environment where they can write their code, test it, debug it, and compile it (if applicable). The IDE simplifies the integration of these tasks, and the choice of an IDE is a personal decision for many developers.

Figure 20.1 shows an example of the open-source RStudio Desktop IDE used with the R programming language.

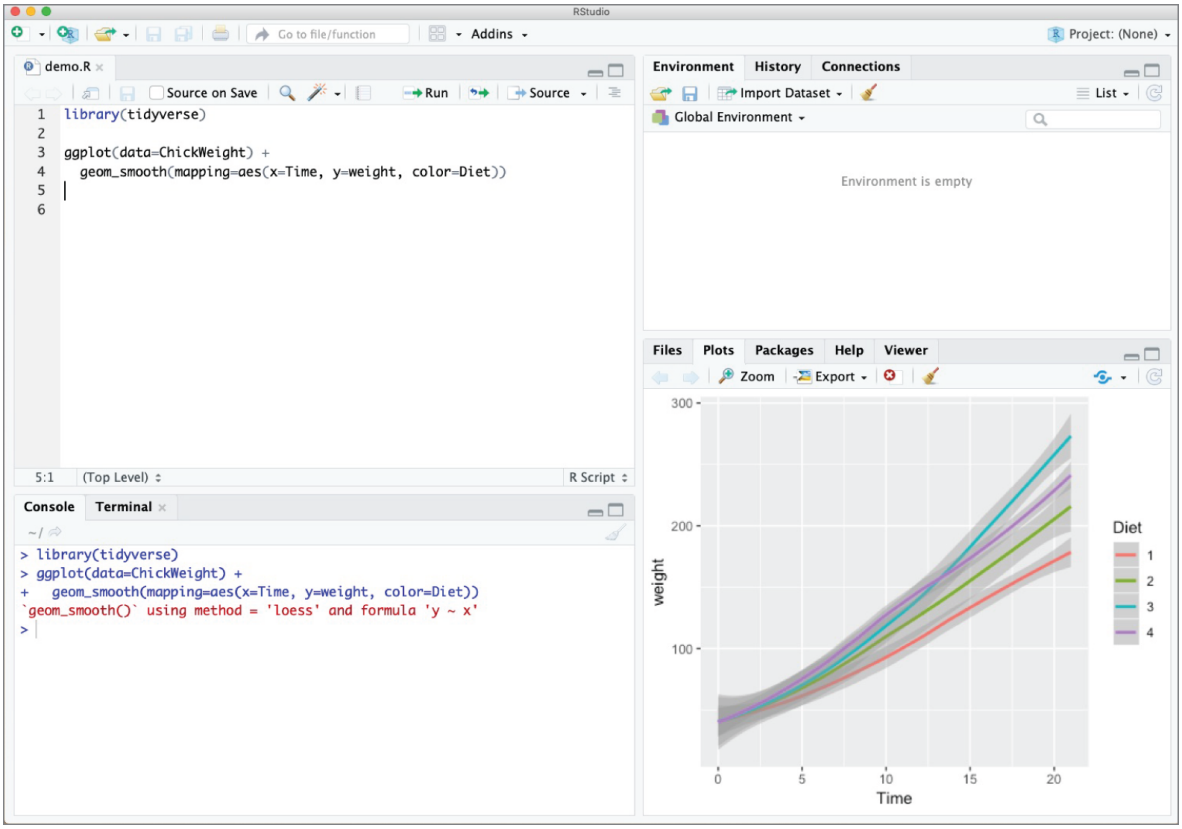

FIGURE 20.1 RStudio Desktop IDE

Object-Oriented Programming

Many modern programming languages, such as C++, Java, and the .NET languages, support the concept of object-oriented programming (OOP). Other programming styles, such as functional programming and scripting, focus on the flow of the program itself and attempt to model the desired behavior as a series of steps. OOP focuses on the objects involved in an interaction. You can think of it as a group of objects that can be requested to perform certain operations or exhibit certain behaviors. Objects work together to provide a system's functionality or capabilities. OOP has the potential to be more reliable and able to reduce the propagation of program change errors. As a type of programming method, it is better suited to modeling or mimicking the real world. For example, a banking program might have three object classes that correspond to accounts, account holders, and employees, respectively. When a new account is added to the system, a new instance, or copy, of

the appropriate object is created to contain the details of that account.

Each object in the OOP model has methods that correspond to specific actions that can be taken on the object. For example, the account object can have methods to add funds, deduct funds, close the account, and transfer ownership.

Objects can also be subclasses of other objects and inherit methods from their parent class. For example, the account object may have subclasses that correspond to specific types of accounts, such as savings, checking, mortgages, and auto loans. The subclasses can use all the methods of the parent class and have additional class-specific methods. For example, the checking object might have a method called `write_check()`, whereas the other subclasses do not.

From a security point of view, object-oriented programming provides a black-box approach to abstraction. Users need to know the details of an object's interface (generally the inputs, outputs, and actions that correspond to each of the object's methods) but don't necessarily need to know the inner workings of the object to use it effectively. To provide the desired characteristics of object-oriented systems, the objects are encapsulated (self-contained), and they can be accessed only through specific messages (in other words, input). Objects can also exhibit the substitution property, which allows different objects providing compatible operations to be substituted for each other.

Here are some common object-oriented programming terms you might come across in your work:

Message A message is a communication to or input of an object.

Method A method is internal code that defines the actions an object performs in response to a message.

Behavior The results or output exhibited by an object is a behavior. Behaviors are the results of a message being processed through a method.

Class A class is a collection of the common methods from a set of objects that defines the behavior of those objects.

Instance Objects are instances of or examples of classes that contain their methods.

Inheritance Inheritance occurs when methods from a class (parent or superclass) are inherited by another subclass (child) or object.

Delegation Delegation is the forwarding of a request by an object to another object or delegate. An object delegates if it does not have a method to handle the message.

Polymorphism A polymorphism is the characteristic of an object that allows it to respond with different behaviors to the same message or method because of changes in external conditions.

Cohesion Cohesion describes the strength of the relationship between the purposes of the methods within the same class. When all the methods have similar purposes, there is high cohesion, a desirable condition that promotes good software design principles. When the methods of a class have low cohesion, this is a sign that the system is not well designed.

Coupling Coupling is the level of interaction between objects. Lower coupling means less interaction. Lower coupling provides better software design because objects are more independent. Lower coupling is easier to troubleshoot and update. Objects that have low cohesion require lots of assistance from other objects to perform tasks and have high coupling.

If you're interested in learning more about the difference between cohesion and coupling, see http://ducmanhphan.github.io/2019-03-23-Coupling-and-Cohension-in-OOP.

Assurance

To ensure that the security control mechanisms built into a new application properly implement the security policy throughout the life cycle of the system, administrators use *assurance procedures*. Assurance procedures are simply formalized processes by which trust is built into the life cycle of a system. The Common Criteria provide a standardized approach to assurance used in government settings. For more information on assurance and the Common Criteria, see Chapter 8, "Principles of Security Models, Design, and Capabilities."

Avoiding and Mitigating System Failure

No matter how advanced your development team, your systems will likely fail at some point in time. You should plan for this type of failure when you put the software and hardware controls in place, ensuring that the system will respond appropriately. You can employ many methods to avoid failure, including using input validation and creating fail-secure or fail-open procedures. Let's talk about these in more detail.

Input Validation As users interact with software, they often provide information to the application in the form of input. This may include typing in values that are later used by a program. Developers often expect these values to fall within certain parameters. For example, if the programmer asks the user to enter a month, the program may expect to see an integer value between 1 and 12. If the user enters a value outside that range, a poorly written program may crash, at best, or allow the user to gain control of the underlying system, at worst.

Input validation verifies that the values provided by a user match the programmer's expectation before allowing further processing. For example, input validation would check whether a month value is an integer between 1 and 12. If the value falls outside that range, the program will not try to process the number as a date and will inform the user of the input expectations. This type of input validation, where the code checks to ensure that a number falls within an acceptable range, is known as a *limit check*.

Input validation also may check for unusual characters, such as single quotation marks within a text field, which may be indicative of an attack. In some cases, the input validation routine can transform the input to remove risky character sequences and replace them with safe values. This process, known as *escaping input*, is performed by replacing occurrences of sensitive characters with alternative code that will render the same to the end user but will not be executed by the system. For example, this HTML code would normally execute a script within the user's browser:

```
<SCRIPT>alert('script executed')</SCRIPT>
```

When we escape this input, we replace the sensitive < and > characters used to create HTML tags. < is replaced with < and > is replaced with > giving us this:

```
<SCRIPT>alert('script executed')</SCRIPT>
```

Input validation should always occur on the server side of the transaction. Any code sent to the user's browser is subject to manipulation by the user and is therefore easily circumvented.

In most organizations, security professionals come from a system administration background and don't have professional experience in software development. If your background doesn't include this type of experience, don't let that stop you from learning about it and educating your organization's developers on the importance of secure coding.

Authentication and Session Management Many applications, particularly web applications, require that users authenticate prior to accessing sensitive information or modifying data in the application. One of the core security tasks facing developers is ensuring that those users are properly authenticated, that they perform only authorized actions, and that their session is securely tracked from start to finish.

The level of authentication required by an application should be tied directly to the level of sensitivity of that application. For example, if an application provides a user with access to sensitive information or allows the user to perform business-critical applications, it should require the use of strong multifactor authentication.

In most cases, developers should seek to integrate their applications with the organization's existing authentication systems. It is generally more secure to make use of an existing, hardened authentication system than to try to develop an authentication system for a specific application. If this is not possible, consider using externally developed and validated authentication libraries.

Similarly, developers should use established methods for session management. This includes ensuring that any cookies used for web session management be transmitted only over secure, encrypted channels and that the identifiers used in those cookies be long and randomly generated. Session tokens should expire after a specified period of time and require that the user reauthenticate.

Error Handling Developers love detailed error messages. The in-depth information returned in those errors is crucial to debugging code and makes it easier for technical staff to diagnose problems experienced by users.

However, those error messages may also expose sensitive internal information to attackers, including the structure of database tables, the addresses of internal servers, and other data that may be useful in reconnaissance efforts that precede an attack. Therefore, developers should disable detailed error messages (also known as debugging mode) on any servers and applications that are publicly accessible.

Logging While user-facing detailed error messages may present a security threat, the information that those messages contain is quite useful, not only to developers but also to cybersecurity analysts. Therefore, applications should be

configured to send detailed logging of errors and other security events to a centralized log repository.

The Open Worldwide Application Security Project (OWASP) Secure Coding Practices suggest logging the following events:

- Input validation failures
- Authentication attempts, especially failures
- Access control failures
- Tampering events, including unexpected changes to state data
- Attempts to connect with invalid or expired session tokens
- All system exceptions
- All administrative functions, including changes to the security configuration settings
- All backend TLS connection failures
- Cryptographic module failures

This information can be useful in diagnosing security issues and in the investigation of security incidents.

Fail-Secure and Fail-Open In spite of the best efforts of programmers, product designers, and project managers, developed applications will be used in unexpected ways. Some of these conditions will cause failures. Since failures are unpredictable, programmers should design into their code a general sense of how to respond to and handle failures.

There are two basic choices when planning for system failure:

- The *fail-secure failure state* puts the system into a high level of security (and possibly even disables it entirely) until an administrator can diagnose the problem and restore the system to normal operation.
- The *fail-open state* allows users to bypass failed security controls, erring on the side of permissiveness.

In the vast majority of environments, fail-secure is the appropriate failure state because it prevents unauthorized access to information and resources.

Software should revert to a fail-secure condition. This may mean closing just the application or possibly stopping the operation of the entire host system. An example of such failure response is seen in the Windows operating system with the appearance of the infamous Blue Screen of Death (BSOD), indicating the occurrence of a STOP error. A STOP error occurs when an undesirable activity occurs in spite of the OS's efforts to prevent it. This could include an application gaining direct access to hardware, an attempt to bypass a security access check, or one process interfering with the memory space of another. Once one of these conditions occurs, the environment is no longer trustworthy. So, rather than continuing to support an unreliable and insecure operating environment, the OS initiates a STOP error as its fail-secure response.

Once a fail-secure operation occurs, the programmer should consider the activities that occur afterward. The options are to remain in a fail-secure state or to automatically reboot the system. The former option requires an administrator to manually reboot the system and oversee the process. This action can be enforced by using a boot password. The latter option does not require human intervention for the system to restore itself to a functioning state, but it has its own unique issues. For example, it must restrict the system to reboot into a nonprivileged state. In other words, the system should not reboot and perform an automatic logon; instead, it should prompt the user for authorized access credentials.

In limited circumstances, it may be possible to implement a fail-open failure state. This is sometimes appropriate for lower-layer components of a multilayered security system. Fail-open systems should be used with extreme caution. Before deploying a system using this failure mode, clearly validate the business requirement for this move. If it is justified, ensure that adequate alternative controls are in place to protect the organization's resources should the system fail. It's extremely rare that you'd want all your security controls to use a fail-open approach.

