environment must be maintained even though it usually is not used for regular production activities until after a disaster takes place that triggers the relocation of services to the redundant site. But "expensive" is relative here. If a company would lose a million dollars if it were out of business for just a few hours, the loss potential would override the cost of this option. Many organizations are subjected to regulations that dictate they must have redundant sites in place, so expense is not a matter of choice in these situations.

EXAM TIP A *hot* site is a subscription service. A *redundant* site, in contrast, is a site owned and maintained by the organization, meaning the organization does not pay anyone else for the site. A redundant site might be "hot" in nature, meaning it is ready for production quickly. However, the CISSP exam differentiates between a hot site (a subscription service) and a redundant site (owned by the organization).

Another type of facility-backup option is a *rolling hot site*, or mobile hot site, where the back of a large truck or a trailer is turned into a data processing or working area. This is a portable, self-contained data facility. The trailer has the necessary power, telecommunications, and systems to do some or all of the processing right away. The trailer can be brought to the organization's parking lot or another location. Obviously, the trailer has to be driven over to the new site, the data has to be retrieved, and the necessary personnel have to be put into place.

Another, similar solution is a prefabricated building that can be easily and quickly put together. Military organizations and large insurance companies typically have rolling hot sites or trucks preloaded with equipment because they often need the flexibility to quickly relocate some or all of their processing facilities to different locations around the world depending on where the need arises.

It is best if an organization is aware of all available options for hardware and facility backups to ensure it makes the best decision for its specific business and critical needs.

Multiple Processing Sites

Another option for organizations is to have *multiple processing sites*. An organization may have ten different facilities throughout the world, which are connected with specific technologies that could move all data processing from one facility to another in a matter of seconds when an interruption is detected. This technology can be implemented within the organization or from one facility to a third-party facility. Certain service providers provide this type of functionality to their customers. So if an organization's data processing is interrupted, all or some of the processing can be moved to the service provider's servers.

Availability

We close this section on recovery strategies by considering the nondisasters to which we referred earlier. These are the incidents that may not require evacuation of personnel or facility repairs but that can still have a significant detrimental effect on the ability of the organization to execute its mission. We want our systems and services to be available all the time, no matter what. However, we all realize this is just not possible. *Availability* can be defined as the portion of the time that a system is operational and able to fulfill its intended purpose. But how can we ensure the availability of the systems and services on which our organizations depend?

High Availability

High availability (HA) is a combination of technologies and processes that work together to ensure that some specific thing is up and running most of the time. The specific thing can be a database, a network, an application, a power supply, and so on. Service providers have *service level agreements (SLAs)* with their customers that outline the amount of uptime the service providers promise to provide. For example, a hosting company can promise to provide 99 percent uptime for Internet connectivity. This means the company is guaranteeing that at least 99 percent of the time, the Internet connection you purchase from it will be up and running. It also means that you can experience up to 3.65 days a year (or 7.2 hours per month) of downtime and it won't be a violation of the SLA. Increase that to 99.999 percent (referred to as "five nines") uptime and the allowable downtime drops to 5.26 seconds per year, but the price you pay for service goes through the roof.

NOTE HA is in the eye of the beholder. For some organizations or systems, an SLA of 90 percent ("one nine") uptime and its corresponding potential 36+ days of downtime a year is perfectly fine, particularly for organizations that are running on a tight budget. Other organizations require "nine nines" or 99.9999999 percent availability for mission-critical systems. You have to balance the cost of HA with the loss you're trying to mitigate.

Just because a service is available doesn't necessarily mean that it is operating acceptably. Suppose your company's high-speed e-commerce server gets infected with a bitcoin miner that drives CPU utilization close to 100 percent. Technically, the server is available and will probably be able to respond to customer requests. However, response times will likely be so lengthy that many of your customers will simply give up and go shop somewhere else. The service is available, but its quality is unacceptable.

Quality of Service

Quality of service (QoS) defines minimum acceptable performance characteristics of a particular service. For example, for the e-commerce server example, we could define parameters like response time, CPU utilization, or network bandwidth utilization, depending on how the service is being provided. SLAs may include one or more specifications for QoS, which allows service providers to differentiate classes of service that are prioritized for different clients. During a disaster, the available bandwidth on external links may be limited, so the affected organization could specify different QoS for its externally facing systems. For example, the e-commerce company in our example could determine the minimum data rate to keep its web presence available to customers and specify that as the minimum QoS rate at the expense of, say, its e-mail or Voice over Internet Protocol (VoIP) traffic.

To provide HA and meet stringent QoS requirements, the hosting company has to have a long list of technologies and processes that provide redundancy, fault tolerance, and failover capabilities. *Redundancy* is commonly built into the network at a routing protocol level. The routing protocols are configured such that if one link goes down or gets congested, traffic is automatically routed over a different network link. An organization can also ensure that it has redundant hardware available so that if a primary device goes down, the backup component can be swapped out and activated.

If a technology has a *failover* capability, this means that if there is a failure that cannot be handled through normal means, then processing is "switched over" to a working system. For example, two servers can be configured to send each other "heartbeat" signals every 30 seconds. If server A does not receive a heartbeat signal from server B after 40 seconds, then all processes are moved to server A so that there is no lag in operations. Also, when servers are *clustered*, an overarching piece of software monitors each server and carries out load balancing. If one server within the cluster goes down, the clustering software stops sending it data to process so that there are no delays in processing activities.

Fault Tolerance and System Resilience

Fault tolerance is the capability of a technology to continue to operate as expected even if something unexpected takes place (a fault). If a database experiences an unexpected glitch, it can roll back to a known-good state and continue functioning as though nothing bad happened. If a packet gets lost or corrupted during a TCP session, the TCP protocol will resend the packet so that system-to-system communication is not affected. If a disk within a RAID system gets corrupted, the system uses its parity data to rebuild the corrupted data so that operations are not affected.

Although the terms fault tolerance and resilience are often used synonymously, they mean subtly different things. Fault tolerance means that when a fault happens, there's a system in place (a backup or redundant one) to ensure services remain uninterrupted. *System resilience* means that the system continues to function, albeit in a degraded fashion, when a fault is encountered. Think of it as the difference between having a spare tire for your car and having run-flat tires. The spare tire provides fault tolerance in that it enables you to recover (fairly) quickly from a flat tire and be on your way. Run-flat tires allow you to continue to drive your car (albeit slower) if you run over a nail on the road. A resilient system is fault tolerant, but a fault tolerant one may not be resilient.

High Availability in Disaster Recovery

Redundancy, fault tolerance, resilience, and failover capabilities increase the reliability of a system or network, where *reliability* is the probability that a system performs the necessary function for a specified period under defined conditions. High reliability allows for high availability, which is a measure of its readiness. If the probability of a system performing as expected under defined conditions is low, then the availability for this system cannot be high. For a system to have the characteristic of high availability,

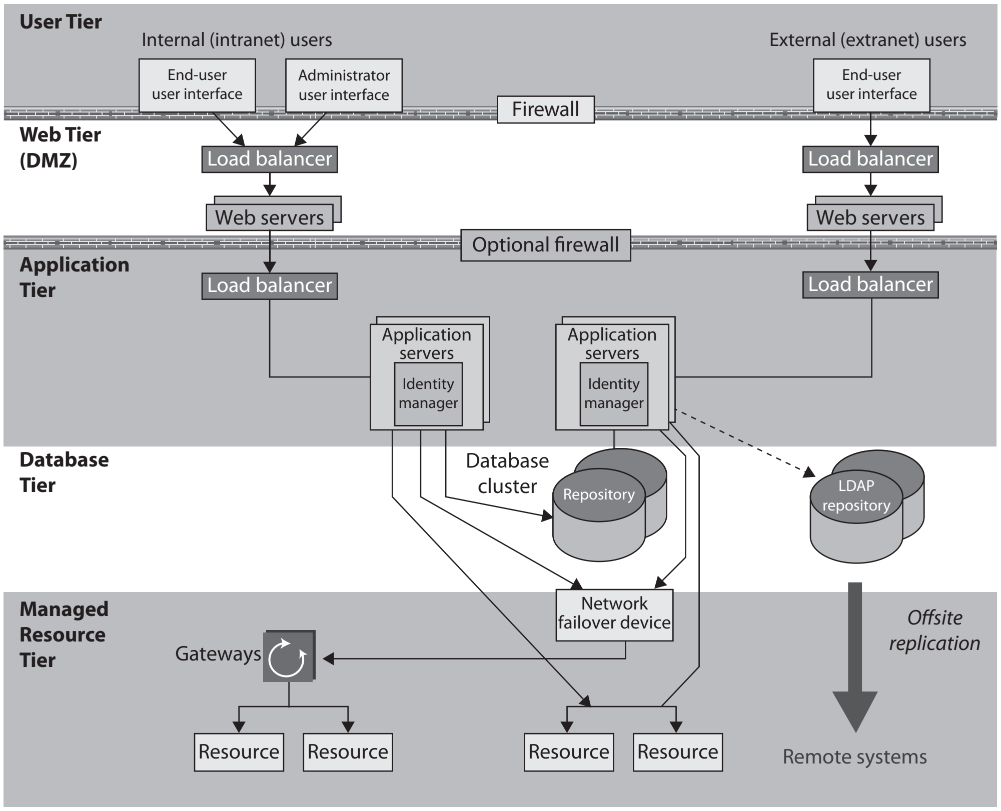


Figure 23-6 High-availability technologies

then high reliability must be in place. Figure 23-6 illustrates where load balancing, clustering, failover devices, and replication commonly take place in a network architecture.

Remember that data restoration (RPO) requirements can be different from processing restoration (RTO) requirements. Data can be restored through backup tapes, electronic vaulting, or synchronous or asynchronous replication. Processing capabilities can be restored through clustering, load balancing, redundancy, and failover technologies. If the results of the BCP team's BIA indicate that the RPO value is two days, then the organization can use tape backups. If the RPO value is one minute, then synchronous replication needs to be in place. If the BIA indicates that the RTO value is three days, then redundant hardware can be used. If the RTO value is one minute, then clustering and load balancing should be used.

HA and disaster recovery are related concepts. HA technologies and processes are commonly put into place so that if a disaster does take place, either the critical functions are likelier to remain available or the delay of getting them back online and running is low.

Many IT and security professionals usually think of HA only in technology terms, but remember that there are many things that an organization needs to have available to keep functioning. Availability of each of the following items must be thought through and planned:

- Facility (cold, warm, hot, redundant, rolling, reciprocal sites)
- Infrastructure (redundancy, fault tolerance)
- Storage (SAN, cloud)
- Server (clustering, load balancing)
- Data (backups, online replication)
- Business processes
- People

NOTE Virtualization and cloud computing are covered in Chapter 7. We will not go over those technologies again in this chapter, but know that the use of these technologies has drastically increased in the realm of business continuity and disaster recovery planning solutions.

Disaster Recovery Processes

Recovering from a disaster begins way before the event occurs. It starts by anticipating threats and developing goals that support the organization's continuity of operations. If you do not have established goals, how do you know when you are done and whether your efforts were actually successful? Goals are established so everyone knows the ultimate objectives. Establishing goals is important for any task, but especially for business continuity and disaster recovery plans. The definition of the goals helps direct the proper allocation of resources and tasks, supports the development of necessary strategies, and assists in financial justification of the plans and program overall. Once the goals are set, they provide a guide to the development of the actual plans themselves. Anyone who has been involved in large projects that entail many small, complex details knows that at times it is easy to get off track and not actually accomplish the major goals of the project. Goals are established to keep everyone on track and to ensure that the efforts pay off in the end.

Great—we have established that goals are important. But the goal could be, "Keep the company in business if an earthquake hits." That's a good goal, but it is not overly useful without more clarity and direction. To be useful, a goal must contain certain key information, such as the following:

- **Responsibility** Each individual involved with recovery and continuity should have their responsibilities spelled out in writing to ensure a clear understanding in a chaotic situation. Each task should be assigned to the individual most logically situated to handle it. These individuals must know what is expected of them, which is done through training, exercises, communication, and documentation. So, for example, instead of just running out of the building screaming, an individual must know that he is responsible for shutting down the servers before he can run out of the building screaming.

- **Authority** In times of crisis, it is important to know who is in charge. Teamwork is important in these situations, and almost every team does much better with an established and trusted leader. Such leaders must know that they are expected to step up to the plate in a time of crisis and understand what type of direction they should provide to the rest of the employees. Everyone else must recognize the authority of these leaders and respond accordingly. Clear-cut authority will aid in reducing confusion and increasing cooperation.
- **Priorities** It is extremely important to know what is critical versus what is merely nice to have. Different departments provide different functionality for an organization. The critical departments must be singled out from the departments that provide functionality that the organization can live without for a week or two. It is necessary to know which department must come online first, which second, and so on. That way, the efforts are made in the most useful, effective, and focused manner. Along with the priorities of departments, the priorities of systems, information, and programs must be established. It may be necessary to ensure that the database is up and running before working to bring the web servers online. The general priorities must be set by management with the help of the different departments and IT staff.
- **Implementation and testing** It is great to write down very profound ideas and develop plans, but unless they are actually carried out and tested, they may not add up to a hill of beans. Once a disaster recovery plan is developed, it actually has to be put into action. It needs to be documented and stored in places that are easily accessible in times of crisis. The people who are assigned specific tasks need to be taught and informed how to fulfill those tasks, and dry runs must be done to walk people through different situations. The exercises should take place at least once a year, and the entire program should be continually updated and improved.

NOTE We address various types of tests, such as walkthrough, tabletop, simulation, parallel, and full interruption, later in this chapter.

According to the U.S. Federal Emergency Management Agency (FEMA), 90 percent of small businesses that experience a disaster and are unable to restore operations within five days will fail within the following year. Not being able to bounce back quickly or effectively by setting up shop somewhere else can make a company lose business and, more importantly, its reputation. In such a competitive world, customers have a lot of options. If one company is not prepared to bounce back after a disruption or disaster, customers may go to another vendor and stay there.

The biggest effect of an incident, especially one that is poorly managed or that was preventable, is on an organization's reputation or brand. This can result in a considerable and even irreparable loss of trust by customers and clients. On the other hand, handling an incident well, or preventing great damage through smart, preemptive measures, can enhance the reputation of, or trust in, an organization.

The *disaster recovery plan (DRP)* should address in detail all of the topics we have covered so far. The actual format of the DRP will depend on the environment, the goals of the plan, priorities, and identified threats. After each of those items is examined and documented, the topics of the plan can be divided into the necessary categories.

Response

The first question the DRP should answer is, "What constitutes a disaster that would trigger this plan?" Every leader within an organization (and, ideally, everyone else too) should know the answer. Otherwise, precious time is lost notifying people who should've self-activated as soon as the incident occurred, a delay that could cost lives or assets. Examples of clear-cut disasters that would trigger a response are loss of power exceeding ten minutes, flooding in the facility, or terrorist attack against or near the site.

Every DRP is different, but most follow a familiar sequence of events:

1. Declaration of disaster
2. Activation of the DR team
3. Internal communications (ongoing from here on out)
4. Protection of human safety (e.g., evacuation)
5. Damage assessment
6. Execution of appropriate system-specific DRPs (each system and network should have its own DRP)
7. Recovery of mission-critical business processes/functions
8. Recovery of all other business processes/functions

Personnel

The DRP needs to define several different teams that should be properly trained and available if a disaster hits. Which types of teams an organization needs depends upon the organization. The following are some examples of teams that an organization may need to construct:

- Damage assessment team
- Recovery team
- Relocation team
- Restoration team
- Salvage team
- Security team

The DR coordinator should have an understanding of the needs of the organization and the types of teams that need to be developed and trained. Employees should be assigned to the specific teams based on their knowledge and skill set. Each team needs to have a designated leader, who will direct the members and their activities. These team leaders will be responsible not only for ensuring that their team's objectives are met but also for communicating with each other to make sure each team is working in parallel phases.

The purpose of the *recovery team* should be to get whatever systems are still operable back up and running as quickly as possible to reduce business disruptions. Think of them as the medics whose job is to stabilize casualties until they can be transported to the hospital. In this case, of course, there is no hospital for information systems, but there may be a recovery site. Getting equipment and people there in an orderly fashion should be the job of the *relocation team*. The *restoration team* should be responsible for getting the alternate site into a working and functioning environment, and the *salvage team* should be responsible for starting the recovery of the original site. Both teams must know how to do many tasks, such as install operating systems, configure workstations and servers, string wire and cabling, set up the network and configure networking services, and install equipment and applications. Both teams must also know how to restore data from backup facilities and how to do so in a secure manner, one that ensures the availability, integrity, and confidentiality of the system and data.

The DRP must outline the specific teams, their responsibilities, and notification procedures. The plan must indicate the methods that should be used to contact team leaders during business hours and after business hours.

Communications

The purpose of the emergency communications plan that is part of the overall DRP is to ensure that everyone knows what to do at all times and that the DR team remains synchronized and coordinated. This all starts with the DR plan itself. As stated previously, copies of the DRP need to be kept in one or more locations other than the primary site, so that if the primary site is destroyed or negatively affected, the plan is still available to the teams. It is also critical that different formats of the plan be available to the teams, including both electronic and paper versions. An electronic version of the plan is not very useful if you don't have any electricity to run a computer.

In addition to having copies of the recovery documents located at their offices and homes, key individuals should have easily accessible versions of critical procedures and call tree information. One simple way to accomplish the latter is to publish a call tree on cards that can be affixed to personnel badges or kept in a wallet. In an emergency situation, valuable minutes are better spent responding to an incident than looking for a document or having to wait for a laptop to power up. Of course, the call tree is only as effective as it is accurate and up to date, so verifying it periodically is imperative.

One limitation of call trees is that they are point to point, which means they're typically good for getting the word out, but not so much for coordinating activities. Group text messages work better, but only in the context of fairly small and static groups. Many organizations have group chat solutions, but if those rely on the organization's servers, they may be unavailable during a disaster. It is a good idea, then, to establish a communications platform that is completely independent of the organizational infrastructure. Solutions like Slack and Mattermost offer a free service that is typically sufficient to keep most organizations connected in emergencies. The catch, of course, is that everyone needs to have the appropriate client installed on their personal devices and know when and how to connect. Training and exercises are the keys to successful execution of any plan, and the communications plan is no exception.

NOTE An organization may need to solidify communications channels and relationships with government officials and emergency response groups. The goal of this activity is to solidify proper protocol in case of a city- or region-wide disaster. During the BIA phase, the DR team should contact local authorities to elicit information about the risks of its geographical location and how to access emergency zones. If the organization has to perform DR, it may need to contact many of these emergency response groups.

PACE Communications Plans

The U.S. armed forces routinely develop Primary, Alternate, Contingency, and Emergency (PACE) communications plans. The PACE plan outlines the different capabilities that exist and aligns them into these four categories based on their ability to meet defined information exchange requirements. Each category is defined here:

- **Primary** The normal or expected capability that is used to achieve the objective.
- **Alternate** A fully satisfactory capability that can be used to achieve the objective with minimal impact to the operation or exercise. This capability is used when the Primary capability is unavailable.
- **Contingency** A workable capability that can be used to achieve the objective. This capability may not be as fast or easy as the Primary or Alternate but is capable of achieving the objective with an acceptable amount of time and effort. This capability is used when the Primary and the Alternate capabilities are unavailable.
- **Emergency** This is the last-resort capability and typically may involve significantly more time and effort than any of the other capabilities. This capability should be used only when the Primary, Alternate, and Contingency capabilities are unavailable.

The PACE plan includes redundant communications capabilities and specifies the order in which the organization will employ the capabilities when communication outages occur.

Assessment

A role, or a team, needs to be created to carry out a *damage assessment* once a disaster has taken place. The assessment procedures should be properly documented in the DRP and include the following steps:

- Determine the cause of the disaster.
- Determine the potential for further damage.
- Identify the affected business functions and areas.
- Identify the level of functionality for the critical resources.
- Identify the resources that must be replaced immediately.
- Estimate how long it will take to bring critical functions back online.

After the damage assessment team collects and assesses this information, the DR coordinator identifies which teams need to be called to action and which system-specific DRPs need to be executed (and in what order). The DRP should specify activation criteria for the different teams and system-specific DRPs. After the damage assessment, if one or more of the situations outlined in the criteria have taken place, then the DR team is moved into restoration mode.

Different organizations have different activation criteria because business drivers and critical functions vary from organization to organization. The criteria may comprise some or all of the following elements:

- Danger to human life
- Danger to state or national security
- Damage to facility
- Damage to critical systems
- Estimated value of downtime that will be experienced

Restoration

Once the damage assessment is completed, various teams are activated, which signals the organization's entry into the *restoration phase*. Each team has its own tasks—for example, the facilities team prepares the offsite facility (if needed), the network team rebuilds the network and systems, and the relocation team starts organizing the staff to move into a new facility.

The restoration process needs to be well organized to get the organization up and running as soon as possible. This is much easier to state in a book than to carry out in reality, which is why written procedures are critical. The critical functions and their resources would already have been identified during the BIA, as discussed earlier in this chapter (with a simplistic example provided in Table 23-1). These are the functions that the teams need to work together on restoring first.

Many organizations create templates during the DR plan development stage. These templates are used by the different teams to step them through the necessary phases and to document their findings. For example, if one step could not be completed until new systems were purchased, this should be indicated on the template. If a step is partially completed, this should be documented so the team does not forget to go back and finish that step when the necessary part arrives. These templates keep the teams on task and also quickly tell the team leaders about the progress, obstacles, and potential recovery time.

NOTE Examples of possible templates can be found in NIST Special Publication 800-34, Revision 1, *Contingency Planning Guide for Federal Information Systems*, which is available online at https://csrc.nist.gov/publications/detail/sp/800-34/rev-1/final.

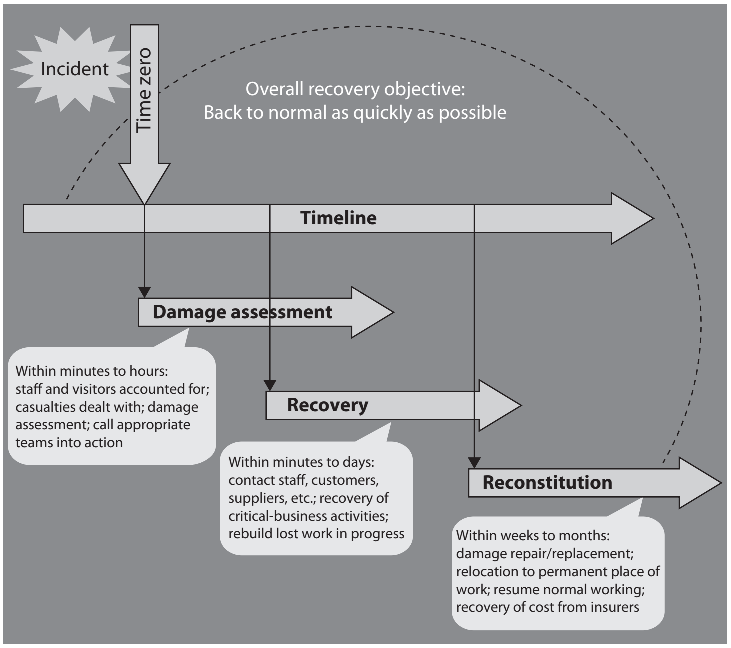
An organization is not out of an emergency state until it is back in operation at the original primary site or at a new site that was constructed to replace the primary original one, because the organization is always vulnerable while operating in a backup facility. Many logistical issues need to be considered as to when an organization should return from the alternate site to the primary one. The following lists a few of these issues:

- Ensuring the safety of employees
- Ensuring an adequate environment is provided (power, facility infrastructure, water, HVAC)
- Ensuring that the necessary equipment and supplies are present and in working order
- Ensuring proper communications and connectivity methods are working
- Properly testing the new environment

Once the coordinator, management, and salvage team sign off on the readiness of the primary site, the salvage team should carry out the following steps:

- Back up data from the alternate site and restore it within the primary site.
- Carefully terminate contingency operations.
- Securely transport equipment and personnel to the primary site.

The least critical functions should be moved back first, so if there are issues in network configurations or connectivity, or important steps were not carried out, the critical operations of the organization are not negatively affected. Why go through the trouble of moving the most critical systems and operations to a safe and stable alternate site, only to return them to a main site that is untested? Let the less critical departments act as the canary in the coal mine. If they survive, then move the more critical components of the organization to the main site.


Incident
Time zero
Overall recovery objective:
Back to normal as quickly as possible
Timeline
Damage assessment
Within minutes to hours: staff and visitors accounted for; casualties dealt with; damage assessment; call appropriate teams into action
Recovery
Within minutes to days: contact staff, customers, suppliers, etc.; recovery of critical-business activities; rebuild lost work in progress
Reconstitution
Within weeks to months: damage repair/replacement; relocation to permanent place of work; resume normal working; recovery of cost from insurers

Training and Awareness

Training your DR team on the execution of a DRP is critical for at least three reasons. First, it allows you to validate that the plan will actually work. If your DR team is doing a walkthrough exercise in response to a fictitious scenario, you'll find out very quickly whether the plan would work or not. If it doesn't work in a training event when the stress level and stakes are low, then there is no chance it would work in a real emergency.

Another reason to train is to ensure that everyone knows what they're supposed to do, when, where, and how. Disasters are stressful, messy affairs and key people may not be thinking clearly. It is important for them to have a familiar routine to fall back on. In a perfect world, you would train often enough for your team to develop "muscle memory" that allows them to automatically do the right things without even thinking.

Lastly, training can help establish that you are exercising due care. This could keep you out of legal trouble in the aftermath of a disaster, particularly if people end up getting hurt. A good plan and evidence of a trained workforce can go a long way to reduce liability if regulators or other investigators come knocking. As always, consult your attorneys to ensure you are meeting all applicable legal and regulatory obligations.

When thinking of training and "muscle memory," you should also consider everyone else in the organization that is not part of the DR team. You want all your staff to have an awareness of the major things they need to do to support DR. This is why many of us conduct fire drills in our facilities: to ensure everyone knows how to get out of the building and where to assemble if we ever face this particular kind of disaster. There are many types of DR awareness events you can run, but you should at least consider three types of responses that everyone should be aware of: evacuations (e.g., for fires or explosives), shelter-in-place (e.g., for tornadoes or active shooters), and remain-at-home (e.g., for overnight flooding).

Lessons Learned

As mentioned on the first page of this chapter, no battle plan ever survived first contact with the enemy. When you try to execute your DRP in a real disaster, you will find the need to disregard parts of it, make on-the-fly changes to others, and faithfully execute the rest. This is why you should incorporate lessons learned from any actual disasters and actual responses. The DR team should perform a "postmortem" on the response and ensure that necessary changes are made to plans, contracts, personnel, processes, and procedures.

Military organizations collect lessons learned in two steps. The first steps, called a *hotwash*, is a hasty one that happens right after the event is concluded (i.e., restoration is completed). The term comes from the military practice of dousing rifles with very hot water immediately after an engagement to quickly get the worst grit and debris off their weapons. The reason you want to conduct a hotwash right away is that memories will be freshest right after restoring the systems. The idea is not necessarily to figure out how to fix anything, but rather to quickly list as many things that went well or poorly as possible before participants start to forget them.

The second event at which lessons learned are collected in the military is much more deliberate. An after-action review (AAR) happens several days after completion of the DR and allows participants to think things through and start formulating possible ways to do better in the future. The AAR facilitator, ideally armed with the notes from the hotwash, presents each issue that was recorded (good or bad), a brief discussion of it, and then opens the floor for recommendations. Keep in mind that since you're dealing with things that went well or poorly, sometimes the group recommendation will be to "sustain" the issue or, in other words, keep doing things the same way in the future. More frequently, however, there are at least minor tweaks that can improve future performance.

Testing Disaster Recovery Plans

The disaster recovery plan should be tested regularly because environments continually change. Interestingly, many organizations are moving away from the concept of "testing," because a test naturally leads to a pass or fail score, and in the end, that type of score is not very productive. Instead, many organizations are adopting the concept of "exercises," which appear less stressful, better focused, and ultimately more productive to the participants. Each time the DRP is exercised or tested, improvements and efficiencies are generally uncovered, yielding better and better results over time. The responsibility of establishing periodic exercises and the maintenance of the plan should be assigned to a specific person or persons who will have overall ownership responsibilities for the disaster recovery initiatives within the organization.

The maintenance of the DRP should be incorporated into change management procedures. That way, any changes in the environment are reflected in the plan itself.

Tests and disaster recovery exercises should be performed at least once a year. An organization should have no real confidence in a developed plan until it has actually been tested. Exercises prepare personnel for what they may face and provide a controlled environment to learn the tasks expected of them. These exercises also point out issues to the planning team and management that may not have been previously thought about and addressed as part of the planning process. The exercises, in the end, demonstrate whether an organization can actually recover after a disaster.

The exercise should have a predetermined scenario that the organization may indeed be faced with one day. Specific parameters and a scope of the exercise must be worked out before sounding the alarms. The team of testers must agree upon what exactly is getting tested and how to properly determine success or failure. The team must agree upon the timing and duration of the exercise, who will participate in the exercise, who will receive which assignments, and what steps should be taken. Also, the team needs to determine whether hardware, software, personnel, procedures, and communications lines are going to be tested and whether it is all or a subset of these resources that will be included in the event. If the test will include moving some equipment to an alternate site, then transportation, extra equipment, and alternate site readiness must be addressed and assessed.

Most organizations cannot afford to have these exercises interrupt production or productivity, so the exercises may need to take place in sections or at specific times, which will require logistical planning. Written exercise plans should be developed that will test for specific weaknesses in the overall DRP. The first exercises should not include all employees, but rather a small representative sample of the organization. This allows both the planners and the participants to refine the plan. It also allows each part of the organization to learn its roles and responsibilities. Then, larger exercises can take place so overall operations will not be negatively affected.

The people conducting these exercises should expect to encounter problems and mistakes. After all, identifying potential problems and mistakes is why they are conducting the exercises in the first place. An organization would rather have employees make mistakes during an exercise so they can learn from them and perform their tasks more effectively during a real disaster.

NOTE After a disaster, telephone service may not be available. For communications purposes, alternatives should be in place, such as mobile phones or hand-held radios.

A few different types of exercises and tests can be used, each with its own pros and cons. The following sections explain the different types of assessment events.

Checklist Test

In this type of test, copies of the DRP are distributed to the different departments and functional areas for review. This enables each functional manager to review the plan and indicate if anything has been left out or if some approaches should be modified or deleted. This method ensures that nothing is taken for granted or omitted, as might be the case in a single-department review. Once the departments have reviewed their copies and made suggestions, the planning team then integrates those changes into the master plan.

NOTE The checklist test is also called the desk check test.

Structured Walkthrough Test

In this test, representatives from each department or functional area come together and go over the plan to ensure its accuracy. The group reviews the objectives of the plan; discusses the scope and assumptions of the plan; reviews the organization's reporting structure; and evaluates the testing, maintenance, and training requirements described. This gives the people responsible for making sure a disaster recovery happens effectively and efficiently an opportunity to review what has been decided upon and what is expected of them.

The group walks through different scenarios of the plan from beginning to end to make sure nothing was left out. This also raises the awareness of team members about the recovery procedures.

Tabletop Exercises

Tabletop exercises (TTXs) may or may not happen at a tabletop, but they do not involve a technical control infrastructure. TTXs can happen at an executive level (e.g., C-suite) or at a team level (e.g., SOC), or anywhere in between. The idea is usually to test procedures and ensure they actually do what they're intended to and that everyone knows their role in responding to a disaster. TTXs require relatively few resources apart from deliberate planning by qualified individuals and the undisturbed time and attention of the participants.