Even when security is properly designed and embedded in software, that security is often disabled in order to support easier installation. Thus, it is common for the IT administrator to have the responsibility of turning on and configuring security to match the needs of their specific environment. Maintaining security is often a trade-off with user-friendliness and functionality, as you can see in Figure 20.2. Additionally, as you add or increase security, you will also increase costs, increase administrative overhead, and reduce productivity/throughput.

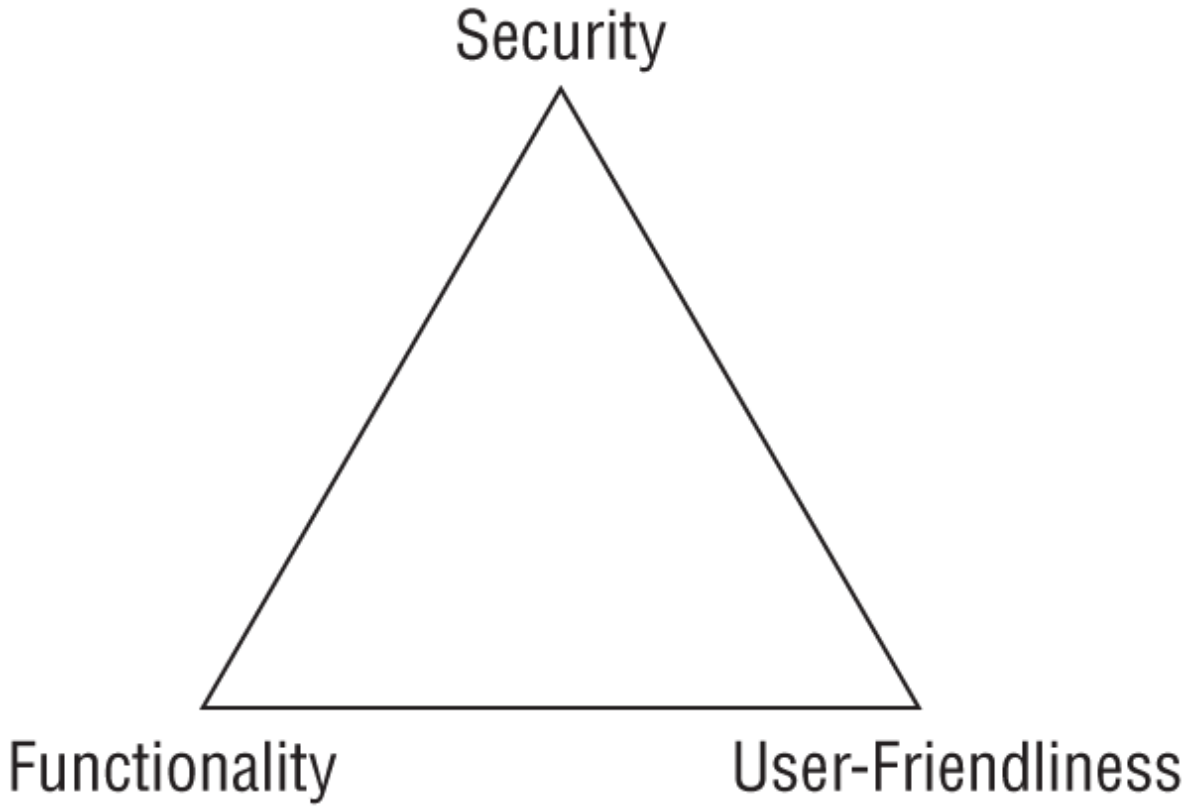

FIGURE 20.2 Security vs. user-friendliness vs. functionality

Systems Development Life Cycle

Security is most effective if it is planned and managed throughout the life cycle of a system or application. Administrators employ project management to keep a development project on target and moving toward the goal of a completed product. Often project management is structured using life cycle models to direct the development process. Using formalized life cycle models helps ensure good coding practices and the embedding of security in every stage of product development.

All systems development processes should have several activities in common. Although they may not necessarily share the same names, these core activities are essential to the development of sound, secure systems:

- Conceptual definition
- Functional requirements determination

- Control specifications development
- Design review
- Coding
- Code review walk-through
- System test review
- Maintenance and change management

The section “Life Cycle Models,” later in this chapter, examines two life cycle models and shows how these activities are applied in real-world software engineering environments.

At this point, the terminology used in systems development life cycles varies from model to model and from publication to publication. Don't spend too much time worrying about the exact terms used in this book or any of the other literature you may come across. When you take the CISSP examination, it's much more important that you have an understanding of how the process works and of the fundamental principles underlying the development of secure systems.

Conceptual Definition

The conceptual definition phase of systems development involves creating the basic concept statement for a system. It's a simple statement agreed on by all interested stakeholders (the developers, customers, and management) that states the purpose of the project as well as the general system requirements. The conceptual definition is a very high-level statement of purpose and should not be longer than one or two paragraphs. If you were reading a detailed summary of the project, you might expect to see the concept statement as an abstract or introduction that enables an outsider to gain a top-level understanding of the project in a short period of time.

The security requirements developed at this phase are generally very high level. They will be refined during the control specifications development phase. At this point in the process, designers commonly identify the classification(s) of data that will be processed by the system and the applicable handling requirements.

It's helpful to refer to the concept statement at all phases of the systems development process. Often, the intricate details of the development process tend to obscure the overarching goal of the project. Simply reading the concept statement periodically can assist in refocusing a team of developers.

Functional Requirements Determination

Once all stakeholders have agreed on the concept statement, it's time for the development team to sit down and begin the functional requirements process. In this phase, specific system functionalities are listed, and developers begin to think about how the parts of the system should interoperate to meet the functional requirements. The deliverable from this phase of development is a functional requirements document that lists the specific system requirements. These requirements should be expressed in a form consumable by software developers. The following are the three major characteristics of a functional requirement:

Input(s) The data provided to a function

Behavior The business logic describing what actions the system should take in response to different inputs

Output(s) The data provided from a function

As with the concept statement, it's important to ensure that all stakeholders agree on the functional requirements document before work progresses to the next level. When it's finally completed, the document shouldn't be simply placed on a shelf to gather dust—the entire development team should constantly refer to this document during all phases to ensure that the project is on track. In the final stages of testing and evaluation,

the project managers should use this document as a checklist to ensure that all functional requirements are met.

Control Specifications Development

Security-conscious organizations also ensure that adequate security controls are designed into every system from the earliest stages of development. It's often useful to have a control specifications development phase in your life cycle model. This phase takes place soon after the development of functional requirements and often continues as the design and design review phases progress.

During the development of control specifications, you should analyze the system from a number of security perspectives. First, adequate access controls must be designed into every system to ensure that only authorized users are allowed to access the system and that they are not permitted to exceed their level of authorization. Second, the system must maintain the confidentiality of vital data through the use of appropriate encryption and data protection technologies. Next, the system should provide both an audit trail to enforce individual accountability and a detection mechanism for illegitimate activity. Finally, depending on the criticality of the system, availability and fault-tolerance issues should be addressed as corrective actions.

Keep in mind that designing security into a system is not a onetime process and it must be done proactively. All too often, systems are designed without security planning, and then developers attempt to retrofit the system with appropriate security mechanisms. Unfortunately, these mechanisms are an afterthought and do not fully integrate with the system's design, which leaves gaping security vulnerabilities. Also, the security requirements should be revisited each time a significant change is made to the design specifications. If a major component of the system changes, it's likely that the security requirements will change as well.

Design Review

Once the functional and control specifications are complete, let the system designers do their thing. In this often-lengthy process, the designers determine exactly how the various parts of the system will interoperate and how the modular system structure will be laid out. Also, during this phase the design management team commonly sets specific tasks for various teams and lays out initial timelines for the completion of coding milestones.

After the design team completes the formal design documents, a review meeting with the stakeholders should be held to ensure that everyone is in agreement that the process is still on track for the successful development of a system with the desired functionality. This design review meeting should include security professionals who can validate that the proposed design meets the control specifications developed in the previous phase.

Coding

Once the stakeholders have given the software design their blessing, it's time for the software developers to start writing code. Developers should use the secure software coding principles discussed in this chapter to craft code that is consistent with the agreed-on design and meets user requirements.

Code Review Walk-Through

Project managers should schedule several code review walk-through meetings at various milestones throughout the coding process. These technical meetings usually involve only development personnel, who sit down with a copy of the code for a specific module and walk through it, looking for problems in logical flow or other design/security flaws. The meetings play an instrumental role in ensuring that the code produced by the various development teams performs according to specification.

System Test Review

After many code reviews and a lot of long nights, there will come a point at which a developer puts in that final semicolon and declares the system complete. As any seasoned software engineer

knows, the system is never complete. Initially, most organizations perform the initial *system testing* using development personnel to seek out any obvious errors. As the testing progresses, developers and actual users validate the system against predefined scenarios that model common and unusual user activities. In cases where the project is releasing updates to an existing system, *regression testing* formalizes the process of verifying that the new code performs in the same manner as the old code, other than any changes expected as part of the new release. These testing procedures should include both functional testing that verifies the software is working properly and security testing that verifies there are no unaddressed significant security issues.

Once developers are satisfied that the code works properly, the process moves into *user acceptance testing* (UAT), where users verify that the code meets their requirements and formally accept it as ready to move into production use.

Once this phase is complete, the code may move to deployment. As with any critical development process, it's important that you maintain a copy of the written test plan and test results for future review.

Maintenance and Change Management

Once a system is operational, a variety of maintenance tasks are necessary to ensure continued operation in the face of changing operational, data processing, storage, and environmental requirements. It's essential that you have a skilled support team in place to handle any routine or unexpected maintenance. It's also important that any changes to the code be handled through a formalized change management process, as described in Chapter 1, “Security Governance Through Principles and Policies.”

Life Cycle Models

One of the major complaints you'll hear from practitioners of the more established engineering disciplines (such as civil, mechanical, and electrical engineering) is that software

engineering is not an engineering discipline at all. In fact, they contend, it's simply a combination of chaotic processes that somehow manage to scrape out workable solutions from time to time. Indeed, some of the “software engineering” that takes place in today's development environments is nothing but bootstrap coding held together by “duct tape and chicken wire.”

However, the adoption of more formalized life cycle management processes is seen in mainstream software engineering as the industry matures. After all, it's hardly fair to compare the processes of a centuries-old discipline such as civil engineering to those of an industry that's still in its first century of existence. In the 1970s and 1980s, pioneers like Winston Royce and Barry Boehm proposed several software development life cycle (SDLC) models to help guide the practice toward formalized processes. In 1991, the Software Engineering Institute published the Capability Maturity Model, which described the process that organizations undertake as they move toward incorporating solid engineering principles into their software development processes. In the following sections, we'll take a look at the work produced by these studies. Having a management model in place should improve the resultant products. However, if the SDLC methodology is inadequate, the project may fail to meet business and user needs. Thus, it is important to verify that the SDLC model is properly implemented and is appropriate for your environment. Furthermore, one of the initial steps of implementing an SDLC should include management approval.

Choosing an SDLC model is normally the work of software development teams and their leadership. Cybersecurity professionals should ensure that security principles are interwoven into the implementation of whatever model(s) the organization uses for software development.

Waterfall Model

Originally developed by Winston Royce in 1970, the waterfall model seeks to view the systems development life cycle as a series of sequential activities. The traditional waterfall model has seven stages of development. As each stage is completed, the

project moves into the next stage. The original, traditional waterfall model was a simple design that was intended to be sequential steps from inception to conclusion. In practical application, the waterfall model, of necessity, evolved to a more modern model. As illustrated by the backward arrows in Figure 20.3, the iterative waterfall model does allow development to return to the previous phase to correct defects discovered during the subsequent phase. This is often known as the *feedback loop characteristic* of the waterfall model.