After determining the goals of the exercise and vetting them with the senior leadership of the organization, the planning team develops a scenario that touches on the important aspects of the response plan. The idea is normally not to cover every contingency, but to ensure the DR team is able to respond to the likeliest and/or most dangerous scenarios. As they develop the exercise, the planning team considers branches and sequels at every point in the scenario. A *branch* is a point in which the participants may choose one of multiple approaches to respond. If the branches are not carefully managed and controlled, the TTX could wander into uncharted and unproductive directions. Conversely, a *sequel* is a follow-on to a given action in the response. For instance, as part of the response, the strategic communications team may issue statements to the news media. A sequel to that could involve a media outlet challenging the statement, which in turn would require a response by the team. Like branches, sequels must be used carefully to keep the exercise on course. Senior leadership support and good scenario development are critical ingredients to attract and engage the right participants. Like any contest, a TTX is only as good as the folks who show up to play.

EXAM TIP Tabletop exercises are also called read-through exercises.

Simulation Test

This type of test takes a lot more planning and people. In this situation, all employees who participate in operational and support functions, or their representatives, come together to practice executing the disaster recovery plan based on a specific scenario. The scenario is used to test the reaction of each operational and support representative. Again, this is done to ensure specific steps were not left out and that certain threats were not overlooked. It raises the awareness of the people involved.

The exercise includes only those materials that will be available in an actual disaster, to portray a more realistic environment. The simulation test continues up to the point of actual relocation to an offsite facility and actual shipment of replacement equipment.

Parallel Test

In a parallel test, some systems are moved to the alternate site and processing takes place. The results are compared with the regular processing that is done at the original site. This ensures that the specific systems can actually perform adequately at the alternate offsite facility and points out any tweaking or reconfiguring that is necessary.

Full-Interruption Test

This type of test is the most intrusive to regular operations and business productivity. The original site is actually shut down, and processing takes place at the alternate site. The recovery team fulfills its obligations in preparing the systems and environment for the alternate site. All processing is done only on devices at the alternate offsite facility.

This is a full-blown exercise that takes a lot of planning and coordination, but it can reveal many holes in the plan that need to be fixed before an actual disaster hits. Full-interruption tests should be performed only after all other types of tests have been successful. They are the riskiest type and can impact the business in very serious and devastating ways if not managed properly; therefore, senior management approval needs to be obtained prior to performing full-interruption tests.

The type of organization and its goals will dictate what approach to the training exercise is most effective. Each organization may have a different approach and unique aspects. If detailed planning methods and processes are going to be taught, then specific training may be required rather than general training that provides an overview. Higher-quality training will result in an increase in employee interest and commitment.

During and after each type of test, a record of the significant events should be documented and reported to management so it is aware of all outcomes of the test.

Other Types of Training

Other types of training that employees need in addition to disaster recovery training include first aid and cardiac pulmonary resuscitation (CPR), how to properly use a fire extinguisher, evacuation routes and crowd control methods, emergency communications procedures, and how to properly shut down equipment in different types of disasters.

The more technical employees may need training on how to redistribute network resources and how to use different telecommunications lines if the main one goes down. They may need to know about redundant power supplies and be trained and tested on the procedures for moving critical systems from one power supply to the next.

Business Continuity

When a disaster strikes, ensuring that the organization is able to continue its operations requires more than simply restoring data from backups. Also necessary are the detailed procedures that outline the activities to keep the critical systems available and ensure that operations and processing are not interrupted. Business continuity planning defines what should take place during and after an incident. Actions that are required to take place for emergency response, continuity of operations, and dealing with major outages must be documented and readily available to the operations staff. There should be at least two instances of these documents: the original that is kept on-site and a copy that is at an offsite location.

BC plans should not be trusted until they have been tested. Organizations should carry out exercises to ensure that the staff fully understands their responsibilities and how to carry them out. We already covered the various types of exercises that can be used to test plans and staff earlier in this chapter when we discussed DR. Another issue to consider is how to keep these plans up to date. As our dynamic, networked environments change, so must our plans on how to rescue them when necessary.

Although in the security industry "contingency planning" and "business continuity planning (BCP)" are commonly used interchangeably, it is important that you understand the actual difference for the CISSP exam. BCP addresses how to keep the organization in business after a major disruption takes place. It is about the survivability of the organization and making sure that critical functions can still take place even after a disaster. Contingency plans address how to deal with small incidents that do not qualify as disasters, as in power outages, server failures, a down communication link to the Internet, or the corruption of software. Organizations must be ready to deal with both large and small issues that they may encounter.

EXAM TIP BCP is broad in scope and deals with survival of the organization. Contingency plans are narrow in scope and deal with specific issues.

As a security professional you will most likely not be in charge of BCP, but you should most certainly be an active participant in developing the BCP. You will also be involved in BC exercises and may even be a lead in those that focus on information systems. To effectively participate in BC planning and exercises, you should be familiar with the BCP life cycle, how to ensure continuous availability of critical information systems, and the particular requirements of the end-user environments. We look at these in the following sections.

BCP Life Cycle

Remember that most organizations aren't static, but change, often rapidly, as do the conditions under which they must operate. Thus, BCP should be considered a life cycle in order to deal with the constant and inevitable change that will affect it. Understanding and maintaining each step of the BCP life cycle is critical to ensuring that the BC plan remains useful to the organization. The BCP life cycle is outlined in Figure 23-7.

Note that this life cycle has two modes: normal management (shown in the top half of Figure 23-7) and incident management (shown in the bottom half). In the normal mode, the focus of the BC team is on ensuring preparedness. Obviously, we want to start

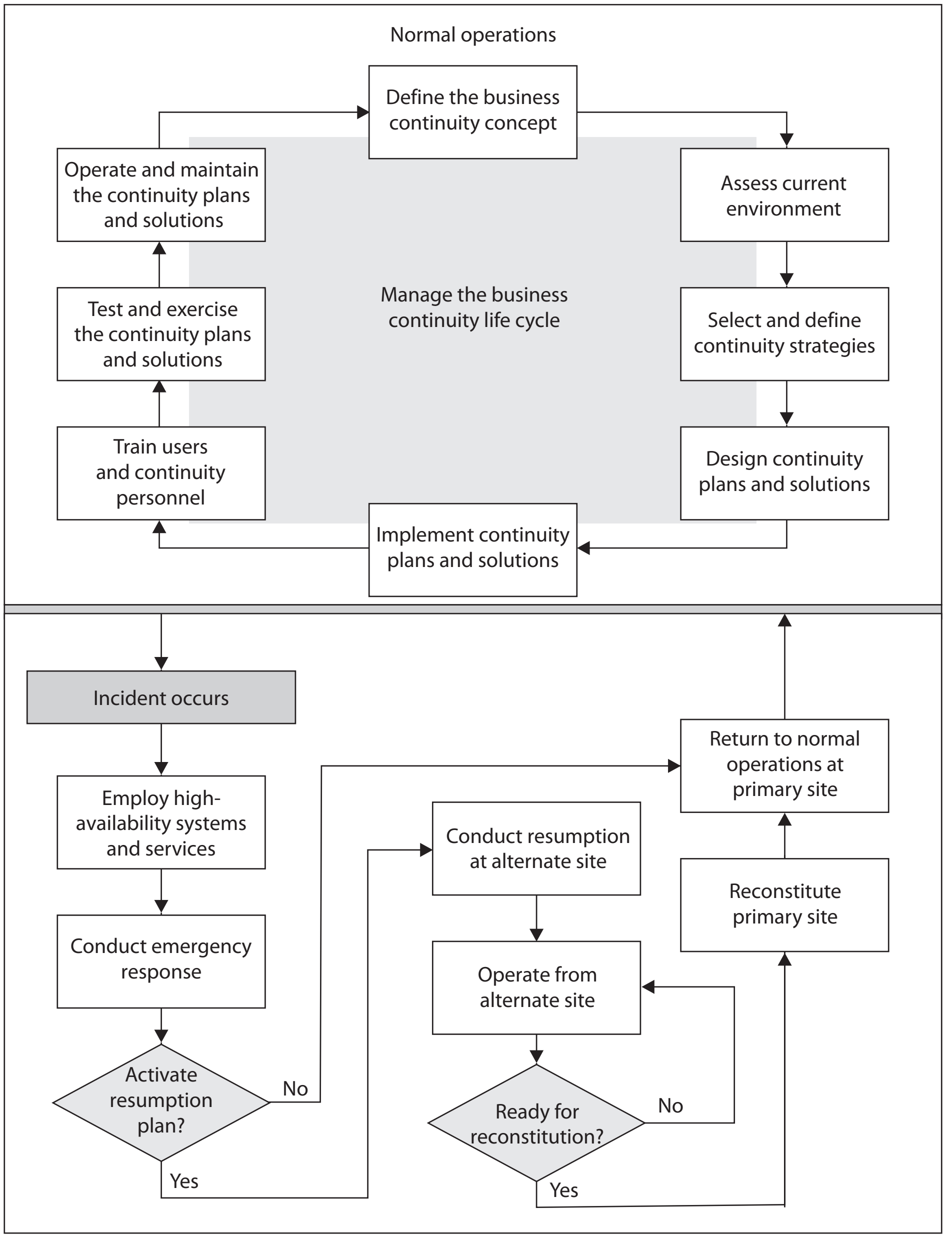

Normal operations
Define the business continuity concept
Assess current environment
Select and define continuity strategies
Design continuity plans and solutions
Implement continuity plans and solutions
Train users and continuity personnel
Test and exercise the continuity plans and solutions
Operate and maintain the continuity plans and solutions
Manage the business continuity life cycle
Incident occurs
Employ high-availability systems and services
Conduct emergency response
Activate resumption plan?
No
Yes
Conduct resumption at alternate site
Operate from alternate site
Ready for reconstitution?
No
Yes
Reconstitute primary site
Return to normal operations at primary site

Figure 23-7 BCP life cycle

with a clearly defined concept for what business continuity means for the organization. What are the critical business functions that must continue to operate regardless of what incident happens? What are the minimum levels of performance that are acceptable for these functions?

Once we define the BC concept, we can take a look at the current environment and consider the strategies that would allow continuity of operations under a variety of conditions. It is important to consider that, unlike DR planning, not every type of incident covered in BCP involves loss of IT capabilities. Many organizations suffered tremendously in 2020 because their BCP didn't account for a global pandemic in which many (or even all) staff members would have to work from home for extended periods of time. Information systems are certainly an important part of the continuity strategies, plans, and solutions, but the scope of the BCP is much broader than that of the DRP.

The BC plan is only useful if the organization in general, and the BC team in particular, knows how to execute the plan. This requires periodic training, tests, and exercises to ensure that both the plan and the staff are able to keep the business going no matter what comes their way. As we find gaps and opportunities for improvement, we get to redefine our BCP concept and start another run through the cycle. This continuous improvement is key to being able to switch into incident management mode (at the bottom of Figure 23-7) when needed and execute the BC plan (and, potentially, the DR plan) to keep the business going.

Information Systems Availability

Our main job as CISSPs in the BCP life cycle is to ensure the continuous availability of organizational information systems. To this end, we should ensure the BCP includes backup solutions for the following:

- Network and computer equipment
- Voice and data communications resources
- Human resources
- Transportation of equipment and personnel
- Environment issues (HVAC)
- Data and personnel security issues
- Supplies (paper, forms, cabling, and so on)
- Documentation

The BCP team must understand the organization's current technical environment. This means the planners have to know the intimate details of the network, communications technologies, computers, network equipment, and software requirements that are necessary to get the critical functions up and running. What is surprising to some people is that many organizations do not *totally* understand how their network is configured and how it actually works, because the network may have been established 10 to 15 years ago and has kept growing and changing under different administrators and personnel.

Outsourcing

Part of the planned response to a disaster may be to outsource some of the affected activities to another organization. Organizations do outsource activities—help-desk services, manufacturing, legal advice—all the time, so why not important functions affected by a disaster? Some companies specialize in disaster response and continuity planning and can act as expert consultants.

That is all well and good. However, be aware that your organization is still ultimately responsible for the continuity of a product or service that is outsourced. Clients and customers will expect the organization to ensure continuity of its products and services, either by itself or by having chosen the right outside vendors to provide the products and services. If outside vendors are brought in, the active participation of key in-house managers in their work is still essential. They still need to supervise the work of the outside vendors.

This same concern applies to normal, third-party suppliers of goods and services to the organization. Any BCP should take them into account as well. Note that the process for evaluating an outsourced company for BCP is like that for evaluating the organization itself. The organization must make sure that the outsourced company is financially viable and has its own solid BCP.

The organization can take the following steps to better ensure the continuity of its outsourcing:

- Make the ability of such companies to reliably assure continuity of products and services part of any work proposals.
- Make sure that business continuity planning is included in contracts with such companies, and that their responsibilities and levels of service are clearly spelled out.
- Draw up realistic and reasonable service levels that the outsourced firm will meet during an incident.
- If possible, have the outsourcing companies take part in BCP awareness programs, training, and testing.

The goal is to make the supply of goods and services from outsources as resilient as possible in the wake of a disaster.

New devices are added, new computers are added, new software packages are added, VoIP may have been integrated, and the DMZ may have been split up into three DMZs, with an extranet for the organization's partners. Maybe a company bought and merged with another company and network. Over ten years, a number of technology refreshes most likely have taken place, and the individuals who are maintaining the environment now likely are not the same people who built it ten years ago. Many IT departments experience extensive employee turnover every five years. And most organizational network schematics are notoriously out of date because everyone is busy with their current tasks (or will come up with new tasks just to get out of having to update the schematic).

So the BCP team has to make sure that if the networked environment is partially or totally destroyed, the recovery team has the knowledge and skill to properly rebuild it.

NOTE Many organizations use VoIP, which means that if the network goes down, network and voice capability are unavailable. The BCP team should address the possible need of redundant voice systems.

The BCP team needs to incorporate into the BCP several things that are commonly overlooked, such as hardware replacements, software products, documentation, environmental needs, and human resources.

Hardware Backups

The BCP needs to identify the equipment required to keep the critical functions up and running. This may include servers, user workstations, routers, switches, tape backup devices, and more. The needed inventory may seem simple enough, but as they say, the devil is in the details. If the recovery team is planning to use images to rebuild newly purchased servers and workstations because the original ones were destroyed, for example, will the images work on the new computers? Using images instead of building systems from scratch can be a time-saving task, unless the team finds out that the replacement equipment is a newer version and thus the images cannot be used. The BCP should plan for the recovery team to use the organization's current images, but also have a manual process of how to build each critical system from scratch with the necessary configurations.

The BCP also needs to be based on accurate estimates of how long it will take for new equipment to arrive. For example, if the organization has identified Dell as its equipment replacement supplier, how long will it take this vendor to send 20 servers and 30 workstations to the offsite facility? After a disaster hits, the organization could be in its offsite facility only to find that its equipment will take three weeks to be delivered. So, the SLA for the identified vendors needs to be investigated to make sure the organization is not further damaged by delays. Once the parameters of the SLA are understood, the BCP team must make a decision between depending upon the vendor and purchasing redundant systems and storing them as backups in case the primary equipment is destroyed.

As described earlier, when potential organizational risks are identified, it is better to take preventive steps to reduce the potential damage. After the calculation of the MTD values, the team will know how long the organization can operate without a specific device. This data should be used to make the decision on whether the organization should depend on the vendor's SLA or make readily available a hot-swappable redundant system. If the organization will lose $50,000 per hour if a particular server goes down, then the team should elect to implement redundant systems and technology.

If an organization is using any legacy computers and hardware and a disaster hits tomorrow, where would it find replacements for this legacy equipment? The BCP team should identify legacy devices and understand the risk the organization is facing if replacements are unavailable. This finding has caused many organizations to move from legacy systems to commercial off-the-shelf (COTS) products to ensure that timely replacement is possible.

Software Backups

Most organizations' IT departments have their array of software disks and licensing information here or there—or possibly in one centralized location. If the facility were destroyed and the IT department's current environment had to be rebuilt, how would it gain access to these software packages? The BCP team should make sure to have an inventory of the necessary software required for mission-critical functions and have backup copies at an offsite facility. Hardware is usually not worth much to an organization without the software required to run on it. The software that needs to be backed up can be in the form of applications, utilities, databases, and operating systems. The business continuity plan must have provisions to back up and protect these items along with hardware and data.

It is common for organizations to work with software developers to create customized software programs. For example, in the banking world, individual financial institutions need software that enables their bank tellers to interact with accounts, hold account information in databases and mainframes, provide online banking, carry out data replication, and perform a thousand other types of bank-like functionalities. This specialized type of software is developed and available through a handful of software vendors that specialize in this market. When bank A purchases this type of software for all of its branches, the software has to be specially customized for its environment and needs. Once this banking software is installed, the whole organization depends upon it for its minute-by-minute activities.

When bank A receives the specialized and customized banking software from the software vendor, bank A does not receive the source code. Instead, the software vendor provides bank A with a compiled version. Now, what if this software vendor goes out of business because of a disaster or bankruptcy? Then bank A will require a new vendor to maintain and update this banking software; thus, the new vendor will need access to the source code.

The protection mechanism that bank A should implement is called *software escrow*, in which a third party holds the source code, backups of the compiled code, manuals, and other supporting materials. A contract between the software vendor, customer, and third party outlines who can do what, and when, with the source code. This contract usually states that the customer can have access to the source code only if and when the vendor goes out of business, is unable to carry out stated responsibilities, or is in breach of the original contract. If any of these activities takes place, then the customer is protected because it can still gain access to the source code and other materials through the third-party escrow agent.

Many organizations have been crippled by not implementing software escrow. They paid a software vendor to develop specialized software, and when the software vendor went belly up, the organizations did not have access to the code that their systems ran on.

End-User Environment

Because the end users are usually the worker bees of an organization, they must be provided a functioning environment as soon as possible after a disaster hits. This means that the BCP team must understand the current operational and technical functioning environment and examine critical pieces so they can replicate them.

In most situations, after a disaster, only a skeleton crew is put back to work. The BCP committee has previously identified the most critical functions of the organization during the analysis stage, and the employees who carry out those functions must be put back to work first. So the recovery process for the user environment should be laid out in different stages. The first stage is to get the most critical departments back online, the next stage is to get the second most important back online, and so on.

The BCP team needs to identify user requirements, such as whether users can work on stand-alone PCs or need to be connected in a network to fulfill specific tasks. For example, in a financial institution, users who work on stand-alone PCs might be able to accomplish some small tasks like filling out account forms, word processing, and accounting tasks, but they might need to be connected to a host system to update customer profiles and to interact with the database.

The BCP team also needs to identify how current automated tasks can be carried out manually if that becomes necessary. If the network is going to be down for 12 hours, could the necessary tasks be accomplished through traditional pen-and-paper methods? If the Internet connection is going to be down for five hours, could the necessary communications take place through phone calls? Instead of transmitting data through the internal mail system, could couriers be used to run information back and forth? Today, we are extremely dependent upon technology, but we often take for granted that it will always be there for us to use. It is up to the BCP team to realize that technology may be unavailable for a period of time and to come up with solutions for those situations.

EXAM TIP As a CISSP, your role in business continuity planning is most likely to be that of an active participant, not to lead it. BCP questions in the exam will be written with this in mind.

Chapter Review

There are four key take-aways in this chapter. The first is that you need to be able to identify and implement strategies that will enable your organization to recover from any disaster, supporting your organization's continuity of operations. Leveraging these strategies, you develop a detailed plan that includes the specific processes that the organization (and particularly the IT and security teams) will execute to recover from specific types of disasters. Thirdly, you have to know how to train your DR team to execute the plan flawlessly, even in the chaos of an actual disaster. This includes ensuring that everyone in the organization is aware of their role in the recovery efforts. Finally, the DRP is the cornerstone of the BCP, so you will be called upon to participate in broader business continuity planning and exercises, even if you are not in charge of that effort.

Quick Review

- Disaster recovery (DR) is the set of practices that enables an organization to minimize loss of, and restore, mission-critical technology infrastructure after a catastrophic incident.
- Business continuity (BC) is the set of practices that enables an organization to continue performing its critical functions through and after any disruptive event.
- The recovery time objective (RTO) is the maximum time period within which a mission-critical system must be restored to a designated service level after a disaster to avoid unacceptable consequences associated with a break in business continuity.
- The work recovery time (WRT) is the maximum amount of time available for certifying the functionality and integrity of restored systems and data so they can be put back into production.
- The recovery point objective (RPO) is the acceptable amount of data loss measured in time.
- The four commonly used data backup strategies are direct-attached storage, network-attached storage, cloud storage, and offline media.
- Electronic vaulting makes copies of files as they are modified and periodically transmits them to an offsite backup site.
- Remote journaling moves transaction logs to an offsite facility for database recovery, where only the reapplication of a series of changes to individual records is required to resynchronize the database.
- Offsite backup locations can supply hot, warm, or cold sites.
- A hot site is fully configured with hardware, software, and environmental needs. It can usually be up and running in a matter of hours. It is the most expensive option, but some organizations cannot be out of business longer than a day without very detrimental results.
- A warm site may have some computers, but it does have some peripheral devices, such as disk drives, controllers, and tape drives. This option is less expensive than a hot site, but takes more effort and time to become operational.
- A cold site is just a building with power, raised floors, and utilities. No devices are available. This is the cheapest of the three options, but can take weeks to get up and operational.
- In a reciprocal agreement, one organization agrees to allow another organization to use its facilities in case of a disaster, and vice versa. Reciprocal agreements are very tricky to implement and may be unenforceable. However, they offer a relatively cheap offsite option and are sometimes the only choice.
- A redundant (or mirrored) site is equipped and configured exactly like the primary site and is completely synchronized, ready to become the primary site at a moment's notice.

- High availability (HA) is a combination of technologies and processes that work together to ensure that some specific thing is up and running most of the time.
- Quality of service (QoS) defines minimum acceptable performance characteristics of a particular service, such as response time, CPU utilization, or network bandwidth utilization.
- Fault tolerance is the capability of a technology to continue to operate as expected even if something unexpected takes place (a fault).
- Resilience means that the system continues to function, albeit in a degraded fashion, when a fault is encountered.
- When returning to the original site after a disaster, the least critical organizational units should go back first.
- Disaster recovery plans can be tested through checklist tests, structured walkthroughs, tabletop exercises, simulation tests, parallel tests, or full-interruption tests.
- Business continuity planning addresses how to keep the organization in business after a major disruption takes place, but it is important to note that the scope is much broader than that of disaster recovery.
- The BCP life cycle includes developing the BC concept; assessing the current environment; implementing continuity strategies, plans, and solutions; training the staff; and testing, exercising, and maintaining the plans and solutions.
- An important part of the business continuity plan is to communicate its requirements and procedures to all employees.

Questions

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

1. Which best describes a hot-site facility versus a warm- or cold-site facility?

A. A site that has disk drives, controllers, and tape drives

B. A site that has all necessary PCs, servers, and telecommunications

C. A site that has wiring, central air-conditioning, and raised flooring

D. A mobile site that can be brought to the organization’s parking lot

2. Which of the following describes a cold site?

A. Fully equipped and operational in a few hours

B. Partially equipped with data processing equipment

C. Expensive and fully configured

D. Provides environmental measures but no equipment

3. Which is the best description of remote journaling?
   - **A.** Backing up bulk data to an offsite facility
   - **B.** Backing up transaction logs to an offsite facility
   - **C.** Capturing and saving transactions to two mirrored servers in-house
   - **D.** Capturing and saving transactions to different media types
4. Which of the following does not describe a reciprocal agreement?
   - **A.** The agreement is enforceable.
   - **B.** It is a cheap solution.
   - **C.** It may be able to be implemented right after a disaster.
   - **D.** It could overwhelm a current data processing site.
5. If a system is fault tolerant, what would you expect it to do?
   - **A.** Continue to operate as expected even if something unexpected takes place
   - **B.** Continue to function in a degraded fashion
   - **C.** Tolerate outages caused by known faults
   - **D.** Raise an alarm, but tolerate an outage caused by any fault
6. Which of the following approaches to testing your disaster recovery plan would be least desirable if you had to maintain high availability of over 99.999 percent?
   - **A.** Checklist test
   - **B.** Parallel test
   - **C.** Full-interruption test
   - **D.** Structured walkthrough test

Use the following scenario to answer Questions 7–10. You are the CISO of a small research and development (R&D) company and realize that you don't have a disaster recovery plan (DRP). The projects your organization handles are extremely sensitive and, despite having a very limited budget, you have to bring the risk of project data being lost as close to zero as you can. Recovery time is not as critical because you bill your work based on monthly deliverables and have some leeway at your disposal. Because of the sensitivity of your work, remote working is frowned upon and you keep your research data on local servers (including Exchange for e-mail, Mattermost for group chat, and Apache for web) at your headquarters (and only) site.

7. Which recovery site strategy would be best for you to consider?
   - **A.** Reciprocal agreement
   - **B.** Hot site
   - **C.** Warm site
   - **D.** Cold site

8. Which of the following recovery site characteristics would be best for your organization?
   - **A.** As close to headquarters as possible within budgetary constraints
   - **B.** 100 miles away from headquarters, on a different power grid
   - **C.** 15 miles away from headquarters on a different power grid
   - **D.** As far away from headquarters as possible
9. Which data backup storage strategy would you want to implement?
   - **A.** Direct-attached storage
   - **B.** Network-attached storage
   - **C.** Offline media
   - **D.** Cloud storage
10. Which of the following would be the best way to communicate with all members of the organization in the event of a disaster that takes out your site?
    - **A.** Internal Mattermost channel
    - **B.** External Slack channel
    - **C.** Exchange e-mail
    - **D.** Call trees

Answers

1. **B.** A hot site is a facility that is fully equipped and properly configured so that it can be up and running within hours to get an organization back into production. Answer B gives the best definition of a fully functional environment.
2. **D.** A cold site only provides environmental measures—wiring, HVAC, raised floors—basically a shell of a building and no more.
3. **B.** Remote journaling is a technology used to transmit data to an offsite facility, but this usually only includes moving the journal or transaction logs to the offsite facility, not the actual files.
4. **A.** A reciprocal agreement is not enforceable, meaning that the organization that agreed to let the damaged organization work out of its facility can decide not to allow this to take place. A reciprocal agreement is a better secondary backup option if the original plan falls through.
5. **A.** Fault tolerance is the capability of a technology to continue to operate as expected even if something unexpected takes place (a fault), with no degradations or outages.
6. **C.** A full-interruption test is the most intrusive to regular operations and business productivity. The original site is actually shut down, and processing takes place at the alternate site. This is almost guaranteed to exceed your allowed downtime unless everything went extremely well.

7. **D.** Because you are working on a tight budget and have the luxury of recovery time, you want to consider the least expensive option. A reciprocal agreement would be ideal except for the sensitivity of your data, which could not be shared with a similar organization (that could, presumably, be a competitor at some point). The next option (cost-wise) is a cold site, which would work in the given scenario.
8. **C.** An ideal recovery site would be on a different power grid to minimize the risk that power will be out on both sites, but close enough for employees to commute. This second point is important because, due to the sensitivity of your work, your organization has a low tolerance for remote work.
9. **C.** Since your data is critical enough that you have to bring the risk of it being lost as close to zero as you can, you would want to use offline media such as tape backups, optical discs, or even external drives that are disconnected after each backup (and potentially removed offsite). This is the slowest and most expensive approach, but is also the most resistant to attacks.
10. **B.** If your site is taken out, you would lose both Exchange and Mattermost since those servers are hosted locally. Call trees only work well for initial notification, leaving an externally hosted Slack channel as the best option. This would require your staff to be aware of this means of communication and have accounts created before the disaster.

PART VIII

Software Development Security

This page intentionally left blank

CHAPTER 24

Software Development

This chapter presents the following:

- Software development life cycle
- Development methodologies
- Operation and maintenance
- Maturity models

Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live.

—John F. Woods

Software is usually developed with a strong focus on functionality, not security. In many cases, security controls are bolted on as an afterthought (if at all). To get the best of both worlds, security and functionality have to be designed and integrated at each phase of the software development life cycle. Security should be interwoven into the core of a software product and provide protection at the necessary layers. This is a better approach than trying to develop a front end or wrapper that may reduce the overall functionality and leave security holes when the software has to be integrated into a production environment.

Before we get too deep into secure software development, however, we have to develop a shared understanding of how code is developed in the first place. In this chapter we will cover the complex world of software development so that we can understand the bad things that can happen when security is not interwoven into products properly (discussed in Chapter 25).

Software Development Life Cycle

The life cycle of software development deals with putting repeatable and predictable processes in place that help ensure functionality, cost, quality, and delivery schedule requirements are met. So instead of winging it and just starting to develop code for a project, how can we make sure we build the best software product possible?

Several *software development life cycle (SDLC)* models have been developed over the years, which we will cover later in this section, but the crux of each model deals with the following phases:

- **Requirements gathering** Determining *why* to create this software, *what* the software will do, and *for whom* the software will be created
- **Design** Encapsulating into a functional design *how* the software will accomplish the requirements
- **Development** Programming software code to meet specifications laid out in the design phase and integrating that code with existing systems and/or libraries
- **Testing** Verifying and validating software to ensure that the software works as planned and that goals are met
- **Operations and maintenance** Deploying the software and then ensuring that it is properly configured, patched, and monitored

EXAM TIP You don't need to memorize the phases of the SDLC. We discuss them here so you understand all the tasks that go into developing software and how to integrate security throughout the whole cycle.

In the following sections we will cover the different phases that make up an SDLC model and some specific items about each phase that are important to understand.

Software Development Roles

The specific roles within a software development team will vary based on the methodology being used, the maturity of the organization, and the size of the project (to name just a few parameters). Typically, however, a team has at least the following roles:

- **Project manager (PM)** This role has overall responsibility for the software development project, particularly with regard to cost, schedule, performance, and risk.
- **Team leads** It is rare for software projects to be tackled by a single team, so we usually divide them up and assign a good developer to lead each part.
- **Architect** Sometimes called a tech lead, this role figures out what technologies to use internally or when interfacing with external systems.
- **Software engineer** The people who actually write the programming code are oftentimes specialists in either frontends (e.g., user interfaces) or various types of backends (e.g., business logic, databases). Engineers that can do all of this are called full-stack developers.
- **Quality assurance (QA)** Whether this is a single person or an entire team, this role implements and runs testing processes that detect software defects as early as possible.

Keep in mind that the discussion that follows covers phases that may happen repeatedly and in limited scope depending on the development methodology being used. Before we get into the phases of the SDLC, let's take a brief look at the glue that holds them together: project management.

Project Management

Many developers know that good project management keeps the project moving in the right direction, allocates the necessary resources, provides the necessary leadership, and hopes for the best but plans for the worst. Project management processes should be put into place to make sure the software development project executes each life-cycle phase properly. Project management is an important part of product development, and security management is an important part of project management.

The project manager draws up a security plan at the beginning of a development project and integrates it into the functional plan to ensure that security is not overlooked. This plan will probably be broad and should refer to documented references for more detailed information. The references could include computer standards (RFCs, IEEE standards, and best practices), documents developed in previous projects, security policies, accreditation statements, incident-handling plans, and national or international guidelines. This helps ensure that the plan stays on target.

The security plan should have a life cycle of its own. It will need to be added to, subtracted from, and explained in more detail as the project continues. Keeping the security plan up to date for future reference is important, because losing track of actions, activities, and decisions is very easy once a large and complex project gets underway.