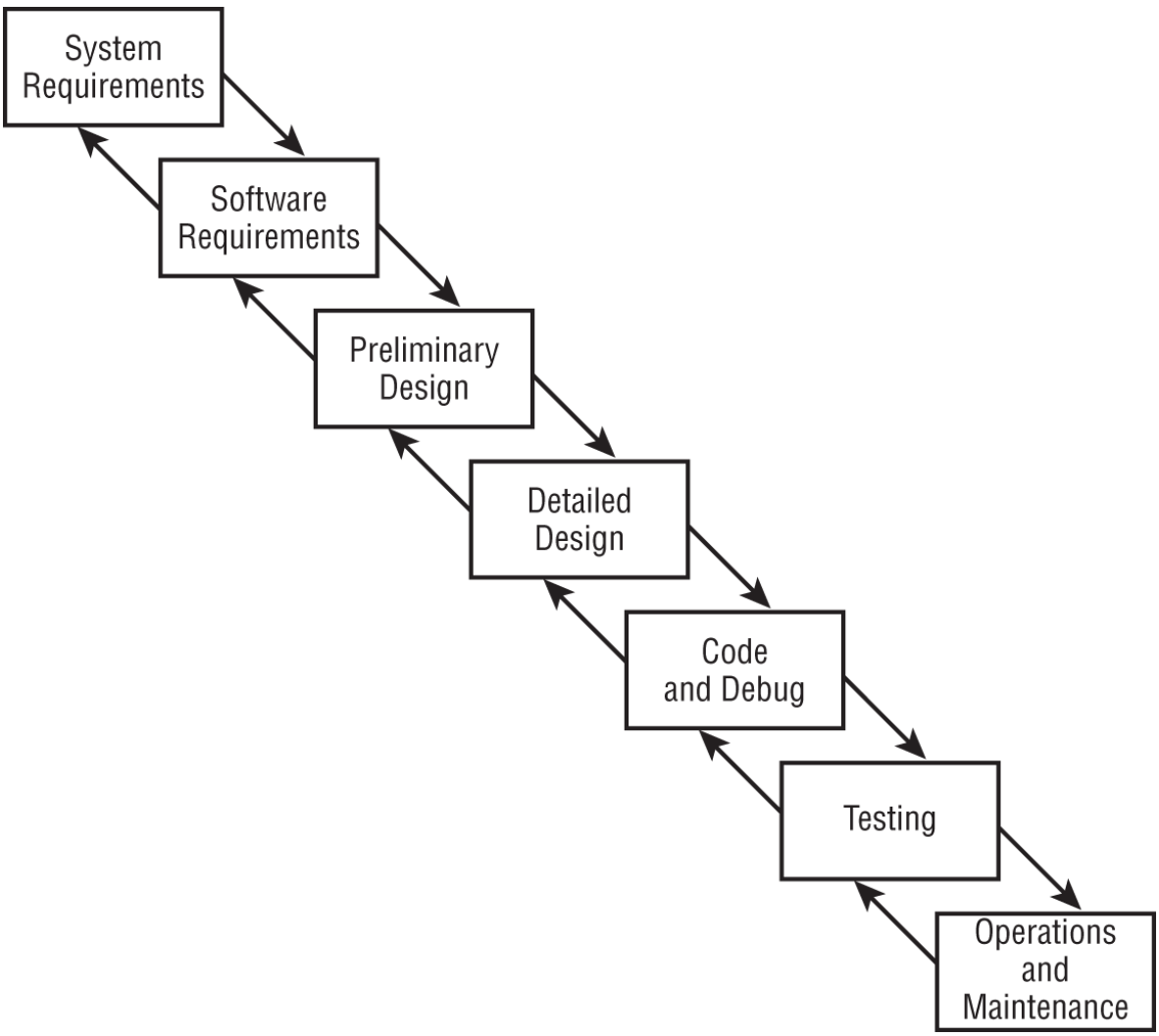

FIGURE 20.3 The iterative life cycle model with feedback loop

The waterfall model was one of the first comprehensive attempts to model the software development process while taking into account the necessity of returning to previous phases to correct system faults. However, one of the major criticisms of this model is that it allows the developers to step back only one phase in the

process. It does not make provisions for the discovery of errors at a later phase in the development cycle.

The waterfall model was improved by adding validation and verification steps to each phase. Verification evaluates the product against specifications, whereas validation evaluates how well the product satisfies real-world requirements. The improved model was labeled the *modified* waterfall model. However, it did not gain widespread use before the spiral model dominated the project management scene.

Spiral Model

In 1988, Barry Boehm of TRW proposed an alternative life cycle model that allows for multiple iterations of a waterfall-style process. Figure 20.4 illustrates this model. Because the spiral model encapsulates a number of iterations of another model (the waterfall model), it is known as a *metamodel*, or a "model of models."

Notice that each "loop" of the spiral results in the development of a new system prototype (represented by P1, P2, and P3 in Figure 20.4). Theoretically, system developers would apply the entire waterfall process to the development of each prototype, thereby incrementally working toward a mature system that incorporates all the functional requirements in a fully validated fashion. Boehm's spiral model provides a solution to the major criticism of the waterfall model—it allows developers to return to the planning stages as changing technical demands and customer requirements necessitate the evolution of a system. The waterfall model focuses on a large-scale effort to deliver a finished system, whereas the spiral model focuses on iterating through a series of increasingly "finished" prototypes that allow for enhanced quality control.

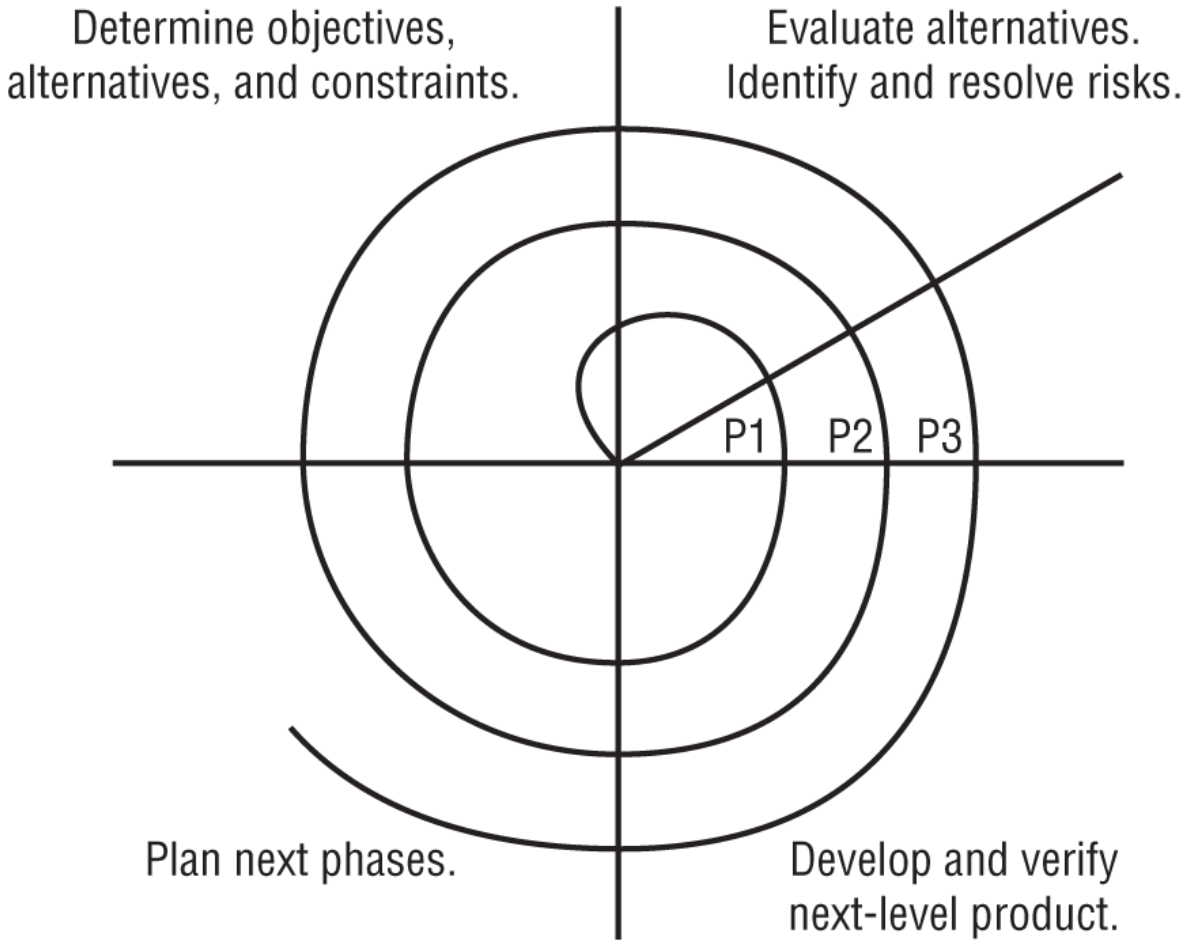

FIGURE 20.4 The spiral life cycle mode

Agile Software Development

More recently, the Agile model of software development has gained popularity within the software engineering community. Beginning in the mid-1990s, developers increasingly embraced approaches to software development that eschewed the rigid models of the past in favor of approaches that placed an emphasis on the needs of the customer and on quickly developing new functionality that meets those needs in an iterative fashion.

Seventeen pioneers of the Agile development approach got together in 2001 and produced a document titled *Manifesto for Agile Software Development* (http://agilemanifesto.org) that states the core philosophy of the Agile approach:

> We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:
>
> **Individuals and interactions** over processes and tools
>
> **Working software** over comprehensive documentation
>
> **Customer collaboration** over contract negotiation
>
> **Responding to change** over following a plan
>
> That is, while there is value in the items on the right, we value the items on the left more.

The *Agile Manifesto* also defines 12 principles that underlie the philosophy, which are available here: http://agilemanifesto.org/principles.

The 12 principles, as stated in the Agile Manifesto, are as follows:

- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
- Business people and developers must work together daily throughout the project.
- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
- Working software is the primary measure of progress.

- Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
- Continuous attention to technical excellence and good design enhances agility.
- Simplicity—the art of maximizing the amount of work not done—is essential.
- The best architectures, requirements, and designs emerge from self-organizing teams.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Today, most software developers embrace the flexibility and customer focus of the Agile approach to software development, and it is quickly becoming the philosophy of choice for developers. In an Agile approach, the team embraces the principles of the Agile Manifesto and meets regularly to review and plan their work.

It's important to note, however, that Agile is a philosophy and not a specific methodology. Several specific methodologies have emerged that take these Agile principles and define specific processes that implement them. These include Scrum, Kanban, Lean, Rapid Application Development (RAD), Agile Unified Process (AUP), the Dynamic Systems Development Model (DSDM), and Extreme Programming (XP).

Of these, the *Scrum* approach is the most popular. Scrum takes its name from the daily team meetings, called *scrums*, that are its hallmark. Each day the team gets together for a short meeting, where they discuss the contributions made by each team member, plan the next day's work, and work to clear any impediments to their progress. These meetings are led by the project's *scrum master*, an individual in a project management role who is responsible for helping the team move forward and meet their objectives.

The Scrum methodology organizes work into short *sprints* of activity. These are well-defined periods of time, typically between one and four weeks, where the team focuses on achieving short-term objectives that contribute to the broader goals of the project. At the beginning of each sprint, the team gathers to plan the work that will be conducted during each sprint. At the end of the sprint, the team should have a fully functioning product that could be released, even if it does not yet meet all user requirements. Each subsequent sprint introduces new functionality into the product.

Integrated Product Teams

Although the Agile concept is a product of recent years, the idea of bringing together stakeholders for software and system development is a long-standing concept. The Department of Defense introduced the idea of integrated product teams (IPTs) in 1995 as an approach to bring together multifunctional teams with a single goal of delivering a product or developing a process or policy. In the guidance creating IPTs, the Defense Department said that "IPTs are set up to foster parallel, rather than sequential, decisions and to guarantee that all aspects of the product, process, or policy are considered throughout the development process."

Scaled Agile Framework (SAFe)

The *Scaled Agile Framework (SAFe)* is a comprehensive approach to applying agile principles and practices at the enterprise scale. Recognizing the complexities that larger organizations face, SAFe offers a structured framework to facilitate the use of Agile across multiple teams, often coordinating the efforts of hundreds or even thousands of practitioners.

At its core, SAFe seeks to provide a shared understanding of how work flows through an organization, ensuring alignment from team-level activities to strategic goals. SAFe is organized into four configuration levels:

1. *Essential SAFe:* This is where the traditional Agile practices, like Scrum, come into play. Teams work in Agile Release Trains (ARTs), which are groups of teams that align to deliver larger pieces of value, often in the form of program increments. Each program increment typically lasts around 8–12 weeks.
2. *Large Solution SAFe:* For particularly vast systems that require multiple ARTs, this SAFe configuration provides additional roles and artifacts to ensure alignment and coordination. This isn't always needed but comes into play in exceptionally large implementations.
3. *Portfolio SAFe:* This SAFe configuration is where strategic direction is translated into actionable items. Investment themes guide the organization's work, ensuring alignment with business objectives. Lean Portfolio Management (LPM) principles drive the efforts, ensuring minimal overhead and focusing on delivering the maximum value.
4. *Full SAFe:* Full SAFe includes all elements of Essential, Large Solution, and Portfolio SAFe, along with additional guidance to help organizations achieve business agility. It's designed to support enterprises that build and maintain large, integrated solutions that require hundreds of practitioners to collaborate effectively.

One of the standout features of SAFe is its emphasis on aligning business goals with development activity. It introduces concepts like Epic, Capability, Feature, and Story, to break down and categorize tasks, ensuring every piece of work can be traced back to a larger objective.

SAFe Principles

The SAFe framework is based on 10 core principles drawn from Agile philosophy:

1. Take an economic view.
2. Apply systems thinking.
3. Assume variability; preserve options.
4. Build incrementally with fast, integrated learning cycles.
5. Base milestones on objective evaluation of working systems.
6. Make value flow without interruptions.
7. Apply cadence; synchronize with cross-domain planning.
8. Unlock the intrinsic motivation of knowledge workers.
9. Decentralize decision-making.
10. Organize around value.

SAFe offers a comprehensive and modular approach to scaling Agile, accommodating the intricacies of larger organizations. It integrates principles from multiple disciplines to provide a holistic method of managing work, ensuring alignment from the strategic level right down to individual teams. While its complexity might not be suitable for every organization, those with the need for structured coordination across large teams often find immense value in its practices.