The security plan and project management activities could be scrutinized later, particularly if a vulnerability causes losses to a third party, so we should document security-related decisions. Being able to demonstrate that security was fully considered in each phase of the SDLC can prove that the team exercised due care and this, in turn, can mitigate future liabilities. To this end, the documentation must accurately reflect how the product was built and how it is supposed to operate once implemented into an environment.

If a software product is being developed for a specific customer, it is common for a *Statement of Work (SOW)* to be developed, which describes the product and customer requirements. A detailed SOW helps to ensure that all stakeholders understand these requirements and don't make any undocumented assumptions.

Sticking to what is outlined in the SOW is important so that *scope creep* does not take place. If the scope of a project continually extends (creeps) in an uncontrollable manner, the project may never end, not meet its goals, run out of funding, or all of the foregoing. If the customer wants to modify its requirements, it is important that the SOW is updated and funding is properly reviewed.

A *work breakdown structure (WBS)* is a project management tool used to define and group a project's individual work elements in an organized manner. It is a deliberate decomposition of the project into tasks and subtasks that result in clearly defined deliverables. The SDLC should be illustrated in a WBS format, so that each phase is properly addressed.

Requirements Gathering Phase

This is the phase in which everyone involved in the software development project attempts to understand why the project is needed and what the scope of the project entails. Typically, either a specific customer needs a new application or a demand for the product exists in the market. During this phase, the software development team examines the software's requirements and proposed functionality, engages in brainstorming sessions, and reviews obvious restrictions.

A conceptual definition of the project should be initiated and developed to ensure everyone is on the right page and that this is a proper product to develop. This phase could include evaluating products currently on the market and identifying any demands not being met by current vendors. This definition could also be a direct request for a specific product from a current or future customer.

Typically, the following tasks should be accomplished in this phase:

- Requirements gathering (including security ones)
- Security risk assessment
- Privacy risk assessment
- Risk-level acceptance

The security requirements of the product should be defined in the categories of availability, integrity, and confidentiality. What type of security is required for the software product and to what degree? Some of these requirements may come from applicable external regulations. For example, if the application will deal with payment cards, PCI DSS will dictate some requirements, such as encryption for card information.

An initial security risk assessment should be carried out to identify the potential threats and their associated consequences. This process usually involves asking many, many questions to elicit and document the laundry list of vulnerabilities and threats, the probability of these vulnerabilities being exploited, and the outcome if one of these threats actually becomes real and a compromise takes place. The questions vary from product to product—such as its intended purpose, the expected environment it will be implemented in, the personnel involved, and the types of businesses that would purchase and use the product.

The sensitivity level of the data that many software products store and process has only increased in importance over the years. After a *privacy risk assessment*, a *privacy impact rating* can be assigned, which indicates the sensitivity level of the data that will be processed or accessible. Some software vendors incorporate the following privacy impact ratings in their software development assessment processes:

- **P1, High Privacy Risk** The feature, product, or service stores or transfers personally identifiable information (PII), monitors the user with an ongoing transfer of anonymous data, changes settings or file type associations, or installs software.
- **P2, Moderate Privacy Risk** The sole behavior that affects privacy in the feature, product, or service is a one-time, user-initiated, anonymous data transfer (e.g., the user clicks a link and is directed to a website).

- **P3, Low Privacy Risk** No behaviors exist within the feature, product, or service that affect privacy. No anonymous or personal data is transferred, no PII is stored on the machine, no settings are changed on the user's behalf, and no software is installed.

The software vendor can develop its own privacy impact ratings and their associated definitions. As of this writing there are several formal approaches to conducting a privacy risk assessment, but none stands out as "the" standardized approach to defining a methodology for an assessment or these rating types, but as privacy increases in importance, we might see more standardization in these ratings and associated metrics.

The team tasked with documenting the requirements must understand the criteria for risk-level acceptance to make sure that mitigation efforts satisfy these criteria. Which risks are acceptable will depend on the results of the security and privacy risk assessments. The evaluated threats and vulnerabilities are used to estimate the cost/benefit ratios of the different security countermeasures. The level of each security attribute should be focused upon so that a clear direction on security controls can begin to take shape and can be integrated into the design and development phases.

The end state of the requirements gathering phase is typically a document called the Software (or System) Requirements Specification (SRS), which describes what the software will do and how it will perform. These two high-level objectives are also known as functional and nonfunctional requirements. A *functional requirement* describes a feature of the software system, such as reporting product inventories or processing customer orders. A *nonfunctional requirement* describes performance standards, such as the minimum number of simultaneous user sessions or the maximum response time for a query. Nonfunctional requirements also include security requirements, such as what data must be encrypted and what the acceptable cryptosystems are. The SRS, in a way, is a checklist that the software development team will use to develop the software and the customer will use to accept it.

The Unified Modeling Language (UML) is a common language used to graphically describe all aspects of software development. We will revisit it throughout the different phases, but in terms of software requirements, it allows us to capture both functional and nonfunctional requirements with use case diagrams (UCDs). We already saw these in Chapter 18 when we discussed testing of technical controls. If you look back to Figure 18-3, each use case (shown as verb phrases inside ovals) represents a high-level functional requirement. The associations can capture nonfunctional requirements through special labels, or these requirements can be spelled out in an accompanying use case description.

Design Phase

Once the requirements are formally documented, the software development team can begin figuring out how they will go about satisfying them. This is the phase that starts to map theory to reality. The theory encompasses all the requirements that were identified in the previous phase, and the design outlines how the product is actually going to accomplish these requirements.

Some organizations skip the design phase, but this can cause major delays and redevelopment efforts down the road because a broad vision of the product needs to be understood before looking strictly at the details. Instead, software development teams should develop written plans for how they will build software that satisfies each requirement. This plan usually comprises three different but interrelated models:

- **Informational model** Dictates the type of information to be processed and how it will move around the software system
- **Functional model** Outlines the tasks and functions the application needs to carry out and how they are sequenced and synchronized
- **Behavioral model** Explains the states the application will be in during and after specific transitions take place

For example, consider an antimalware software application. Its informational model would dictate how it processes information, such as virus signatures, modified system files, checksums on critical files, and virus activity. Its functional model would dictate how it scans a hard drive, checks e-mail for known virus signatures, monitors critical system files, and updates itself. Its behavioral model would indicate that when the system starts up, the antimalware software application will scan the hard drive and memory segments. The computer coming online would be the event that changes the state of the application. If it finds a virus, the application would change state and deal with the virus appropriately. Each state must be accounted for to ensure that the product does not go into an insecure state and act in an unpredictable way.

The data from the informational, functional, and behavioral models is incorporated into the software design document, which includes the data, architectural, and procedural design, as shown in Figure 24-1.

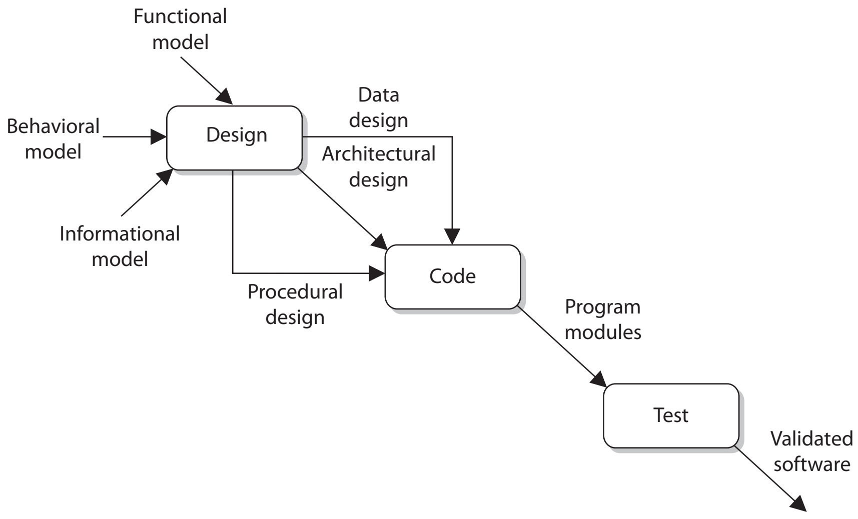


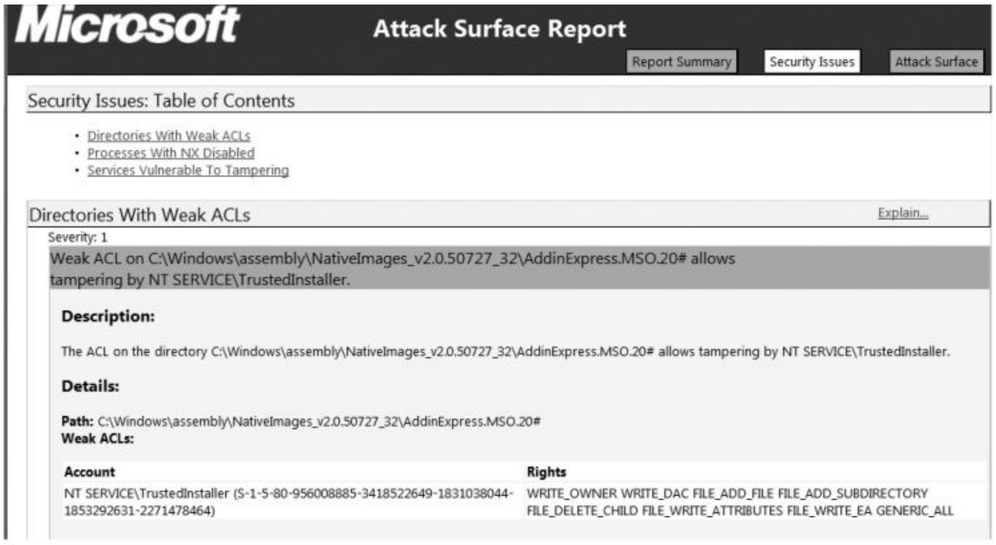
Figure 24-1 Information from three models can go into the design.

From a security point of view, the following items should also be accomplished in the design phase:

- Attack surface analysis
- Threat modeling

An *attack surface* is what is available to be used by an attacker against the product itself. As an analogy, if you were wearing a suit of armor and it covered only half of your body, the other half would be your vulnerable attack surface. Before you went into battle, you would want to reduce this attack surface by covering your body with as much protective armor as possible. The same can be said about software. The software development team should reduce the attack surface as much as possible because the greater the attack surface of software, the more avenues for the attacker; and hence, the greater the likelihood of a successful compromise.

The aim of an *attack surface analysis* is to identify and reduce the amount of code and functionality accessible to untrusted users. The basic strategies of attack surface reduction are to reduce the amount of code running, reduce entry points available to untrusted users, reduce privilege levels as much as possible, and eliminate unnecessary services. Attack surface analysis is generally carried out through specialized tools to enumerate different parts of a product and aggregate their findings into a numeral value. Attack surface analyzers scrutinize files, Registry keys, memory data, session information, processes, and services details. A sample attack surface report is shown in Figure 24-2.



Security Issues: Table of Contents

- Directories With Weak ACLs
- Processes With NX Disabled
- Services Vulnerable To Tampering

Directories With Weak ACLs

Explain...

Severity: 1

Weak ACL on C:\Windows\assembly\NativeImages_v2.0.50727_32\AddinExpress.MSO.20# allows tampering by NT SERVICE\TrustedInstaller.

Description:

The ACL on the directory C:\Windows\assembly\NativeImages_v2.0.50727_32\AddinExpress.MSO.20# allows tampering by NT SERVICE\TrustedInstaller.

Details:

Path: C:\Windows\assembly\NativeImages_v2.0.50727_32\AddinExpress.MSO.20#
Weak ACLs:

Account	Rights
NT SERVICE\TrustedInstaller (S-1-5-80-956008885-3418522649-1831038044-1853292631-2271478464)	WRITE_OWNER WRITE_DAC FILE_ADD_FILE FILE_ADD_SUBDIRECTORY FILE_DELETE_CHILD FILE_WRITE_ATTRIBUTES FILE_WRITE_EA GENERIC_ALL

Figure 24-2 Attack surface analysis result

Threat modeling, which we covered in detail in Chapter 9 in the context of risk management, is a systematic approach used to understand how different threats could be realized and how a successful compromise could take place. As a hypothetical example, if you were responsible for ensuring that the government building in which you work is safe from terrorist attacks, you would run through scenarios that terrorists would most likely carry out so that you fully understand how to protect the facility and the people within it. You could think through how someone could bring a bomb into the building, and then you would better understand the screening activities that need to take place at each entry point. A scenario of someone running a car into the building would bring up the idea of implementing bollards around the sensitive portions of the facility. The scenario of terrorists entering sensitive locations in the facility (data center, CEO office) would help illustrate the layers of physical access controls that should be implemented.

These same scenario-based exercises should take place during the design phase of software development. Just as you would think about how potential terrorists could enter and exit a facility, the software development team should think through how potentially malicious activities can happen at different input and output points of the software and the types of compromises that can take place within the guts of the software itself.

It is common for software development teams to develop threat trees, as shown in Figure 24-3. A *threat tree* is a tool that allows the development team to understand all the ways specific threats can be realized; thus, it helps them understand what type of security controls they should implement in the software to mitigate the risks associated with each threat type.

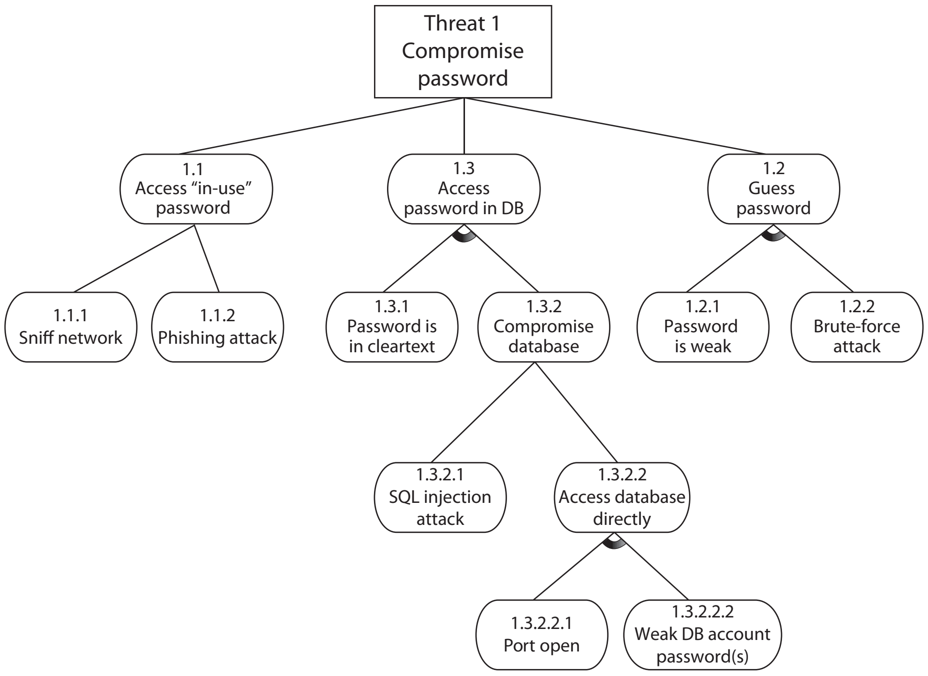

Threat 1 Compromise password
1.1 Access "in-use" password
1.3 Access password in DB
1.2 Guess password
1.1.1 Sniff network
1.1.2 Phishing attack
1.3.1 Password is in cleartext
1.3.2 Compromise database
1.2.1 Password is weak
1.2.2 Brute-force attack
1.3.2.1 SQL injection attack
1.3.2.2 Access database directly
1.3.2.2.1 Port open
1.3.2.2.2 Weak DB account password(s)

Figure 24-3 Threat tree used in threat modeling

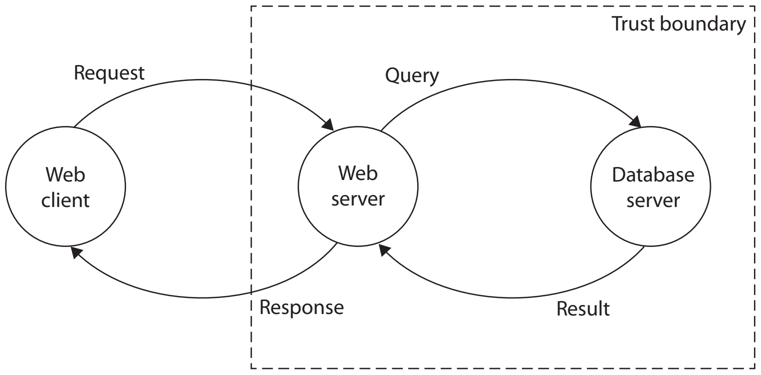


Figure 24-4 A simple flow diagram for threat modeling

There are many automated tools in the industry that software development teams can use to ensure that they address the various threat types during the design stage. One popular open-source solution is the Open Web Application Security Project (OWASP) Threat Dragon. This web-based tool enables the development team to describe threats visually using flow diagrams. Figure 24-4 shows a simple diagram of a three-tier web system showing its trust boundary and the four ways in which the tiers interact. The next step in building the threat model would be to consider how each of these four interactions could be exploited by a threat actor. For example, stolen credentials could allow an adversary to compromise the web server and, from there, issue queries to the database server that could compromise the integrity or availability of records stored there. For each threat identified through this process, the software development team would develop controls to mitigate it.