Capability Maturity Model (CMM)

The Software Engineering Institute (SEI) at Carnegie Mellon University introduced the Capability Maturity Model for Software, also known as the Software Capability Maturity Model (abbreviated as SW-CMM, CMM, or SCMM), which contends that all organizations engaged in software development move through a variety of maturity phases in sequential fashion. The SW-CMM

describes the principles and practices underlying software process maturity. It is intended to help software organizations improve the maturity and quality of their software processes by implementing an evolutionary path from ad hoc, chaotic processes to mature, disciplined software processes. The idea behind the SW-CMM is that the quality of software depends on the quality of its development process. SW-CMM does not explicitly address security, but it is the responsibility of cybersecurity professionals and software developers to ensure that security requirements are integrated into the software development effort.

The stages of the SW-CMM are as follows:

Level 1: Initial In this phase, you'll often find hardworking people charging ahead in a disorganized fashion. There is usually little or no defined software development process.

Level 2: Repeatable In this phase, basic life cycle management processes are introduced. Reuse of code in an organized fashion begins to enter the picture, and repeatable results are expected from similar projects. SEI defines the key process areas for this level as Requirements Management, Software Project Planning, Software Project Tracking and Oversight, Software Subcontract Management, Software Quality Assurance, and Software Configuration Management.

Level 3: Defined In this phase, software developers operate according to a set of formal, documented software development processes. All development projects take place within the constraints of the new standardized management model. SEI defines the key process areas for this level as Organization Process Focus, Organization Process Definition, Training Program, Integrated Software Management, Software Product Engineering, Intergroup Coordination, and Peer Reviews.

Level 4: Managed In this phase, management of the software process proceeds to the next level. Quantitative measures are used to gain a detailed understanding of the development process. SEI defines the key process areas for this level as

Quantitative Process Management and Software Quality Management.

Level 5: Optimizing In the optimized organization, a process of continuous improvement occurs. Sophisticated software development processes are in place that ensure that feedback from one phase reaches to the previous phase to improve future results. SEI defines the key process areas for this level as Defect Prevention, Technology Change Management, and Process Change Management.

CMM has largely been superseded by a new model called the Capability Maturity Model Integration (CMMI). The CMMI uses the same five stages as the CMM but calls level 4 Quantitatively Managed, rather than Managed. The major difference between CMM and CMMI is that CMM focuses on isolated processes, whereas CMMI focuses on the integration among those processes.

Software Assurance Maturity Model (SAMM)

The Software Assurance Maturity Model (SAMM) is an open-source project maintained by OWASP. It seeks to provide a framework for integrating security activities into the software development and maintenance process and to offer organizations the ability to assess their maturity.

SAMM divides the software development process into five business functions:

Governance The activities an organization undertakes to manage its software development process. This function includes practices for strategy, metrics, policy, compliance, education, and guidance.

Design The process used by the organization to define software requirements and create software. This function includes practices for threat modeling, threat assessment, security requirements, and security architecture.

Implementation The process of building and deploying software components and managing flaws in those components. This function includes the secure build, secure deployment, and defect management practices.

Verification The set of activities undertaken by the organization to confirm that code meets business and security requirements. This function includes architecture assessment, requirements-driven testing, and security testing.

Operations The actions taken by an organization to maintain confidentiality, integrity, and availability throughout the software life cycle after code is released. This function includes incident management, environment management, and operational management.

Each of these business functions is then broken out by applicable security practices, as shown in Figure 20.5.

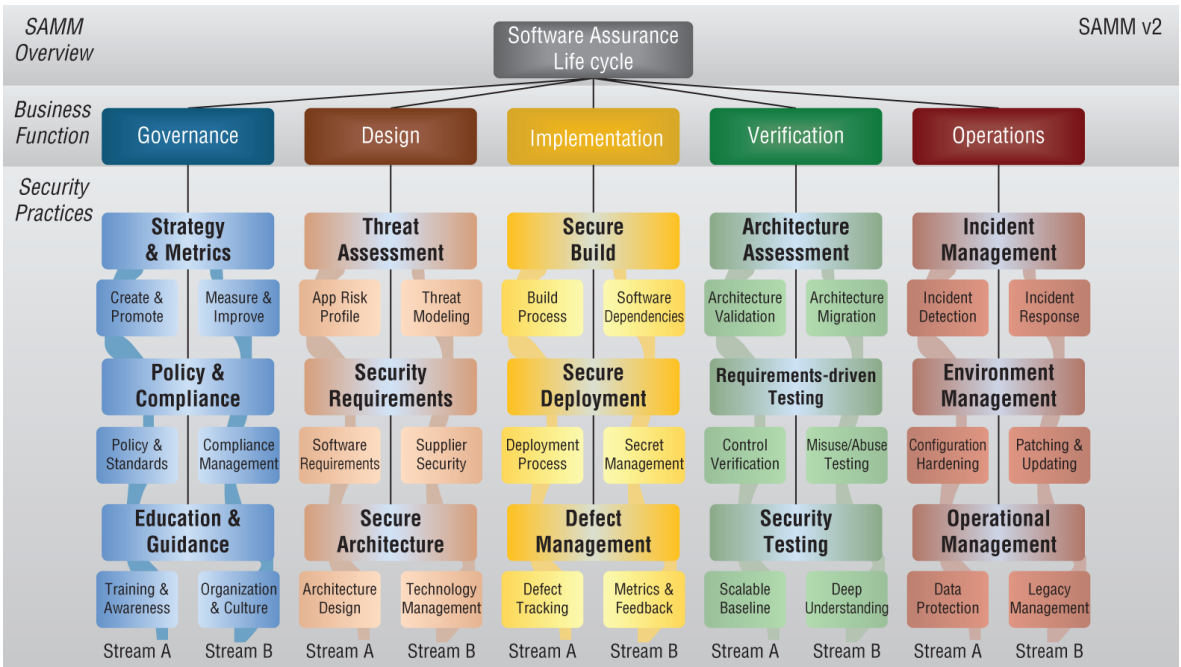

FIGURE 20.5 Software Assurance Maturity Model

IDEAL Model

The Software Engineering Institute also developed the IDEAL model for software development, which implements many of the SW-CMM attributes. The IDEAL model has five phases:

1. *Initiating:* In the initiating phase of the IDEAL model, the business reasons behind the change are outlined, support is built for the initiative, and the appropriate infrastructure is put in place.
2. *Diagnosing:* During the diagnosing phase, engineers analyze the current state of the organization and make general recommendations for change.
3. *Establishing:* In the establishing phase, the organization takes the general recommendations from the diagnosing phase and develops a specific plan of action that helps achieve those changes.
4. *Acting:* In the acting phase, it's time to stop "talking the talk" and "walk the walk." The organization develops solutions and then tests, refines, and implements them.
5. *Learning:* As with any quality improvement process, the organization must continuously analyze its efforts to determine whether it has achieved the desired goals, and when necessary, propose new actions to put the organization back on course.

The IDEAL model is illustrated in Figure 20.6.

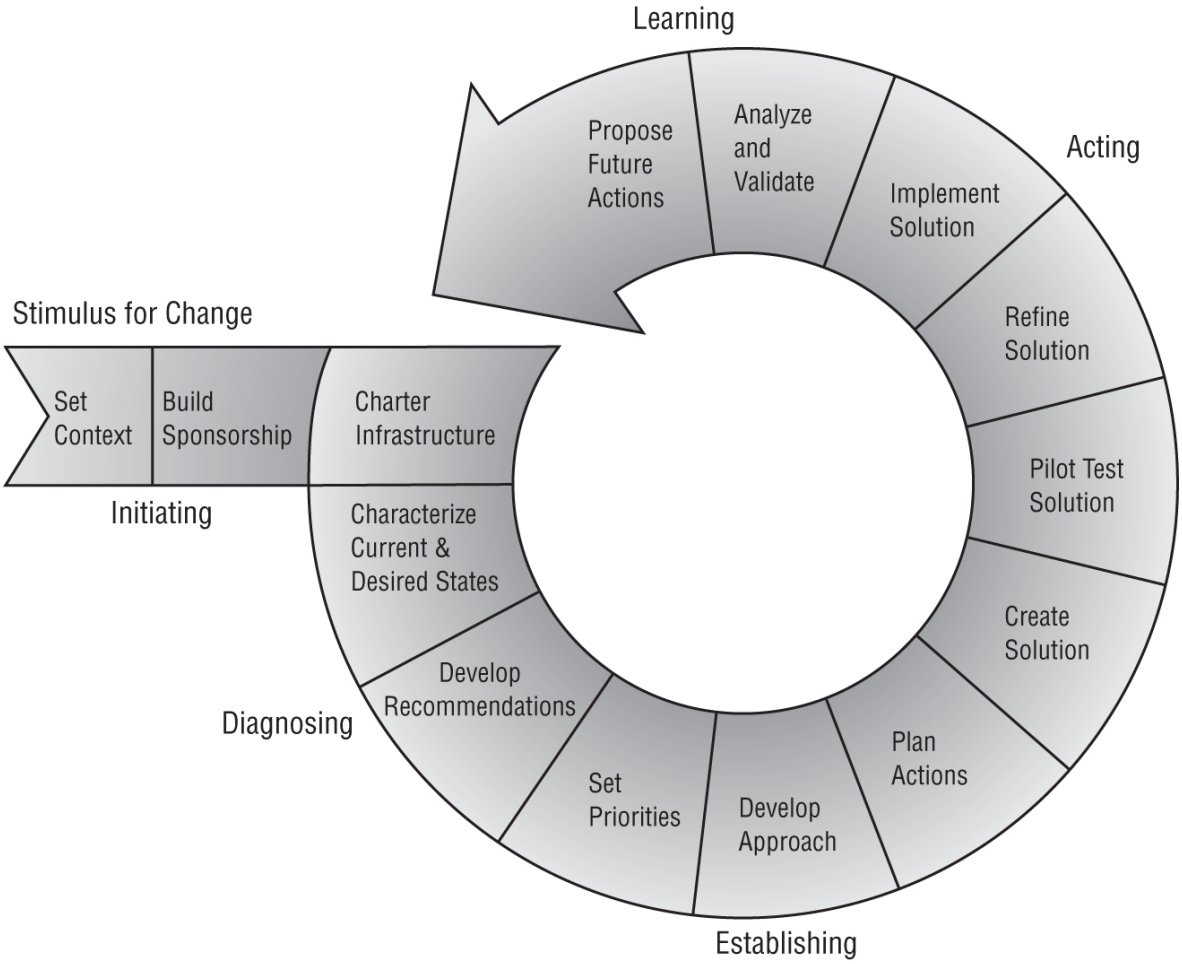

FIGURE 20.6 The IDEAL model

Source: IDEAL Model, 2004 / Carnegie Mellon University.

SW-CMM and IDEAL Model Memorization

To help you remember the initial letters of each of the 10 level names of the SW-CMM and IDEAL models (II DR ED AM LO), imagine yourself sitting on the couch in a psychiatrist's office saying, “I...I, Dr. Ed, am lo(w).” If you can remember that phrase, then you can extract the 10 initial letters of the level names. If you write the letters out into two columns, you can reconstruct the level names in order of the two systems. The left column is the IDEAL model, and the right represents the levels of the SW-CMM.

IDEAL Phases	SW-CMM Phases
Initiating	Initial

IDEAL Phases	SW-CMM Phases
Diagnosing	Repeatable
Establishing	Defined
Acting	Managed
Learning	Optimizing

Gantt Charts and PERT

A *Gantt chart* is a type of bar chart that shows the interrelationships over time between projects and schedules. It provides a graphical illustration of a schedule that helps you plan, coordinate, and track specific tasks in a project. Gantt charts are particularly useful when coordinating tasks that require the use of the same team members or other resources. Figure 20.7 shows an example of a Gantt chart.

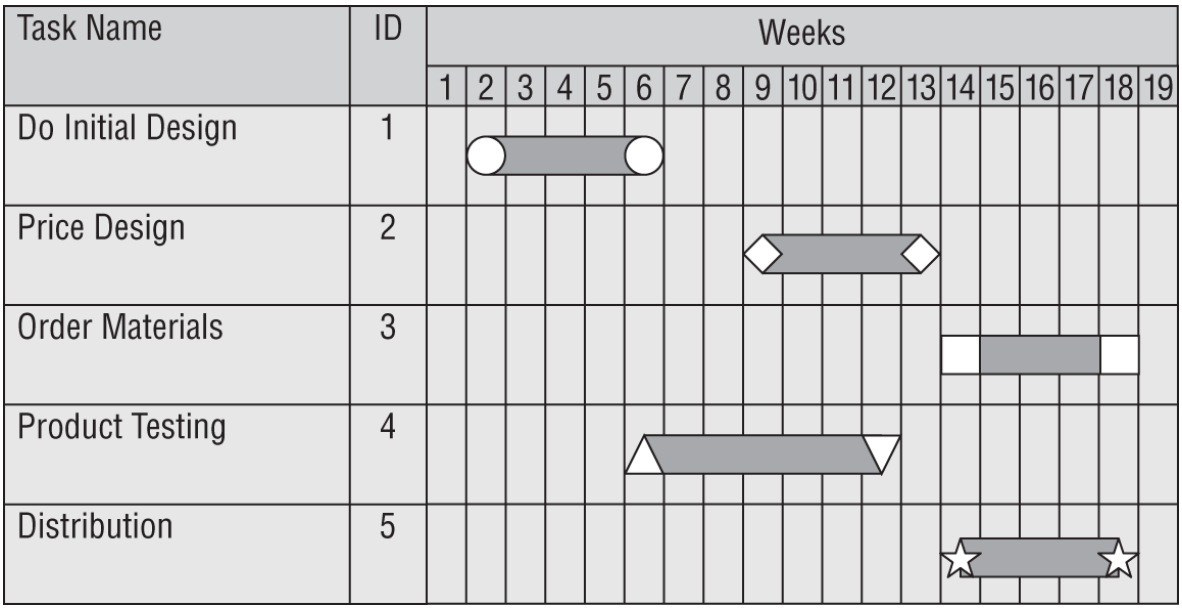

FIGURE 20.7 Gantt chart

Program Evaluation Review Technique (PERT) is a project-scheduling tool used to judge the size of a software product in development and calculate the standard deviation (SD) for risk assessment. PERT relates the estimated lowest possible size, the most likely size, and the highest possible size of each component. The PERT chart clearly shows the dependencies between

different project tasks. Project managers can use these size estimates and dependencies to better manage the time of team members and perform task scheduling. PERT is used to direct improvements to project management and software coding in order to produce more efficient software. As the capabilities of programming and management improve, the actual produced size of software should be smaller.

Change and Configuration Management

Once software has been released into a production environment, users will inevitably request the addition of new features, correction of bugs, and other modifications to the code. Just as the organization developed a regimented process for developing software, they must also put a procedure in place to manage changes in an organized fashion. Those changes should then be logged to a central repository to support future auditing, investigation, troubleshooting, and analysis requirements.

Real World Scenario

Change Management as a Security Tool

Change management (also known as control management) plays an important role when monitoring systems in the controlled environment of a data center. One of the authors recently worked with an organization that used change management as an essential component of its efforts to detect unauthorized changes to computing systems.

File integrity monitoring tools allow you to monitor a system for changes. This organization used such a tool to monitor hundreds of production servers. However, the organization quickly found itself overwhelmed by file modification alerts resulting from normal activity. The author worked with them to tune the file integrity monitoring policies and integrate them with the organization's change management process. Now all file integrity alerts go to a centralized monitoring center, where administrators correlate them with approved changes. System administrators receive an alert only if the security team identifies a change that does not appear to correlate with an approved change request.

This approach greatly reduced the time spent by administrators reviewing file integrity reports and improved the usefulness of the tool to security administrators.

The change management process has three basic components:

Request Control The request control process provides an organized framework within which users can request modifications, managers can conduct cost/benefit analysis, and developers can prioritize tasks.

Change Control The change control process is used by developers to re-create the situation encountered by the user and to analyze the appropriate changes to remedy the situation. It

also provides an organized framework within which multiple developers can create and test a solution prior to rolling it out into a production environment. Change control includes conforming to quality control restrictions, developing tools for update or change deployment, properly documenting any coded changes, and restricting the effects of new code to minimize diminishment of security.

Release Control Once the changes are finalized, they must be approved for release through the release control procedure. An essential step of the release control process is to double-check and ensure that any code inserted as a programming aid during the change process (such as debugging code and/or backdoors) is removed before releasing the new software to production. This process also ensures that only approved changes are made to production systems. Release control should also include acceptance testing to ensure that any alterations to end-user work tasks are understood and functional.

In addition to the change management process, security administrators should be aware of the importance of *software configuration management (SCM)*. This process is used to control the version(s) of software used throughout an organization and to formally track and control changes to the software configuration. It has four main components:

Configuration Identification During the configuration identification process, administrators document the configuration of covered software products throughout the organization.

Configuration Control The configuration control process ensures that changes to software versions are made in accordance with the change control and configuration management policies. Updates can be made only from authorized distributions in accordance with those policies.

Configuration Status Accounting Formalized procedures are used to keep track of all authorized changes that take place.

Configuration Audit A periodic configuration audit should be conducted to ensure that the actual production environment is

consistent with the accounting records and that no unauthorized configuration changes have taken place.

Together, change and configuration management techniques form an important part of the software engineer's arsenal and protect the organization from development-related security issues.

The DevOps Approach

Recently, many technology professionals recognized a disconnect between the major IT functions of software development, quality assurance, and technology operations. These functions, typically staffed with very different types of individuals and located in separate organizational silos, often conflicted with one another. This conflict resulted in lengthy delays in creating code, testing it, and deploying it onto production systems. When problems arose, instead of working together to cooperatively solve the issue, teams often “threw problems over the fence” at one another, resulting in bureaucratic back and forth.

The DevOps approach seeks to resolve these issues by bringing the three functions together in a single operational model. The word *DevOps* is a combination of Development and Operations, symbolizing that these functions must merge and cooperate to meet business requirements. The model in Figure 20.8 illustrates the overlapping nature of software development, quality assurance, and IT operations.

FIGURE 20.8 The DevOps model

The DevOps model is closely aligned with the Agile development approach and aims to dramatically decrease the time required to develop, test, and deploy software changes. Although traditional approaches often resulted in major software deployments on an infrequent basis, perhaps annually, organizations using the DevOps model often deploy code several times per day. Some organizations even strive to reach the goal of *continuous integration/continuous delivery (CI/CD),* where code may roll out dozens or even hundreds of times per day. This requires a high degree of automation, including integrating code repositories, the software configuration management process, and the movement of code between development, testing, and production environments.

If you're interested in learning more about DevOps, the authors highly recommend the book *The Phoenix Project: A Novel About IT, DevOps, and Helping Your Business Win* by Gene Kim, Kevin Behr, and George Spafford (IT Revolution Press, 2013). This book presents the case for DevOps and shares DevOps strategies in an entertaining, engaging novel form.

The tight integration of development and operations also calls for the simultaneous integration of security controls. If code is being rapidly developed and moved into production, security must also move with that same agility. For this reason, many people prefer to use the term *DevSecOps* to refer to the integration of development, security, and operations. The DevSecOps approach also supports the concept of *software-defined security*, where security controls are actively managed by code, allowing them to be directly integrated into the CI/CD pipeline.

Application Programming Interfaces

Although early web applications were often stand-alone systems that processed user requests and provided output, modern web applications are much more complex. They often include interactions between a number of different web services. For example, a retail website might make use of an external credit card processing service, allow users to share their purchases on social media, integrate with shipping provider sites, and offer a referral program on other websites.

For these cross-site functions to work properly, the websites must interact with one another. Many organizations offer *application programming interfaces (APIs)* for this purpose. APIs allow application developers to bypass traditional web pages and interact directly with the underlying service through function calls. For example, a social media API might include some of the following API function calls:

- Post status
- Follow user
- Unfollow user
- Like/Favorite a post

Offering and using APIs creates tremendous opportunities for service providers, but it also poses some security risks. Developers must be aware of these challenges and address them when they create and use APIs.

First, developers must consider authentication requirements. Some APIs, such as those that allow checking weather forecasts or product inventory, may be available to the general public and not require any authentication for use. Other APIs, such as those that allow modifying information, placing orders, or accessing sensitive information, may be limited to specific users and depend on secure authentication. API developers must know when to require authentication and ensure that they verify credentials and authorization for every API call. This authentication is typically done by providing authorized API users with a complex API key that is passed with each API call. The backend system validates this API key before processing a request, ensuring that the system making the request is authorized to make the specific API call.

API keys are like passwords and should be treated as sensitive information. They should always be stored in secure locations and transmitted only over encrypted communications channels. If someone gains access to your API key, they can interact with a web service as if they were you.

`curl` is an open-source tool available for major operating systems that allows users to directly access websites without the use of a browser. For this reason, `curl` is commonly used for API testing

and also for potential API exploits by an attacker. For example, consider this `curl` command:

```
curl -H "Content-Type: application/json" -X POST -d
'{"week": 10,
"hrv": 80, "sleephrs": 9, "sleepquality": 2, "stress": 3,
"paxid": 1
}'https://prod.myapi.com/v1
```

The purpose of this command is to send a `POST` request to the URL https://prod.myapi.com/v1 that contains information being sent to the API in JSON format. You don't need to worry about the format of this command as you prepare for the exam, but you should be familiar with the concept that `curl` may be used to post requests to an API.

APIs must also be tested thoroughly for security flaws, just like any web application. You'll learn more about this in the next section.

Software Testing

As part of the development process, your organization should thoroughly test any software before distributing it internally (or releasing it to market). This testing is a crucial component of the risk analysis and mitigation efforts associated with software development. The organization uses comprehensive testing to identify potential risks and mitigates them by modifying code and/or adopting compensating controls.

The best time to address testing is as the modules are designed. In other words, the mechanisms you use to test a product and the datasets you use to explore that product should be designed in parallel with the product itself. Your programming team should develop special test suites of data that exercise all paths of the software to the fullest extent possible and know the correct resulting outputs beforehand.

One of the tests you should perform is a *reasonableness check*. The reasonableness check ensures that values returned by software match specified criteria that are within reasonable bounds. For example, a routine that calculated optimal weight for

a human being and returned a value of 612 pounds would certainly fail a reasonableness check!

Furthermore, while conducting software testing, you should check how the product handles normal and valid input data, incorrect types, out-of-range values, and other bounds and/or conditions. Live workloads provide the best stress testing possible. However, you should not use live or actual field data for testing, especially in the early development stages, since a flaw or error could result in the violation of integrity or confidentiality of the test data. This process should involve the use of both *use cases*, which mirror normal activity, and *misuse cases*, which attempt to model the activity of an attacker. Including both of these approaches helps testers understand how the code will perform under normal activity (including normal errors) and when subjected to the extreme conditions imposed by an attacker.

When testing software, you should apply the same rules of separation of duties that you do for other aspects of your organization. In other words, you should assign the testing of your software to someone other than the programmer(s) who developed the code to avoid a conflict of interest and assure a more secure and functional finished product. When a third party tests your software, you have a greater likelihood of receiving an objective and nonbiased examination. The third-party test allows for a broader and more thorough test and prevents the bias and inclinations of the programmers from affecting the results of the test.

There are three different philosophies that you can adopt when applying software security testing techniques:

White-Box Testing White-box testing examines the internal logical structures of a program and steps through the code line by line, analyzing the program for potential errors. The key attribute of a white-box test is that the testers have access to the source code.

Black-Box Testing Black-box testing examines the program from a user perspective by providing a wide variety of input

scenarios and inspecting the output. Black-box testers do not have access to the internal code. Final acceptance testing that occurs prior to system delivery is a common example of black-box testing.

Gray-Box Testing Gray-box testing combines the two approaches and is popular for software validation. In this approach, testers examine the software from a user perspective, analyzing inputs and outputs. They also have access to the source code and use it to help design their tests. They do not, however, analyze the inner workings of the program during their testing.

In addition to assessing the quality of software, programmers and security professionals should carefully assess the security of their software to ensure that it meets the organization's security requirements. This assessment is especially critical for web applications that are exposed to the public. For more on code review and testing techniques, such as static and dynamic testing, see Chapter 15, "Security Assessment and Testing."

Proper software test implementation is a key element in the project development process. Many of the common mistakes and oversights often found in commercial and in-house software can be eliminated. Keep the test plan and results as part of the system's permanent documentation.

Code Repositories

Software development is a collaborative effort, and large software projects require teams of developers who may simultaneously work on different parts of the code. Further complicating the situation is the fact that these developers may be geographically dispersed around the world.

Code repositories provide several important functions supporting these collaborations. Primarily, they act as a central storage point for developers to place their source code. In addition, code repositories such as GitHub, Bitbucket, and SourceForge also provide version control, bug tracking, web hosting, release management, and communications functions that support software development. Code repositories are often integrated

with popular code management tools. For example, the `git` tool is popular among many software developers, and it is tightly integrated with GitHub and other repositories.

Earlier in this chapter, you learned about code libraries. Libraries are packages of reusable code that may be shared within an organization or with the public. Repositories are broader platforms that provide the tools for shared software development and distribution. Repositories may be used to manage and distribute code libraries.

Code repositories are wonderful collaborative tools that facilitate software development, but they also have security risks of their own. First, developers must appropriately control access to their repositories. Some repositories, such as those supporting open-source software development, may allow public access. Others, such as those hosting code containing trade secret information, may be more limited, restricting access to authorized developers. Repository owners must carefully design access controls to only allow appropriate users read and/or write access. Improperly granting users read access may allow unauthorized individuals to retrieve sensitive information, whereas improperly granting write access may allow unauthorized tampering with code.

Sensitive Information and Code Repositories

Developers must take care not to include sensitive information in public code repositories. This is particularly true of API keys.