The decisions made during the design phase are pivotal steps to the development phase. Software design serves as a foundation and greatly affects software quality. If good product design is not put into place in the beginning of the project, the following phases will be much more challenging.

Development Phase

This is the phase where the programmers become deeply involved. The software design that was created in the previous phase is broken down into defined deliverables, and programmers develop code to meet the deliverable requirements.

There are many *computer-aided software engineering (CASE)* tools that programmers can use to generate code, test software, and carry out debugging activities. When these types of activities are carried out through automated tools, development usually takes place more quickly with fewer errors.

CASE refers to any type of software tool that supports automated development of software, which can come in the form of program editors, debuggers, code analyzers, version-control mechanisms, and more. These tools aid in keeping detailed records of requirements, design steps, programming activities, and testing. A CASE tool is designed to support one or more software engineering tasks in the process of developing software. Many vendors can get their products to the market faster because they are "computer aided."

In the next chapter we will delve into the abyss of "secure coding," but let's take a quick peek at it here to illustrate its importance in the development phase. As stated previously, most vulnerabilities that corporations, organizations, and individuals have to worry about reside within the programming code itself. When programmers do not follow strict and secure methods of creating programming code, the effects can be widespread and the results can be devastating. But programming securely is not an easy task. The list of errors that can lead to serious vulnerabilities in software is long.

The MITRE organization's Common Weakness Enumeration (CWE) initiative (https://cwe.mitre.org/top25) describes "a demonstrative list of the most common and impactful issues experienced over the previous two calendar years." Table 24-1 shows the most recent list.

Rank	Name
1	Out-of-bounds Write
2	Improper Neutralization of Input During Web Page Generation ("Cross-site Scripting")
3	Out-of-bounds Read
4	Improper Input Validation
5	Improper Neutralization of Special Elements used in an OS Command ("OS Command Injection")
6	Improper Neutralization of Special Elements used in an SQL Command ("SQL Injection")
7	Use After Free
8	Improper Limitation of a Pathname to a Restricted Directory ("Path Traversal")
9	Cross-Site Request Forgery (CSRF)
10	Unrestricted Upload of File with Dangerous Type
11	Missing Authentication for Critical Function
12	Integer Overflow or Wraparound
13	Deserialization of Untrusted Data
14	Improper Authentication
15	NULL Pointer Dereference
16	Use of Hard-coded Credentials
17	Improper Restriction of Operations within the Bounds of a Memory Buffer
18	Missing Authorization
19	Incorrect Default Permissions
20	Exposure of Sensitive Information to an Unauthorized Actor
21	Insufficiently Protected Credentials
22	Incorrect Permission Assignment for Critical Resource
23	Improper Restriction of XML External Entity Reference
24	Server-Side Request Forgery (SSRF)
25	Improper Neutralization of Special Elements used in a Command ("Command Injection")

Table 24-1 2021 CWE Top 25 Most Dangerous Software Weaknesses List

Many of these software issues are directly related to improper or faulty programming practices. Among other issues to address, the programmers need to check input lengths so buffer overflows cannot take place, inspect code to prevent the presence of covert channels, check for proper data types, make sure checkpoints cannot be bypassed by users, verify syntax, and verify checksums. The software development team should play out different attack scenarios to see how the code could be attacked or modified in an unauthorized fashion. Code reviews and debugging should be carried out by peer developers, and everything should be clearly documented.

A particularly important area of scrutiny is input validation because it can lead to serious vulnerabilities. Essentially, we should treat every single user input as malicious until proven otherwise. For example, if we don't put limits on how many characters users can enter when providing, say, their names on a web form, they could cause a buffer overflow, which is a classic example of a technique used to exploit improper input validation. A *buffer overflow* (which is described in detail in Chapter 18) takes place when too much data is accepted as input to a specific process. The process's memory buffer can be overflowed by shoving arbitrary data into various memory segments and inserting a carefully crafted set of malicious instructions at a specific memory address.

Buffer overflows can also lead to illicit escalation of privileges. *Privilege escalation* is the process of exploiting a process or configuration setting to gain access to resources that would normally not be available to the process or its user. For example, an attacker can compromise a regular user account and escalate its privileges to gain administrator or even system privileges on that computer. This type of attack usually exploits the complex interactions of user processes with device drivers and the underlying operating system. A combination of input validation and configuring the system to run with least privilege can help mitigate the threat of escalation of privileges.

What is important to understand is that secure coding practices need to be integrated into the development phase of the SDLC. Security has to be addressed at each phase of the SDLC, with this phase being one of the most critical.

Testing Phase

Formal and informal testing should begin as soon as possible. *Unit testing* is concerned with ensuring the quality of individual code modules or classes. Mature developers develop the unit tests for their modules before they even start coding, or at least in parallel with the coding. This approach is known as *test-driven development* and tends to result in much higher-quality code with significantly fewer vulnerabilities.

Unit tests are meant to simulate a range of inputs to which the code may be exposed. These inputs range from the mundanely expected, to the accidentally unfortunate, to the intentionally malicious. The idea is to ensure the code always behaves in an expected and secure manner. Once a module and its unit tests are finished, the unit tests are run (usually in an automated framework) on that code. The goal of this type of testing is to isolate each part of the software and show that the individual parts are correct.

Unit testing usually continues throughout the development phase. A totally different group of people should carry out the formal testing. Depending on the methodology and the organization, this could be a QA, testing, audit, or even red team. This is an example

Separation of Duties

Different environmental types (development, testing, and production) should be properly separated, and functionality and operations should not overlap. Developers should not have access to modify code used in production. The code should be tested, submitted to a library, and then sent to the production environment.

of separation of duties. A programmer should not develop, test, and release software. The more eyes that see the code, the greater the chance that flaws will be found before the product is released.

No cookie-cutter recipe exists for security testing because the applications and products can be so diverse in functionality and security objectives. It is important to map security risks to test cases and code. The software development team can take a linear approach by identifying a vulnerability, providing the necessary test scenario, performing the test, and reviewing the code for how it deals with such a vulnerability. At this phase, tests are conducted in an environment that should mirror the production environment to ensure the code does not work only in the labs.

Security attacks and penetration tests usually take place during the testing phase to identify any missed vulnerabilities. Functionality, performance, and penetration resistance are evaluated. All the necessary functionality required of the product should be in a checklist to ensure each function is accounted for.

Security tests should be run to test against the vulnerabilities identified earlier in the project. Buffer overflows should be attempted, interfaces should be hit with unexpected inputs, denial-of-service (DoS) situations should be tested, unusual user activity should take place, and if a system crashes, the product should react by reverting to a secure state. The product should be tested in various environments with different applications, configurations, and hardware platforms. A product may respond fine when installed on a clean Windows 10 installation on a stand-alone PC, but it may throw unexpected errors when installed on a laptop that is remotely connected to a network and has a virtual private network (VPN) client installed.

Verification vs. Validation

Verification determines if the software product accurately represents and meets the specifications. After all, a product can be developed that does not match the original specifications, so this step ensures the specifications are being properly met. It answers the question, "Did we build the product right?"

Validation determines if the software product provides the necessary solution for the intended real-world problem. In large projects, it is easy to lose sight of the overall goal. This exercise ensures that the main goal of the project is met. It answers the question, "Did we build the right product?"

Testing Types

Software testers on the software development team should subject the software to various types of tests to discover the variety of potential flaws. The following are some of the most common testing approaches:

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

A well-rounded security test encompasses both manual tests and automated tests. Automated tests help locate a wide range of flaws generally associated with careless or erroneous code implementations. Some automated testing environments run specific inputs in a scripted and repeatable manner. While these tests are the bread and butter of software testing, we sometimes want to simulate random and unpredictable inputs to supplement the scripted tests.

A manual test is used to analyze aspects of the program that require human intuition and can usually be judged using computing techniques. Testers also try to locate design flaws. These include logical errors, which may enable attackers to manipulate program flow by using shrewdly crafted program sequences to access greater privileges or bypass authentication mechanisms. Manual testing involves code auditing by security-centric programmers who try to modify the logical program structure using rogue inputs and reverse-engineering techniques. Manual tests simulate the live scenarios involved in real-world attacks. Some manual testing also involves the use of social engineering to analyze the human weakness that may lead to system compromise.

At this stage, issues found in testing procedures are relayed to the development team in problem reports. The problems are fixed and programs retested. This is a continual process until everyone is satisfied that the product is ready for production. If there is a specific customer, the customer would run through a range of tests before formally accepting the product; if it is a generic product, beta testing can be carried out by various potential customers and agencies. Then the product is formally released to the market or customer.

NOTE Sometimes developers include lines of code in a product that will allow them to do a few keystrokes and get right into the application. This allows them to bypass any security and access controls so they can quickly access the application's core components. This is referred to as a "back door" or "maintenance hook" and must be removed before the code goes into production.

Operations and Maintenance Phase

Once the software code is developed and properly tested, it is released so that it can be implemented within the intended production environment. The software development team's role is not finished at this point. Newly discovered problems and vulnerabilities are commonly

PART VIII

identified at this phase. For example, if a company developed a customized application for a specific customer, the customer could run into unforeseen issues when rolling out the product within its various networked environments. Interoperability issues might come to the surface, or some configurations may break critical functionality. The developers would need to make the necessary changes to the code, retest the code, and re-release the code.

Almost every software system requires the addition of new features over time. Frequently, these have to do with changing business processes or interoperability with other systems. This highlights the need for the operations and development teams to work particularly closely during the operations and maintenance (O&M) phase. The operations team, which is typically the IT department, is responsible for ensuring the reliable operation of all production systems. The development team is responsible for any changes to the software in development systems up until the time the software goes into production. Together, the operations and development teams address the transition from development to production as well as management of the system's configuration.

Another facet of O&M is driven by the fact that new vulnerabilities are regularly discovered. While the developers may have carried out extensive security testing, it is close to impossible to identify all the security issues at one point and time. Zero-day vulnerabilities may be identified, coding errors may be uncovered, or the integration of the software with another piece of software may uncover security issues that have to be addressed. The development team must develop patches, hotfixes, and new releases to address these items. In all likelihood, this is where you as a CISSP will interact the most with the SDLC.

Change Management

One of the key processes on which to focus for improvement involves how we deal with the inevitable changes. These can cause a lot of havoc if not managed properly and in a deliberate manner. We already discussed change management in general in Chapter 20, but it is particularly important during the lifetime of a software development project.

The need to change software arises for several reasons. During the development phase, a customer may alter requirements and ask that certain functionalities be added, removed, or modified. In production, changes may need to happen because of other changes in the environment, new requirements of a software product or system, or newly released patches or upgrades. These changes should be carefully analyzed, approved, and properly incorporated such that they do not affect any original functionality in an adverse way.

Change management is a systematic approach to deliberately regulating the changing nature of projects, including software development projects. It is a management process that takes into account not just the technical issues but also resources (like people and money), project life cycle, and even organizational climate. Many times, the hardest part of managing change is not the change itself, but the effects it has in the organization. Many of us have been on the receiving end of a late-afternoon phone call in which we're told to change our plans because of a change in a project on which we weren't even working. An important part of change management is controlling change.

Change Control

Change control is the process of controlling the specific changes that take place during the life cycle of a system and documenting the necessary change control activities. Whereas change management is the project manager's responsibility as an overarching process, change control is what developers do to ensure the software doesn't break when they change it.

Change control involves a bunch of things to consider. The change must be approved, documented, and tested. Some tests may need to be rerun to ensure the change does not affect the product's capabilities. When a programmer makes a change to source code, she should do so on the test version of the code. Under no conditions should a programmer change the code that is already in production. After making changes to the code, the programmer should test the code and then deliver the new code to the librarian. Production code should come only from the librarian and not from a programmer or directly from a test environment.

A process for controlling changes needs to be in place at the beginning of a project so that everyone knows how to deal with changes and knows what is expected of each entity when a change request is made. Some projects have been doomed from the start because proper change control was not put into place and enforced. Many times in development, the customer and vendor agree on the design of the product, the requirements, and the specifications. The customer is then required to sign a contract confirming this is the agreement and that if they want any further modifications, they will have to pay the vendor for that extra work. If this agreement is not put into place, then the customer can continually request changes, which requires the software development team to put in the extra hours to provide these changes, the result of which is that the vendor loses money, the product does not meet its completion deadline, and scope creep occurs.

Other reasons exist to have change control in place. These reasons deal with organizational policies, standard procedures, and expected results. If a software product is in the last phase of development and a change request comes in, the development team should know how to deal with it. Usually, the team leader must tell the project manager how much extra time will be required to complete the project if this change is incorporated and what steps need to be taken to ensure this change does not affect other components within the product. If these processes are not controlled, one part of a development team could implement the change without another part of the team being aware of it. This could break some of the other development team's software pieces. When the pieces of the product are integrated and some pieces turn out to be incompatible, some jobs may be in jeopardy, because management never approved the change in the first place.

Change control processes should be evaluated during system audits. It is possible to overlook a problem that a change has caused in testing, so the procedures for how change control is implemented and enforced should be examined during a system audit.

The following are some necessary steps for a change control process:

1. Make a formal request for a change.
2. Analyze the request:
    a. Develop the implementation strategy.
    b. Calculate the costs of this implementation.
    c. Review security implications.
3. Record the change request.
4. Submit the change request for approval.

5. Develop the change:
   a. Recode segments of the product and add or subtract functionality.
   b. Link these changes in the code to the formal change control request.
   c. Submit software for testing and quality control.
   d. Repeat until quality is adequate.
   e. Make version changes.
6. Report results to management.

The changes to systems may require another round of certification and accreditation. If the changes to a system are significant, then the functionality and level of protection

SDLC and Security

The main phases of a software development life cycle are shown here with some specific security tasks.

Requirements gathering:

- Security risk assessment
- Privacy risk assessment
- Risk-level acceptance
- Informational, functional, and behavioral requirements

Design:

- Attack surface analysis
- Threat modeling

Development:

- Automated CASE tools
- Secure coding

Testing:

- Automated testing
- Manual testing
- Unit, integration, acceptance, and regression testing

Operations and maintenance:

- Vulnerability patching
- Change management and control

may need to be reevaluated (certified), and management would have to approve the overall system, including the new changes (accreditation).