Many developers use APIs to access the underlying functionality of infrastructure-as-a-service (IaaS) providers, such as Amazon Web Services (AWS), Microsoft Azure, and Google Compute Engine. This provides tremendous benefits, allowing developers to quickly provision servers, modify network configuration, and allocate storage using simple API calls.

Of course, IaaS providers charge for these services. When a developer provisions a server, it triggers an hourly charge for that server until it is shut down. The API key used to create a server ties the server to a particular user account (and credit card).

If developers write code that includes API keys and then upload that key to a public repository, anyone in the world can then gain access to their API key. This allows anyone to create IaaS resources and charge it to the original developer's credit card.

Further worsening the situation, malicious actors have written bots that scour public code repositories searching for exposed API keys. These bots may detect an inadvertently posted key in seconds, allowing the malicious actor to quickly provision massive computing resources before the developer even knows of their mistake.

Similarly, developers should also be careful to avoid placing passwords, internal server names, database names, and other sensitive information in code repositories.

Service-Level Agreements

Using service-level agreements (SLAs) is an increasingly popular way to ensure that organizations providing services to internal and/or external customers maintain an appropriate level of service agreed on by both the service provider and the vendor. It's a wise move to put SLAs in place for any data circuits, applications, information processing systems, databases, or other critical components that are vital to your organization's continued viability. The following issues are commonly addressed in SLAs:

- System uptime (as a percentage of overall operating time)
- Maximum consecutive downtime (in seconds/minutes/and so on)
- Peak load
- Average load
- Responsibility for diagnostics
- Failover time (if redundancy is in place)

Service-level agreements also commonly include financial and other contractual remedies that kick in if the agreement is not maintained. In these situations, the service provider and customer both carefully monitor performance metrics to ensure compliance with the SLA. For example, if a critical circuit is down for more than 15 minutes, the service provider might be required to waive all charges on that circuit for one week.

Third-Party Software Acquisition

Most of the software used by enterprises is not developed internally but purchased from third-party vendors. *Commercial off-the-shelf (COTS)* software is purchased to run on servers managed by the organization, either on-premises or in an IaaS environment. Other software is purchased and delivered over the Internet through web browsers, in a software-as-a-service (SaaS) approach. Still more software is created and maintained by

community-based *open-source software (OSS)* projects. These open-source projects are freely available for anyone to download and use, either directly or as a component of a larger system. In fact, many COTS software packages incorporate open-source code. Most organizations use a combination of commercial and open-source, depending on business needs and software availability.

For example, organizations may approach email service in two ways. They might purchase physical or virtual servers and then install email software, such as Microsoft Exchange, on them. In that case, the organization purchases Exchange licenses from Microsoft and then installs, configures, and manages the email environment.

As an alternative, the organization might choose to outsource email entirely to Google, Microsoft, or another vendor. Users then access email through their web browsers or other tools, interacting directly with the email servers managed by the vendor. In this case, the organization is only responsible for creating accounts and managing some application-level settings.

In either case, security is of paramount concern. When the organization purchases and configures software itself, security professionals must understand the proper configuration of that software to meet security objectives. They also must remain vigilant about security bulletins and patches that correct newly discovered vulnerabilities. Failure to meet these obligations may result in an insecure environment.

In the case of SaaS environments, most security responsibility rests with the vendor, but the organization's security staff isn't off the hook. Although they might not be responsible for as much configuration, they now take on responsibility for monitoring the vendor's security. This may include audits, assessments, vulnerability scans, and other measures designed to verify that the vendor maintains proper controls. The organization may also retain full or partial responsibility for legal compliance obligations, depending on the nature of the regulation and the agreement that is in place with the service provider.

Whenever an organization acquires any type of software, be it COTS or OSS, run on-premises or in the cloud, that software should be tested for security vulnerabilities. Organizations may conduct their own testing, rely on the results of tests provided by vendors, and/or hire third parties to conduct independent testing.

Establishing Databases and Data Warehousing

Almost every modern organization maintains some sort of database that contains information critical to operations—be it customer contact information, order-tracking data, human resource and benefits information, or sensitive trade secrets. It's likely that many of these databases contain personal information that users hold secret, such as credit card usage activity, travel habits, grocery store purchases, and telephone records. Because of the growing reliance on database systems, information security professionals must ensure that adequate security controls exist to protect them against unauthorized access, tampering, or destruction of data.

In the following sections, we'll discuss database management system (DBMS) architecture, including the various types of DBMSs and their features. Then, we'll discuss database security considerations, including polyinstantiation, Open Database Connectivity (ODBC), aggregation, inference, and machine learning.

Database Management System Architecture

Although a variety of DBMS architectures are available today, the vast majority of contemporary systems implement a technology known as relational database management systems (RDBMSs). For this reason, the following sections focus primarily on

relational databases. However, first we'll discuss two other important DBMS architectures: hierarchical and distributed.

Hierarchical and Distributed Databases

A hierarchical data model combines records and fields that are related in a logical tree structure. This results in a one-to-many data model, where each node may have zero, one, or many children but only one parent. An example of a hierarchical data model appears in Figure 20.9.

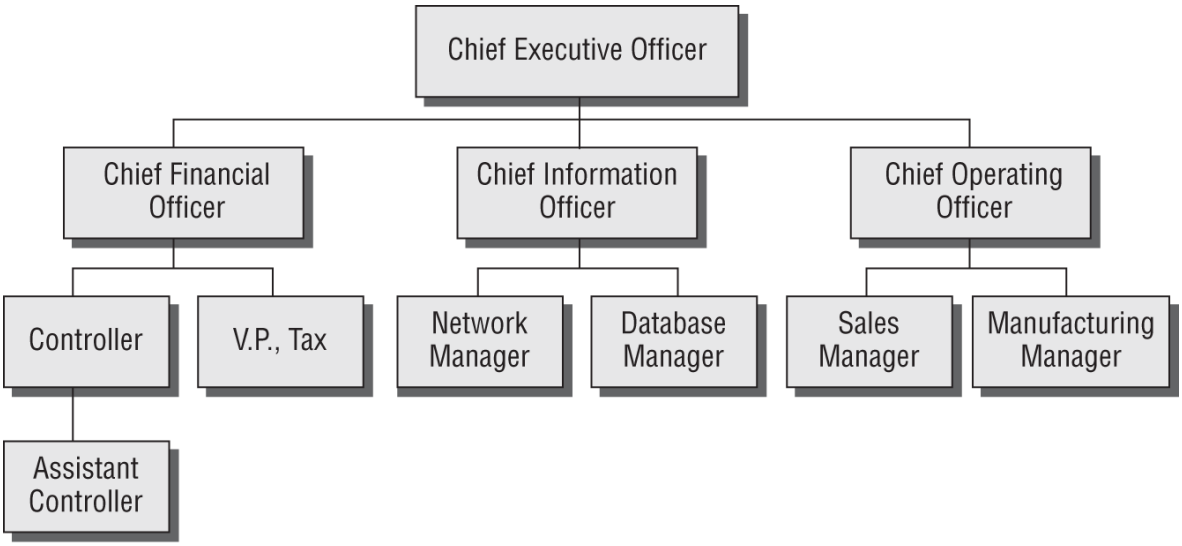

FIGURE 20.9 Hierarchical data model

The hierarchical model in Figure 20.9 is a corporate organization chart. Notice that the one-to-many data model holds true in this example. Each employee has only one manager (the *one* in *one-to-many*), but each manager may have one or more (the *many*) employees. Other examples of hierarchical data models include the NCAA March Madness bracket system and the hierarchical distribution of Domain Name System (DNS) records used on the Internet. Hierarchical databases store data in this type of hierarchical fashion and are useful for specialized applications that fit the model. For example, biologists might use a hierarchical database to store data on specimens according to the kingdom/phylum/class/order/family/genus/species hierarchical model used in that field.

The distributed data model has data stored in more than one database, but those databases are logically connected. The user

perceives the database as a single entity, even though it consists of numerous parts interconnected over a network. Each field can have numerous children as well as numerous parents. Thus, the data mapping relationship for distributed databases is many-to-many.

Relational Databases

A relational database consists of flat two-dimensional tables made up of rows and columns. In fact, each table looks similar to a spreadsheet file. The row and column structure provides for one-to-one data mapping relationships. The main building block of the relational database is the table (also known as a *relation*). Each table contains a set of related records. For example, a sales database might contain the following tables:

- Customers table that contains contact information for all the organization's clients
- Sales Reps table that contains identity information on the organization's sales force
- Orders table that contains records of orders placed by each customer

Object-Oriented Programming and Databases

Object-relational databases combine relational databases with the power of object-oriented programming. True object-oriented databases (OODBs) benefit from ease of code reuse, ease of troubleshooting analysis, and reduced overall maintenance. OODBs are also better suited than other types of databases for supporting complex applications involving multimedia, CAD, video, graphics, and expert systems.

Each table contains a number of attributes, or *fields*. Each attribute corresponds to a column in the table. For example, the Customers table might contain columns for company name,

address, city, state, zip code, and telephone number. Each customer would have their own record, or *tuple*, represented by a row in the table. The number of rows in the relation is referred to as *cardinality*, and the number of columns is the *degree*. The *domain* of an attribute is the set of allowable values that the attribute can take. Figure 20.10 shows an example of a Customers table from a relational database.

Company ID	Company Name	Address	City	State	ZIP Code	Telephone	Sales Rep
1	Acme Widgets	234 Main Street	Columbia	MD	21040	(301) 555-1212	14
2	Abrams Consulting	1024 Sample Street	Miami	FL	33131	(305) 555-1995	14
3	Dome Widgets	913 Sorin Street	South Bend	IN	46556	(574) 555-5863	26

FIGURE 20.10 Customers table from a relational database

In this example, the table has a cardinality of 3 (corresponding to the three rows in the table) and a degree of 8 (corresponding to the eight columns). It's common for the cardinality of a table to change during the course of normal business, such as when a sales rep adds new customers. The degree of a table normally does not change frequently and usually requires database administrator intervention.

To remember the concept of cardinality, think of a deck of cards on a desk, with each card (the first four letters of *cardinality*) being a row. To remember the concept of degree, think of a wall thermometer as a column (in other words, the temperature in degrees as measured on a thermometer).

Relationships between the tables are defined to identify related records. In this example, a relationship exists between the Customers table and the Sales Reps table because each customer is assigned a sales representative and each sales representative is assigned to one or more customers. This relationship is reflected by the Sales Rep field/column in the Customers table, shown in Figure 20.10. The values in this column refer to a Sales Rep ID field contained in the Sales Rep table (not shown). Additionally, a relationship would probably exist between the Customers table and the Orders table because each order must be associated with

a customer, and each customer is associated with one or more product orders. The Orders table (not shown) would likely contain a Customer field that contained one of the Customer ID values shown in Figure 20.10.

Records are identified using a variety of keys. Quite simply, *keys* are a subset of the fields of a table and are used to uniquely identify records. They are also used to join tables when you wish to cross-reference information. You should be familiar with four types of keys:

Candidate Keys A *candidate key* is a subset of attributes that can be used to uniquely identify any record in a table. No two records in the same table will ever contain the same values for all attributes composing a candidate key. Each table may have one or more candidate keys, which are chosen from column headings.

Primary Keys A *primary key* is selected from the set of candidate keys for a table to be used to uniquely identify the records in a table. Each table has only one primary key, selected by the database designer from the set of candidate keys. The RDBMS enforces the uniqueness of primary keys by disallowing the insertion of multiple records with the same primary key. In the Customers table shown in Figure 20.10, the Company ID would likely be the primary key.

Alternate Keys Any candidate key that is not selected as the primary key is referred to as an *alternate key*. For example, if the telephone number is unique to a customer in Figure 20.10, then Telephone could be considered a candidate key. Since Company ID was selected as the primary key, then Telephone is an alternate key.

Foreign Keys A *foreign key* is used to enforce relationships between two tables, also known as *referential integrity*. Referential integrity ensures that if one table contains a foreign key, it corresponds to a still-existing primary key in the other table in the relationship. It makes certain that no record/tuple/row contains a reference to a primary key of a nonexistent record/tuple/row. In the example described earlier,

the Sales Rep field shown in Figure 20.10 is a foreign key referencing the primary key of the Sales Reps table.

All relational databases use a standard language, SQL, to provide users with a consistent interface for the storage, retrieval, and modification of data and for administrative control of the DBMS. Each DBMS vendor implements a slightly different version of SQL (like Microsoft's Transact-SQL and Oracle's PL/SQL), but all support a core feature set. SQL's primary security feature is its granularity of authorization. This means that SQL allows you to set permissions at a very fine level of detail. You can limit user access by table, row, column, or even by individual cell in some cases.

Database Normalization

Database developers strive to create well-organized and efficient databases. To assist with this effort, they've defined several levels of database organization known as *normal forms*. The process of bringing a database table into compliance with normal forms is known as *normalization*.

Although a number of normal forms exist, the three most common are first normal form (1NF), second normal form (2NF), and third normal form (3NF). Each of these forms adds requirements to reduce redundancy in the tables, eliminate misplaced data, and perform a number of other housekeeping tasks. The normal forms are cumulative—in other words, to be in 2NF, a table must first be 1NF compliant. Before making a table 3NF compliant, it must first be in 2NF.