Development Methodologies

Several software development methodologies are in common use around the world. While some include security issues in certain phases, these are not considered "security-centric development methodologies." They are simply classical approaches to building and developing software. Let's dive into some of the methodologies that you should know as a CISSP.

EXAM TIP It is exceptionally rare to see a development methodology used in its pure form in the real world. Instead, organizations typically start with a base methodology and modify it to suit their own unique environment. For purposes of the CISSP exam, however, you should focus on what differentiates each development approach.

Waterfall Methodology

The *Waterfall methodology* uses a linear-sequential life-cycle approach, illustrated in Figure 24-5. Each phase must be completed in its entirety before the next phase can begin. At the end of each phase, a review takes place to make sure the project is on the correct path and should continue.

In this methodology all requirements are gathered in the initial phase and there is no formal way to integrate changes as more information becomes available or requirements change. It is hard to know everything at the beginning of a project, so waiting until the whole project is complete to integrate necessary changes can be ineffective and time consuming. As an analogy, let's say that you are planning to landscape your backyard that is one acre in size. In this scenario, you can go to the gardening store only one time to get

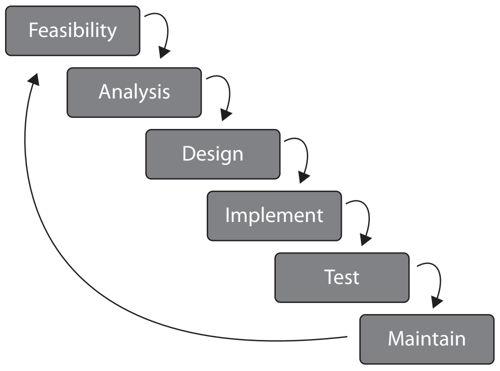


Figure 24-5 Waterfall methodology used for software development

all your supplies. If you identify during the project that you need more topsoil, rocks, or pipe for the sprinkler system, you have to wait and complete the whole yard before you can return to the store for extra or more suitable supplies.

The Waterfall methodology is a very rigid approach that could be useful for smaller projects in which all the requirements are fully understood up front. It may also be a good choice in some large projects for which different organizations will perform the work at each phase. Overall, however, it is not an ideal methodology for most complex projects, which commonly contain many variables that affect the scope as the project continues.

Prototyping

A *prototype* is a sample of software code or a model that can be developed to explore a specific approach to a problem before investing expensive time and resources. A team can identify the usability and design problems while working with a prototype and adjust their approach as necessary. Within the software development industry, three main prototype models have been invented and used. These are the rapid prototype, evolutionary prototype, and operational prototype.

Rapid prototyping is an approach that allows the development team to quickly create a prototype (sample) to test the validity of the current understanding of the project requirements. In a software development project, the team could develop a *rapid prototype* to see if their ideas are feasible and if they should move forward with their current solution. The rapid prototype approach (also called throwaway) is a "quick and dirty" method of creating a piece of code and seeing if everyone is on the right path or if another solution should be developed. The rapid prototype is not developed to be built upon, but to be discarded after serving its purposes.

When *evolutionary prototypes* are developed, they are built with the goal of incremental improvement. Instead of being discarded after being developed, as in the rapid prototype approach, the evolutionary prototype is continually improved upon until it reaches the final product stage. Feedback that is gained through each development phase is used to improve the prototype and get closer to accomplishing the customer's needs.

Operational prototypes are an extension of the evolutionary prototype method. Both models (operational and evolutionary) improve the quality of the prototype as more data is gathered, but the operational prototype is designed to be implemented within a production environment as it is being tweaked. The operational prototype is updated as customer feedback is gathered, and the changes to the software happen within the working site.

In summary, a rapid prototype is developed to give a quick understanding of the suggested solution, an evolutionary prototype is created and improved upon within a lab environment, and an operational prototype is developed and improved upon within a production environment.

Incremental Methodology

If a development team follows the *Incremental methodology*, this allows them to carry out multiple development cycles on a piece of software throughout its development stages. This would be similar to "multi-Waterfall" cycles taking place on one piece of software as

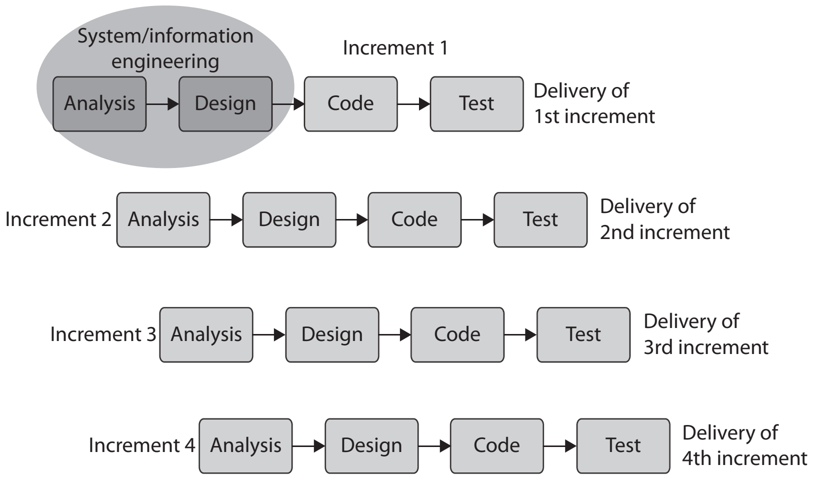


Figure 24-6 Incremental development methodology

it matures through the development stages. A version of the software is created in the first iteration and then it passes through each phase (requirements analysis, design, coding, testing, implementation) of the next iteration process. The software continues through the iteration of phases until a satisfactory product is produced. This methodology is illustrated in Figure 24-6.

When using the Incremental methodology, each incremental phase results in a deliverable that is an operational product. This means that a working version of the software is produced after the first iteration and that version is improved upon in each of the subsequent iterations. Some benefits to this methodology are that a working piece of software is available in early stages of development, the flexibility of the methodology allows for changes to take place, testing uncovers issues more quickly than the Waterfall methodology since testing takes place after each iteration, and each iteration is an easily manageable milestone.

Because each incremental phase delivers an operational product, the customer can respond to each build and help the development team in its improvement processes, and because the initial product is delivered more quickly compared to other methodologies, the initial product delivery costs are lower, the customer gets its functionality earlier, and the risks of critical changes being introduced are lower.

This methodology is best used when issues pertaining to risk, program complexity, funding, and functionality requirements need to be understood early in the product development life cycle. If a vendor needs to get the customer some basic functionality quickly as it works on the development of the product, this can be a good methodology to follow.

Spiral Methodology

The *Spiral methodology* uses an iterative approach to software development and places emphasis on risk analysis. The methodology is made up of four main phases: determine objectives, identify and resolve risks, development and test, and plan the next iteration. The development team starts with the initial requirements and goes through each of these phases, as shown in Figure 24-7. Think about starting a software development project at the center of this graphic. You have your initial understanding and requirements of the project, develop specifications that map to these requirements, identify and resolve risks, build prototype specifications, test your specifications, build a development plan, integrate newly discovered information, use the new information to carry out a new risk analysis, create a prototype, test the prototype, integrate resulting data into the process, and so forth. As you gather more information about the project, you integrate it into the risk analysis process, improve your prototype, test the prototype, and add more granularity to each step until you have a completed product.

The iterative approach provided by the Spiral methodology allows new requirements to be addressed as they are uncovered. Each prototype allows for testing to take place

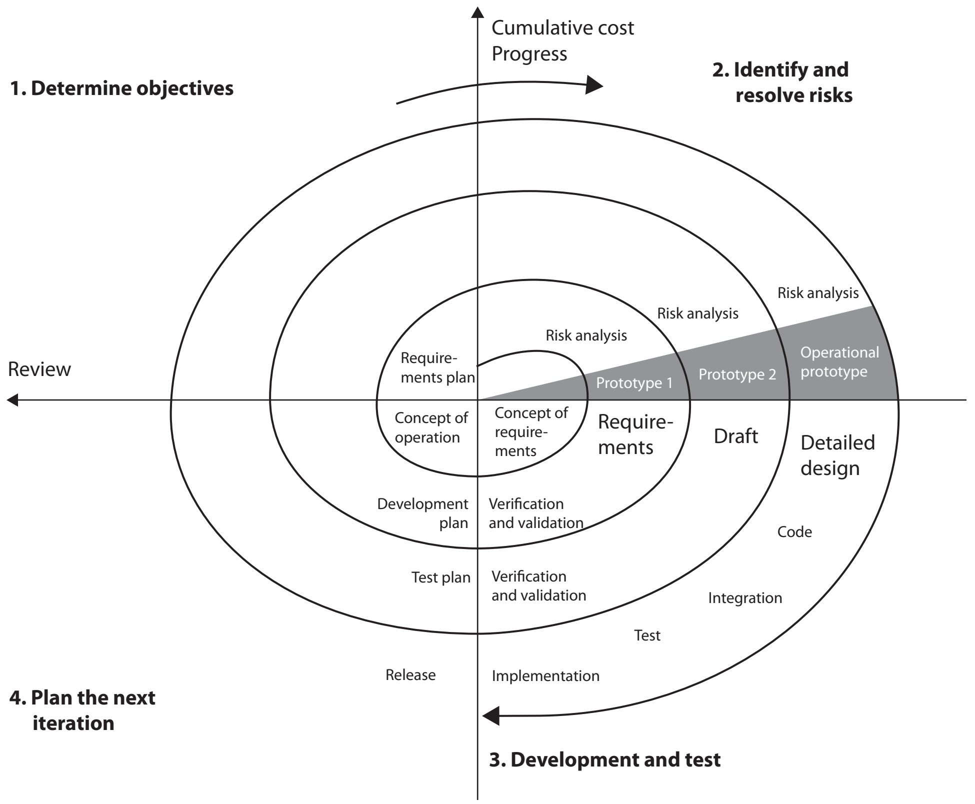


Figure 24-7 Spiral methodology for software development

early in the development project, and feedback based upon these tests is integrated into the following iteration of steps. The risk analysis ensures that all issues are actively reviewed and analyzed so that things do not "slip through the cracks" and the project stays on track.

In the Spiral methodology the last phase allows the customer to evaluate the product in its current state and provide feedback, which is an input value for the next spiral of activity. This is a good methodology for complex projects that have fluid requirements.

NOTE Within this methodology the angular aspect represents progress and the radius of the spirals represents cost.

Rapid Application Development

The *Rapid Application Development (RAD)* methodology relies more on the use of rapid prototyping than on extensive upfront planning. In this methodology, the planning of how to improve the software is interleaved with the processes of developing the software, which allows for software to be developed quickly. The delivery of a workable piece of software can take place in less than half the time compared to the Waterfall methodology. The RAD methodology combines the use of prototyping and iterative development procedures with the goal of accelerating the software development process. The development process begins with creating data models and business process models to help define what the end-result software needs to accomplish. Through the use of prototyping, these data and process models are refined. These models provide input to allow for the improvement of the prototype, and the testing and evaluation of the prototype allow for the improvement of the data and process models. The goal of these steps is to combine business requirements and technical design statements, which provide the direction in the software development project.

Figure 24-8 illustrates the basic differences between traditional software development approaches and RAD. As an analogy, let's say that the development team needs you to tell them what it is you want so that they can build it for you. You tell them that the thing you want has four wheels and an engine. They bring you a two-seat convertible and ask, "Is this what you want?" You say, "No, it must be able to seat four adults." So they leave the prototype with you and go back to work. They build a four-seat convertible and deliver it to you, and you tell them they are getting closer but it still doesn't fit your requirements. They get more information from you, deliver another prototype, get more feedback, and on and on. That back and forth is what is taking place in the circle portion of Figure 24-8.

The main reason that RAD was developed was that by the time software was completely developed following other methodologies, the requirements changed and the developers had to "go back to the drawing board." If a customer needs you to develop a software product and it takes you a year to do so, by the end of that year the customer's needs for the software have probably advanced and changed. The RAD methodology allows for the customer to be involved during the development phases so that the end result maps to their needs in a more realistic manner.

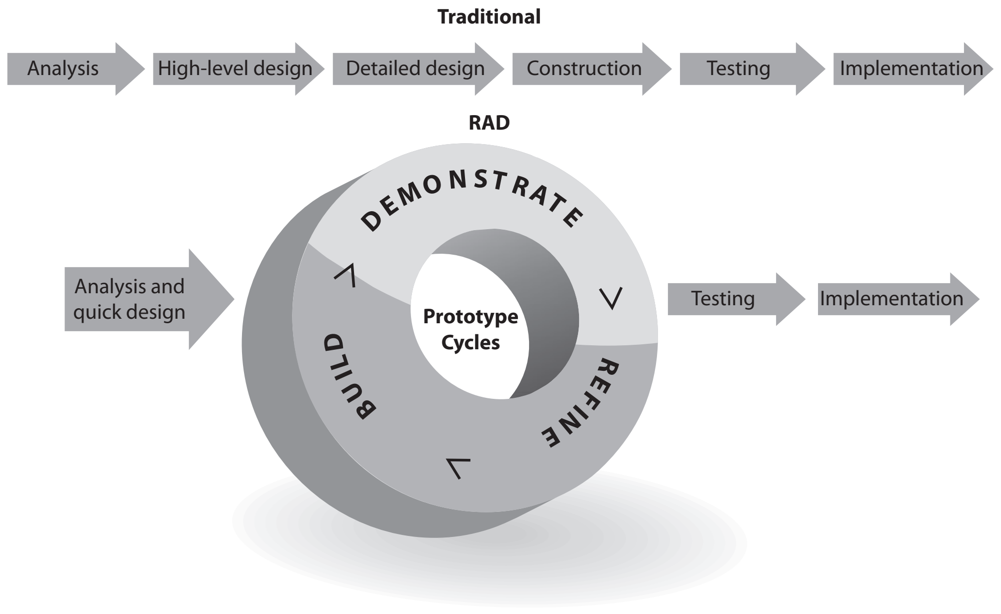


Figure 24-8 Rapid Application Development methodology

Agile Methodologies

The industry seems to be full of software development methodologies, each trying to improve upon the deficiencies of the ones before it. Before the Agile approach to development was created, teams were following rigid process-oriented methodologies. These approaches focused more on following procedures and steps instead of potentially carrying out tasks in a more efficient manner. As an analogy, if you have ever worked within or interacted with a large government agency, you may have come across silly processes that took too long and involved too many steps. If you are a government employee and need to purchase a new chair, you might have to fill out four sets of documents that need to be approved by three other departments. You probably have to identify three different chair vendors, who have to submit a quote, which goes through the contracting office. It might take you a few months to get your new chair. The focus is to follow a protocol and rules instead of efficiency.

Many of the classical software development approaches, as in Waterfall, provide rigid processes to follow that do not allow for much flexibility and adaptability. Commonly, the software development projects that follow these approaches end up failing by not meeting schedule time release, running over budget, and/or not meeting the needs of the customer. Sometimes you need the freedom to modify steps to best meet the situation's needs.

Agile methodology is an umbrella term for several development methodologies. The overarching methodology focuses not on rigid, linear, stepwise processes, but instead on incremental and iterative development methods that promote cross-functional teamwork and continuous feedback mechanisms. This methodology is considered "lightweight" compared to the traditional methodologies that are "heavyweight," which just means this methodology is not confined to a tunnel-visioned and overly structured approach. It is nimble and flexible enough to adapt to each project's needs. The industry found out that even an exhaustive library of defined processes cannot handle every situation that could arise during a development project. So instead of investing time and resources into deep upfront design analysis, the Agile methodology focuses on small increments of functional code that are created based on business need.

The various methodologies under the Agile umbrella focus on individual interaction instead of processes and tools. They emphasize developing the right software product over comprehensive and laborious documentation. They promote customer collaboration instead of contract negotiation, and emphasize abilities to respond to change instead of strictly following a plan.