The details of normalizing a database table are beyond the scope of the CISSP exam, but several web resources can help you understand the requirements of the normal forms in greater detail. For example, refer to the article “Database Normalization — in Easy to Understand English”: www.essentialsql.com/database-normalization.

SQL provides the complete functionality necessary for administrators, developers, and end users to interact with the database. In fact, the graphical database interfaces popular today merely wrap some extra bells and whistles around a standard SQL interface to the DBMS. SQL itself is divided into two major components: the Data Definition Language (DDL), which allows for the creation and modification of the database's structure (known as the *schema*), and the Data Manipulation Language (DML), which allows users to interact with the data contained within that schema.

Database Transactions

Relational databases support the explicit and implicit use of transactions to ensure data integrity. Each transaction is a discrete set of SQL instructions that should either succeed or fail as a group. It's not possible for one part of a transaction to succeed while another part fails. Consider the example of a transfer between two accounts at a bank. You might use the following SQL code to first add $250 to account 1001 and then subtract $250 from account 2002:

```
BEGIN TRANSACTION
UPDATE accounts
SET balance = balance + 250
WHERE account_number = 1001;

UPDATE accounts
SET balance = balance - 250
WHERE account_number = 2002

END TRANSACTION
```

Imagine a case where these two statements were not executed as part of a transaction but were instead executed separately. If the database failed during the moment between completion of the first transaction and completion of the second transaction, $250 would have been added to account 1001, but there would be no corresponding deduction from account 2002. The $250 would have appeared out of thin air. Flipping the order of the two statements wouldn't help—this would cause $250 to disappear

into thin air if interrupted. This simple example underscores the importance of transaction-oriented processing.

When a transaction successfully finishes, it is said to be committed to the database and cannot be undone. Transaction committing may be explicit, using SQL's `COMMIT` command, or it can be implicit if the end of the transaction is successfully reached. If a transaction must be aborted, it can be rolled back explicitly using the `ROLLBACK` command or implicitly if there is a hardware or software failure. When a transaction is rolled back, the database restores itself to the condition it was in before the transaction began.

Relational database transactions have four required characteristics: atomicity, consistency, isolation, and durability. Together, these attributes are known as the *ACID model*, which is a critical concept in the development of database management systems. Let's take a brief look at each of these requirements:

Atomicity Database transactions must be atomic—that is, they must be an "all-or-nothing" affair. If any part of the transaction fails, the entire transaction must be rolled back as if it never occurred.

Consistency All transactions must begin operating in an environment that is consistent with all of the database's rules (for example, all records have a unique primary key). When the transaction is complete, the database must again be consistent with the rules, regardless of whether those rules were violated during the processing of the transaction itself. No other transaction should ever be able to use any inconsistent data that might be generated during the execution of another transaction.

Isolation The isolation principle requires that transactions operate separately from each other. If a database receives two SQL transactions that modify the same data, one transaction must be completed in its entirety before the other transaction is allowed to modify the same data. This prevents one transaction from working with invalid data generated as an intermediate step by another transaction.

Durability Database transactions must be durable. That is, once they are committed to the database, they must be preserved. Databases ensure durability through the use of backup mechanisms, such as transaction logs.

In the following sections, we'll discuss a variety of specific security issues of concern to database developers and administrators.

Security for Multilevel Databases

As you learned in Chapter 1, many organizations use data classification schemes to enforce access control restrictions based on the security labels assigned to data objects and individual users. When mandated by an organization's security policy, this classification concept must also be extended to the organization's databases.

Multilevel security databases contain information at a number of different classification levels. They must verify the labels assigned to users and, in response to user requests, provide only information that's appropriate. However, this concept becomes somewhat more complicated when considering security for a database.

When multilevel security is required, it's essential that administrators and developers strive to keep data with different security requirements separate. Mixing data with different classification levels and/or need-to-know requirements, known as *database contamination*, is a significant security challenge. Often, administrators will deploy a trusted front end to add multilevel security to a legacy or insecure DBMS.

Real World Scenario

Restricting Access with Views

Another way to implement multilevel security in a database is through the use of database views. Views are simply SQL statements that present data to the user as if the views were tables themselves. Views may be used to collate data from multiple tables, aggregate individual records, or restrict a user's access to a limited subset of database attributes and/or records.

Views are stored in the database as SQL commands rather than as tables of data. This dramatically reduces the space requirements of the database and allows views to violate the rules of normalization that apply to tables. However, retrieving data from a complex view can take significantly longer than retrieving it from a table because the DBMS may need to perform calculations to determine the value of certain attributes for each record.

Because views are so flexible, many database administrators use them as a security tool—allowing users to interact only with limited views rather than with the raw tables of data underlying them.

Concurrency

Concurrency, or edit control, is a preventive security mechanism that endeavors to make certain that the information stored in the database is always correct or at least has its integrity and availability protected. This feature can be employed on a single-level or multilevel database.

Databases that fail to implement concurrency correctly may suffer from the following issues:

Lost Updates Occur when two different processes make updates to a database, unaware of each other's activity. For example, imagine an inventory database in a warehouse with different receiving stations. The warehouse might currently have 10 copies of the *CISSP Study Guide* in stock. If two different receiving stations each receive a copy of the *CISSP Study Guide* at the same time, they both might check the current inventory level, find that it is 10, increment it by 1, and update the table to read 11, when the actual value should be 12.

Dirty Reads Occur when a process reads a record from a transaction that did not successfully commit. Returning to our warehouse example, if a receiving station begins to write new inventory records to the database but then crashes in the middle of the update, it may leave partially incorrect information in the database if the transaction is not completely rolled back.

Concurrency uses a "lock" feature to allow one user to make changes but deny other users access to views or make changes to data elements at the same time. Then, after the changes have been made, an "unlock" feature restores the ability of other users to access the data they need. In some instances, administrators will use concurrency with auditing mechanisms to track document and/or field changes. When this recorded data is reviewed, concurrency becomes a detection control.

Aggregation

SQL provides a number of functions that combine records from one or more tables to produce potentially useful information. This process is called *aggregation*. Aggregation is not without its security vulnerabilities. Aggregation attacks are used to collect numerous low-level security items or low-value items and combine them to create something of a higher security level or value. In other words, malicious actors may be able to collect multiple facts about or from a system and then use these facts to launch an attack.

These functions, although extremely useful, also pose a risk to the security of information in a database. For example, suppose a low-level military records clerk is responsible for updating

records of personnel and equipment as they are transferred from base to base. As part of their duties, this clerk may be granted the database permissions necessary to query and update personnel tables.

The military might not consider an individual transfer request (in other words, Sergeant Jones is being moved from Base X to Base Y) to be classified information. The records clerk has access to that information because they need it to process Sergeant Jones's transfer. However, with access to aggregate functions, the records clerk might be able to count the number of troops assigned to each military base around the world. These force levels are often closely guarded military secrets, but the low-ranking records clerk could deduce them by using aggregate functions across a large number of unclassified records.

For this reason, it's especially important for database security administrators to strictly control access to aggregate functions and adequately assess the potential information they may reveal to unauthorized individuals. Combining defense-in-depth, need-to-know, and least privilege principles help prevent access aggregation attacks.

Inference

The database security issues posed by inference attacks are similar to those posed by the threat of data aggregation. Inference attacks involve combining several pieces of nonsensitive information to gain access to information that should be classified at a higher level. However, inference makes use of the human mind's deductive capacity rather than the raw mathematical ability of modern database platforms.

A commonly cited example of an inference attack is that of the accounting clerk at a large corporation who is allowed to retrieve the total amount the company spends on salaries for use in a top-level report but is not allowed to access the salaries of individual employees. The accounting clerk often has to prepare those reports with effective dates in the past and so is allowed to access the total salary amounts for any day in the past year. Say, for example, that this clerk must also know the hiring and

termination dates of various employees and has access to this information. This opens the door for an inference attack. If an employee was the only person hired on a specific date, the accounting clerk can now retrieve the total salary amount on that date and the day before and deduce the salary of that particular employee—sensitive information that the user would not be permitted to access directly.

As with aggregation, the best defense against inference attacks is to maintain constant vigilance over the permissions granted to individual users. Furthermore, intentional blurring of data may be used to prevent the inference of sensitive information. For example, if the accounting clerk were able to retrieve only salary information rounded to the nearest million, they would probably not be able to gain any useful information about individual employees. Finally, you can use database partitioning (discussed in the next section) to help subvert these attacks.

Other Security Mechanisms

Administrators can deploy several other security mechanisms when using a DBMS. These features are relatively easy to implement and are common in the industry. The mechanisms related to semantic integrity, for instance, are common security features of a DBMS. Semantic integrity ensures that user actions don't violate any structural rules. It also checks that all stored data types are within valid domain ranges, ensures that only logical values exist, and confirms that the system complies with any and all uniqueness constraints.

Administrators may employ time and date stamps to maintain data integrity and availability. Time and date stamps often appear in distributed database systems. When a timestamp is placed on all change transactions and those changes are distributed or replicated to the other database members, all changes are applied to all members, but they are implemented in correct chronological order.

Another common security feature of a DBMS is that objects can be controlled granularly within the database; this can also improve security control. Content-dependent access control is an

example of granular object control and is based on the contents or payload of the object being accessed. Because decisions must be made on an object-by-object basis, content-dependent control increases processing overhead. Another form of granular control is *cell suppression*. Cell suppression is the concept of hiding individual database fields or cells or imposing more security restrictions on them.

Context-dependent access control is often discussed alongside content-dependent access control because of the similarity of the terms. Context-dependent access control evaluates the big picture to make access control decisions. The key factor in context-dependent access control is how each object or packet or field relates to the overall activity or communication. Any single element may look innocuous by itself, but in a larger context that element may be revealed to be benign or malign.

Administrators might employ database partitioning to subvert aggregation and inference vulnerabilities. Database partitioning is the process of splitting a single database into multiple parts, each with a unique and distinct security level or type of content.

Polyinstantiation, in the context of databases, occurs when two or more rows in the same relational database table appear to have identical primary key elements but contain different data for use at differing classification levels. Polyinstantiation is often used as a defense against some types of inference attacks, but it introduces additional storage costs to store copies of data designed for different clearance levels.

Consider a database table containing the location of various naval ships on patrol. Normally, this database contains the exact position of each ship stored at the secret classification level. However, one particular ship, the *USS UpToNoGood*, is on an undercover mission to a top-secret location. Military commanders do not want anyone to know that the ship deviated from its normal patrol. If the database administrators simply change the classification of the *UpToNoGood*'s location to top secret, a user with a secret clearance would know that something unusual was going on when they couldn't query the location of the ship. However, if polyinstantiation is used, two records could

be inserted into the table. The first one, classified at the top-secret level, would reflect the true location of the ship and be available only to users with the appropriate top-secret security clearance. The second record, classified at the secret level, would indicate that the ship was on routine patrol and would be returned to users with a secret clearance.

Finally, administrators can insert false or misleading data into a DBMS in order to redirect or thwart information confidentiality attacks. This is a concept known as *noise and perturbation*. You must be extremely careful when using this technique to ensure that noise inserted into the database does not affect business operations.

Open Database Connectivity

Open Database Connectivity (ODBC) is a database feature that allows applications to communicate with different types of databases without having to be directly programmed for interaction with each type. ODBC acts as a proxy between applications and backend database drivers, giving application programmers greater freedom in creating solutions without having to worry about the backend database system. Figure 20.11 illustrates the relationship between ODBC and a backend database system.

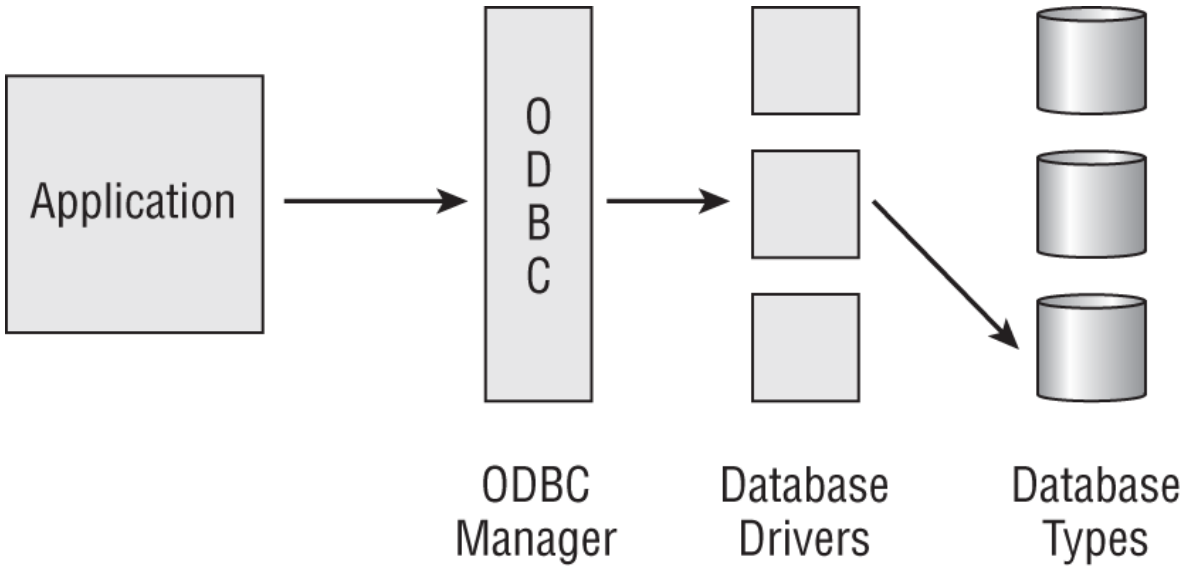

FIGURE 20.11 ODBC as the interface between applications and a backend database system

NoSQL

As database technology evolves, many organizations are turning away from the relational model for cases where they require increased speed or their data does not neatly fit into tabular form. NoSQL databases are a class of databases that use models other than the relational model to store data.

There are many different types of NoSQL database. As you prepare for the CISSP exam, you should be familiar with these common types:

- *Key-value stores* are perhaps the simplest possible form of database. They store information in key-value pairs, where the key is essentially an index used to uniquely identify a record, which consists of a data value. Key-value stores are useful for high-speed applications and very large datasets where the rigid structure of a relational model would require significant, and perhaps unnecessary, overhead.
- *Graph databases* store data in graph format, using nodes to represent objects and edges to represent relationships. They are useful for representing any type of network, such as social networks, geographic locations, and other datasets that lend themselves to graph representations.

- *Document stores* are similar to key-value stores in that they store information using keys, but the type of information they store is typically more complex than that in a key-value store and is in the form of a document. Common document types used in document stores include XML and JSON.

The security models used by NoSQL databases may differ significantly from relational databases. Security professionals in organizations that use this technology should familiarize themselves with the security features of the solutions they use and consult with database teams in the design of appropriate security controls.

Storage Threats

Database management systems have helped harness the power of data and grant some control over who can access it and the actions they can perform on it. However, security professionals must keep in mind that DBMS security covers access to information through only the traditional "front-door" channels. Data is also processed through a computer's storage resources—both memory and physical media. Precautions must be in place to ensure that these basic resources are protected against security vulnerabilities as well. After all, you would never incur a lot of time and expense to secure the front door of your home and then leave the backdoor wide open, would you?

Chapter 9, "Security Vulnerabilities, Threats, and Countermeasures," included a comprehensive look at different types of storage. Let's take a look at two main threats posed against data storage systems. First, the threat of illegitimate access to storage resources exists no matter what type of storage is in use. If administrators do not implement adequate file system access controls, an intruder might stumble across sensitive data simply by browsing the file system. In more sensitive environments, administrators should also protect against attacks that involve bypassing operating system controls and directly accessing the physical storage media to retrieve data. This is best accomplished through the use of an encrypted file

system, which is accessible only through the primary operating system. Furthermore, systems that operate in a multilevel security environment should provide adequate controls to ensure that shared memory and storage resources are set up with appropriate controls so that data from one classification level is not readable at a lower classification level.

Errors in storage access controls become particularly dangerous in cloud computing environments, where a single misconfiguration can publicly expose sensitive information on the web. Organizations leveraging cloud storage systems, such as Amazon's Simple Storage Service (S3), should take particular care to set strong default security settings that restrict public access and then carefully monitor any changes to that policy that allow public access.

Covert channel attacks pose the second primary threat against data storage resources. Covert storage channels allow the transmission of sensitive data between classification levels through the direct or indirect manipulation of shared storage media. This may be as simple as writing sensitive data to an inadvertently shared portion of memory or physical storage. More complex covert storage channels might be used to manipulate the amount of free space available on a disk or the size of a file to covertly convey information between security levels. For more information on covert channel analysis, see Chapter 8.

Understanding Knowledge-Based Systems

Since the advent of computing, engineers and scientists have worked toward developing systems capable of performing routine actions that would bore a human and consume a significant amount of time. The majority of the achievements in this area have focused on relieving the burden of computationally intensive tasks. However, researchers have also made giant

strides toward developing systems that have an “artificial intelligence” that can simulate (to some extent) the purely human power of reasoning.

The following sections examine three types of knowledge-based artificial intelligence (AI) systems: expert systems, machine learning, and neural networks. We'll also take a look at their potential applications to computer security problems.

Expert Systems

Expert systems seek to embody the accumulated knowledge of experts on a particular subject and apply it in a consistent fashion to future decisions. Several studies have shown that expert systems, when properly developed and implemented, often make better decisions than some of their human counterparts when faced with routine decisions.

Every expert system has two core components: the knowledge base and the inference engine.

The knowledge base contains the rules known by an expert system. The knowledge base seeks to codify the knowledge of human experts in a series of “if/then” statements. Let's consider a simple expert system designed to help homeowners decide whether they should evacuate an area when a hurricane threatens. The knowledge base might contain the following statements (these statements are for example purposes only):

- If the hurricane is a Category 4 storm or higher, then flood waters normally reach a height of 20 feet above sea level.
- If the hurricane has winds in excess of 120 miles per hour (mph), then wood-frame structures will be destroyed.
- If it is late in the hurricane season, then hurricanes tend to get stronger as they approach the coast.

In an actual expert system, the knowledge base would contain hundreds or thousands of assertions such as those just listed.

The second major component of an expert system—the inference engine—analyzes information in the knowledge base to arrive at the appropriate decision. The expert system user employs some sort of user interface to provide the inference engine with details about the current situation, and the inference engine uses a combination of logical reasoning and fuzzy logic techniques to draw a conclusion based on past experience. Continuing with the hurricane example, a user might inform the expert system that a Category 4 hurricane is approaching the coast with wind speeds averaging 140 mph. The inference engine would then analyze information in the knowledge base and make an evacuation recommendation based on that past knowledge.

Expert systems are not infallible—they're only as good as the data in the knowledge base and the decision-making algorithms implemented in the inference engine. However, they have one major advantage in stressful situations—their decisions do not involve judgment clouded by emotion. Expert systems can play an important role in analyzing emergency events, stock trading, and other scenarios in which emotional investment sometimes gets in the way of a logical decision. For this reason, many lending institutions now use expert systems to make credit decisions instead of relying on loan officers who might say to themselves, “Well, Jim hasn't paid his bills on time, but he seems like a perfectly nice guy.”

Machine Learning

Machine learning techniques use analytic capabilities to develop knowledge from datasets without the direct application of human insight. The core approach of machine learning is to allow the computer to analyze and learn directly from data, developing and updating models of activity.

Machine learning techniques fall into two major categories:

- *Supervised learning* techniques use labeled data for training. The analyst creating a machine learning model provides a dataset along with the correct answers and allows the algorithm to develop a model that may then be applied to

future cases. For example, if an analyst would like to develop a model of malicious system logins, the analyst would provide a dataset containing information about logins to the system over a period of time and indicate which were malicious. The algorithm would use this information to develop a model of malicious logins.

- *Unsupervised learning* techniques use unlabeled data for training. The dataset provided to the algorithm does not contain the "correct" answers; instead, the algorithm is asked to develop a model independently. In the case of logins, the algorithm might be asked to identify groups of similar logins. An analyst could then look at the groups developed by the algorithm and attempt to identify groups that may be malicious.

Neural Networks

In neural networks, chains of computational units are used in an attempt to imitate the biological reasoning process of the human mind. In an expert system, a series of rules is stored in a knowledge base, whereas in a neural network, a long chain of computational decisions that feed into each other and eventually sum to produce the desired output is set up. Neural networks are a subset of machine learning techniques and are also commonly referred to as *deep learning*.

Keep in mind that no neural network designed to date comes close to having the reasoning power of the human mind. Nevertheless, neural networks show great potential to advance the AI field beyond its current state. Benefits of neural networks include linearity, input-output mapping, and adaptivity. These benefits are evident in the implementations of neural networks for voice recognition, face recognition, weather prediction, and the exploration of models of thinking and consciousness.

Typical neural networks involve many layers of summation, each of which requires weighting information to reflect the relative importance of the calculation in the overall decision-making process. The weights must be custom-tailored for each type of

decision the neural network is expected to make. This is accomplished through the use of a training period during which the network is provided with inputs for which the proper decision is known. The algorithm then works backward from these decisions to determine the proper weights for each node in the computational chain. This activity is performed using what is known as the *Delta rule*. Through the use of the Delta rule, neural networks are able to learn from experience.

Knowledge-based analytic techniques have great applications in the field of computer security. One of the major advantages offered by these systems is their capability to rapidly make consistent decisions. One of the major problems in computer security is the inability of system administrators to consistently and thoroughly analyze massive amounts of log and audit trail data to look for anomalies. It seems like a match made in heaven.

Summary

Data is the most valuable resource many organizations possess. Therefore, it's critical that information security practitioners understand the necessity of safeguarding the data itself and the systems and applications that assist in the processing of that data. Protections against malicious code, database vulnerabilities, and system/application development flaws must be implemented in every technology-aware organization.

By this point, you no doubt recognize the importance of placing adequate access controls and audit trails on these valuable information resources. Database security is a rapidly growing field; if databases play a major role in your security duties, take the time to sit down with database administrators, courses, and textbooks and learn the underlying theory. It's a valuable investment.

Finally, various controls can be put into place during the system and application development process to ensure that the end product of these processes is compatible with operation in a secure environment. Such controls include process isolation, hardware segmentation, abstraction, and contractual

arrangements such as service-level agreements (SLAs). Security should always be introduced in the early planning phases of any development project and continually monitored throughout the design, development, deployment, and maintenance phases of production.

Study Essentials

Explain the basic architecture of a relational database management system (RDBMS). Know the structure of relational databases. Be able to explain the function of tables (relations), rows (records/tuples), and columns (fields/attributes). Know how relationships are defined between tables and the roles of various types of keys. Describe the database security threats posed by aggregation and inference.

Explain how expert systems, machine learning, and neural networks function. Expert systems consist of two core components: a knowledge base that contains a series of "if/then" rules and an inference engine that uses that information to draw conclusions about other data. Machine learning techniques attempt to algorithmically discover knowledge from datasets. Neural networks simulate the functioning of the human mind to a limited extent by arranging a series of layered calculations to solve problems. Neural networks require extensive training on a particular problem before they are able to offer solutions.

Understand the models of systems development. Know that the waterfall model describes a sequential development process that results in the development of a finished product. Developers may step back only one phase in the process if errors are discovered. The spiral model uses several iterations of the waterfall model to produce a number of fully specified and tested prototypes. Agile development models place an emphasis on the needs of the customer and quickly developing new functionality that meets those needs in an iterative fashion.

Explain the Scrum approach to Agile software development. Scrum is an organized approach to

implementing the Agile philosophy. It relies on daily scrum meetings to organize and review work. Development focuses on short sprints of activity that deliver finished products. Integrated product teams (IPTs) are an early effort at this approach that was used by the U.S. Department of Defense.

Describe software development maturity models. Know that maturity models help software organizations improve the maturity and quality of their software processes by implementing an evolutionary path from ad hoc, chaotic processes to mature, disciplined software processes. Be able to describe the SW-CMM, IDEAL, and SAMM models.

Understand the importance of change and configuration management. Know the three basic components of the change management process—request control, change control, and release control—and how they contribute to security. Explain how software configuration management controls the versions of software used in an organization. Understand how the auditing and logging of changes mitigates risk to the organization.

Understand the importance of testing. Software testing should be designed as part of the development process. Testing should be used as a management tool to improve the design, development, and production processes.

Explain the role of DevOps and DevSecOps in the modern enterprise. DevOps approaches seek to integrate software development and operations activities by embracing automation and collaboration between teams. DevSecOps approaches expand on the DevOps model by introducing security operations activities into the integrated model. Continuous integration and delivery (CI/CD) techniques automate the DevOps and DevSecOps pipelines.

Know the role of different coding tools in software development ecosystems. Developers write code in different programming languages, which is then either compiled into machine language or executed through an interpreter. Developers may make use of software development tool sets and integrated development environments to facilitate the code writing process.

Software libraries create shared and reusable code, whereas code repositories provide a management platform for the software development process.

Explain the impact of acquired software on the organization. Organizations may purchase commercial off-the-shelf (COTS) software to meet their requirements, and they may also rely on free open-source software (OSS). All of this software expands the potential attack surface and requires security review and testing.

Written Lab

1. What is the main purpose of a primary key in a database table?
2. What is polyinstantiation?
3. Explain the difference between supervised and unsupervised machine learning.

Review Questions

1. Christine is helping her organization implement a DevOps approach to deploying code. Which one of the following is *not* a component of the DevOps model?

   A. Information security

   B. Software development

   C. Quality assurance

   D. IT operations

2. Bob is developing a software application and has a field where users may enter a date. He wants to ensure that the values provided by the users are accurate dates to prevent security issues. What technique should Bob use?

   A. Polyinstantiation

   B. Input validation