A notable element of many Agile methodologies is their focus on user stories. A *user story* is a sentence that describes what a user wants to do and why. For instance, a user story could be "As a customer, I want to search for products so that I can buy some." Notice the structure of the story is "As a <user role>, I want to <accomplish some goal> so that <reason for accomplishing the goal>." For example, "As a network analyst, I want to record pcap (packet capture) files so that I can analyze downloaded malware." This method of documenting user requirements is very familiar to the customers and enables their close collaboration with the development team. Furthermore, by keeping this user focus, validation of the features is simpler because the "right system" is described up front by the users in their own words.

EXAM TIP The Agile methodologies do not use prototypes to represent the full product, but break the product down into individual features that are continuously being delivered.

Another important characteristic of the Agile methodologies is that the development team can take pieces and parts of all of the available SDLC methodologies and combine them in a manner that best meets the specific project needs. These various combinations have resulted in many methodologies that fall under the Agile umbrella.

Scrum

Scrum is one of the most widely adopted Agile methodologies in use today. It lends itself to projects of any size and complexity and is very lean and customer focused. Scrum is a methodology that acknowledges the fact that customer needs cannot be completely understood and will change over time. It focuses on team collaboration, customer involvement, and continuous delivery.

The term *scrum* originates from the sport of rugby. Whenever something interrupts play (e.g., a penalty or the ball goes out of bounds) and the game needs to be restarted, all players come together in a tight formation. The ball is then thrown into the middle and the players struggle with each other until one team or the other gains possession of the ball, allowing the game to continue. Extending this analogy, the Scrum methodology allows the project to be reset by allowing product features to be added, changed, or removed at clearly defined points. Since the customer is intimately involved in the development process, there should be no surprises, cost overruns, or schedule delays. This allows a product to be iteratively developed and changed even as it is being built.

The change points happen at the conclusion of each *sprint*, a fixed-duration development interval that is usually (but not always) two weeks in length and promises delivery of a very specific set of features. These features are chosen by the team, but with a lot of input from the customer. There is a process for adding features at any time by inserting them in the feature backlog. However, these features can be considered for actual work only at the beginning of a new sprint. This shields the development team from changes during a sprint, but allows for changes in between sprints.

Extreme Programming

If you take away the regularity of Scrum's sprints and backlogs and add a lot of code reviewing, you get our next Agile methodology. Extreme Programming (XP) is a development methodology that takes code reviews (discussed in Chapter 18) to the extreme (hence the name) by having them take place continuously. These continuous reviews are accomplished using an approach called *pair programming*, in which one programmer dictates the code to her partner, who then types it. While this may seem inefficient, it allows two pairs of eyes to constantly examine the code as it is being typed. It turns out that this approach significantly reduces the incidence of errors and improves the overall quality of the code.

Another characteristic of XP is its reliance on test-driven development, in which the unit tests are written before the code. The programmer first writes a new unit test case, which of course fails because there is no code to satisfy it. The next step is to add just enough code to get the test to pass. Once this is done, the next test is written, which fails, and so on. The consequence is that only the minimal amount of code needed to pass the tests is developed. This extremely minimal approach reduces the incidence of errors because it weeds out complexity.

Kanban

Kanban is a production scheduling system developed by Toyota to more efficiently support just-in-time delivery. Over time, Kanban was adopted by IT and software systems developers. In this context, the *Kanban* development methodology is one that stresses visual tracking of all tasks so that the team knows what to prioritize at what point in time in order to deliver the right features right on time. Kanban projects used to be very noticeable because entire walls in conference rooms would be covered in sticky notes representing the various tasks that the team was tracking. Nowadays, many Kanban teams opt for virtual walls on online systems.

The Kanban wall is usually divided vertically by production phase. Typical columns are labeled Planned, In Progress, and Done. Each sticky note can represent a user story as it moves through the development process, but more importantly, the sticky note can also be some other work that needs to be accomplished. For instance, suppose that one of the user stories is the search feature described earlier in this section. While it is being developed, the team realizes that the searches are very slow. This could result in a task being added to change the underlying data or network architecture or to upgrade hardware. This sticky note then gets added to the Planned column and starts being prioritized and tracked together with the rest of the remaining tasks. This process highlights how Kanban allows the project team to react to changing or unknown requirements, which is a common feature among all Agile methodologies.

DevOps

Traditionally, the software development team and the IT team are two separate (and sometimes antagonistic) groups within an organization. Many problems stem from poor collaboration between these two teams during the development process. It is not rare to have the IT team berating the developers because a feature push causes the IT team to have to stay late or work on a weekend or simply drop everything they were doing in order to "fix" something that the developers "broke." This friction makes a lot of sense when you consider that each team is incentivized by different outcomes. Developers want to push out finished code, usually under strict schedules. The IT staff, on the other hand, wants to keep the IT infrastructure operating effectively. Many project managers who have managed software development efforts will attest to having received complaints from developers that the IT team was being unreasonable and uncooperative, while the IT team was simultaneously complaining about buggy code being tossed over the fence at them at the worst possible times and causing problems on the rest of the network.

A good way to solve this friction is to have both developers and members of the operations staff (hence the term DevOps) on the software development team. *DevOps* is the practice of incorporating development, IT, and quality assurance (QA) staff into software development projects to align their incentives and enable frequent, efficient, and reliable releases of software products. This relationship is illustrated in Figure 24-9.

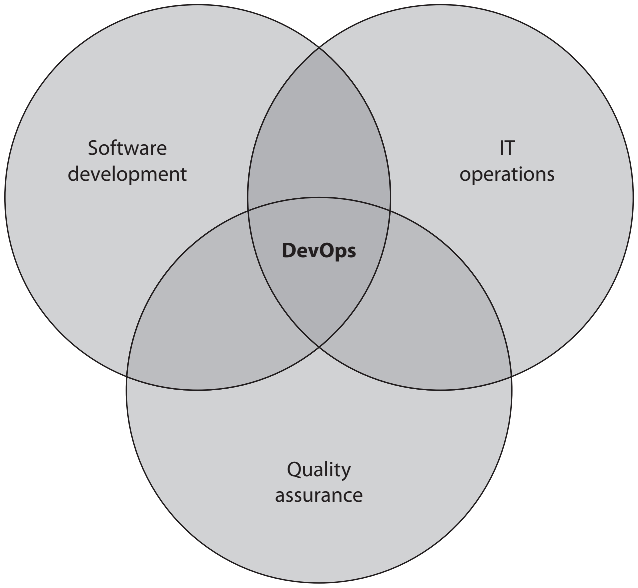


Figure 24-9 DevOps exists at the intersection of software development, IT, and QA.

Ultimately, DevOps is about changing the culture of an organization. It has a huge positive impact on security, because in addition to QA, the IT teammates will be involved at every step of the process. Multifunctional integration allows the team to identify potential defects, vulnerabilities, and friction points early enough to resolve them proactively. This is one of the biggest selling points for DevOps. According to multiple surveys, there are a few other, perhaps more powerful benefits: DevOps increases trust within an organization and increases job satisfaction among developers, IT staff, and QA personnel. Unsurprisingly, it also improves the morale of project managers.

DevSecOps

It is not all that common for the security team to be involved in software development efforts, but it makes a lot of sense for them to be. Their job is to find vulnerabilities before the threat actors can and then do something about it. As a result, most security professionals develop an "adversarial mindset" that allows them to think like attackers in order to better defend against them. Imagine being a software developer and having someone next to you telling you all the ways they could subvert your code to do bad things. It'd be kind of like having a spell-checker but for vulnerabilities instead of spelling!

DevSecOps is the integration of development, security, and operations professionals into a software development team. It's just like DevOps but with security added in. One of the main advantages of DevSecOps is that it bakes security right into the development process, rather than bolting it on at the end of it. Rather than implementing controls to mitigate vulnerabilities, the vulnerabilities are prevented from being implemented in the first place.

Other Methodologies

There seems to be no shortage of SDLC and software development methodologies in the industry. The following is a quick summary of a few others that can also be used:

- **Exploratory methodology** A methodology that is used in instances where clearly defined project objectives have not been presented. Instead of focusing on explicit tasks, the exploratory methodology relies on covering a set of specifications likely to affect the final product's functionality. Testing is an important part of exploratory development, as it ascertains that the current phase of the project is compliant with likely implementation scenarios.
- **Joint Application Development (JAD)** A methodology that uses a team approach in application development in a workshop-oriented environment. This methodology is distinguished by its inclusion of members other than coders in the team. It is common to find executive sponsors, subject matter experts, and end users spending hours or days in collaborative development workshops.

Integrated Product Team

An *integrated product team (IPT)* is a multidisciplinary development team with representatives from many or all the stakeholder populations. The idea makes a lot of sense when you think about it. Why should programmers learn or guess the manner in which the accounting folks handle accounts payable? Why should testers and quality control personnel wait until a product is finished before examining it? Why should the marketing team wait until the project (or at least the prototype) is finished before determining how best to sell it? A comprehensive IPT includes business executives and end users and everyone in between.

The Joint Application Development methodology, in which users join developers during extensive workshops, works well with the IPT approach. IPTs extend this concept by ensuring that the right stakeholders are represented in every phase of the development as formal team members. In addition, whereas JAD is focused on involving the user community, an IPT is typically more inward facing and focuses on bringing in the business stakeholders.

An IPT is not a development methodology. Instead, it is a management technique. When project managers decide to use IPTs, they still have to select a methodology. These days, IPTs are often associated with Agile methodologies.

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

We covered only the most commonly used methodologies in this section, but there are many more that exist. New methodologies have evolved as technology and research have advanced and various weaknesses of older approaches have been addressed. Most of the methodologies exist to meet a specific software development need, and choosing the wrong approach for a certain project could be devastating to its overall success.

EXAM TIP While all the methodologies we covered are used in many organizations around the world, you should focus on Agile, Waterfall, DevOps, and DevSecOps for the CISSP exam.

Review of Development Methodologies

A quick review of the various methodologies we have covered up to this point is provided here:

- **Waterfall** Very rigid, sequential approach that requires each phase to complete before the next one can begin. Difficult to integrate changes. Inflexible methodology.
- **Prototyping** Creating a sample or model of the code for proof-of-concept purposes.
- **Incremental** Multiple development cycles are carried out on a piece of software throughout its development stages. Each phase provides a usable version of software.
- **Spiral** Iterative approach that emphasizes risk analysis per iteration. Allows for customer feedback to be integrated through a flexible evolutionary approach.
- **Rapid Application Development** Combines prototyping and iterative development procedures with the goal of accelerating the software development process.
- **Agile** Iterative and incremental development processes that encourage team-based collaboration. Flexibility and adaptability are used instead of a strict process structure.
- **DevOps** The software development and IT operations teams work together at all stages of the project to ensure a smooth transition from development to production environments.
- **DevSecOps** Just like DevOps, but also integrates the security team into every stage of the project.

Maturity Models

Regardless of which software development methodology an organization adopts, it is helpful to have a framework for determining how well-defined and effective its development activities are. Maturity models identify the important components of software development processes and then organize them in an evolutionary scale that proceeds from ad hoc to mature. Each maturity level comprises a set of goals that, when they are met, stabilize one or more of those components. As an organization moves up this maturity scale, the effectiveness, repeatability, and predictability of its software development processes increase, leading to higher-quality code. Higher-quality code, in turn, means fewer vulnerabilities, which is why we care so deeply about this topic as cybersecurity leaders. Let's take a look at the two most popular models: the Capability Maturity Model Integration (CMMI) and the Software Assurance Maturity Model (SAMM).

Capability Maturity Model Integration

Capability Maturity Model Integration (CMMI) is a comprehensive set of models for developing software. It addresses the different phases of a software development life cycle, including concept definition, requirements analysis, design, development, integration, installation, operations, and maintenance, and what should happen in each phase. It can be used to evaluate security engineering practices and identify ways to improve them. It can also be used by customers in the evaluation process of a software vendor. Ideally, software vendors would use the model to help improve their processes, and customers would use the model to assess the vendors' practices.

EXAM TIP For exam purposes, the terms CMM and CMMI are equivalent.

CMMI describes procedures, principles, and practices that underlie software development process maturity. This model was developed to help software vendors improve their development processes by providing an evolutionary path from an ad hoc "fly by the seat of your pants" approach to a more disciplined and repeatable method that improves software quality, reduces the life cycle of development, provides better project management capabilities, allows for milestones to be created and met in a timely manner, and takes a more proactive approach than the less effective reactive approach. It provides best practices to allow an organization to develop a standardized approach to software development that can be used across many different groups. The goal is to continue to review and improve upon the processes to optimize output, increase capabilities, and provide higher-quality software at a lower cost through the implementation of continuous improvement steps.

If the company Stuff-R-Us wants a software development company, Software-R-Us, to develop an application for it, it can choose to buy into the sales hype about how wonderful Software-R-Us is, or it can ask Software-R-Us whether it has been evaluated against CMMI. Third-party companies evaluate software development companies to certify their product development processes. Many software companies have this evaluation done so they can use this as a selling point to attract new customers and provide confidence for their current customers.

The five maturity levels of CMMI are shown in Figure 24-10 and described here:

- **Level 0: Incomplete** Development process is ad hoc or even chaotic. Tasks are not always completed at all, so projects are regularly cancelled or abandoned.
- **Level 1: Initial** The organization does not use effective management procedures and plans. There is no assurance of consistency, and quality is unpredictable. Success is usually the result of individual heroics.
- **Level 2: Managed** A formal management structure, change control, and quality assurance are in place for individual projects. The organization can properly repeat processes throughout each project.

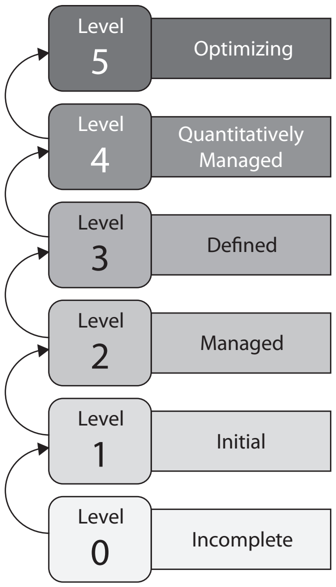


Figure 24-10 CMMI staged maturity levels

- **Level 3: Defined** Formal procedures are in place that outline and define processes carried out in all projects across the organization. This allows the organization to be proactive rather than reactive.
- **Level 4: Quantitatively Managed** The organization has formal processes in place to collect and analyze quantitative data, and metrics are defined and fed into the process-improvement program.
- **Level 5: Optimizing** The organization has budgeted and integrated plans for continuous process improvement, which allow it to quickly respond to opportunities and changes.

Each level builds upon the previous one. For example, a company that accomplishes a Level 5 CMMI rating must meet all the requirements outlined in Levels 1–4 along with the requirements of Level 5.

If a software development vendor is using the Prototyping methodology that was discussed earlier in this chapter, the vendor would most likely only achieve a CMMI Level 1, particularly if its practices are ad hoc, not consistent, and the level of the quality that its software products contain is questionable. If this company practiced a strict Agile SDLC methodology consistently and carried out development, testing, and documentation precisely, it would have a higher chance of obtaining a higher CMMI level.

Capability maturity models (CMMs) are used for many different purposes, software development processes being one of them. They are general models that allow for maturity-level identification and maturity improvement steps. We showed how a CMM can be used for organizational security program improvement processes in Chapter 4.

The software industry ended up with several different CMMs, which led to confusion. CMMI was developed to bring many of these different maturity models together and allow them to be used in one framework. CMMI was developed by industry experts, government entities, and the Software Engineering Institute at Carnegie Mellon University. So CMMI has replaced CMM in the software engineering world, but you may still see CMM referred to within the industry and on the CISSP exam. Their ultimate goals are the same, which is process improvement.

NOTE CMMI is continually being updated and improved upon. You can view the latest documents on it at https://cmmiinstitute.com/learning/appraisals/levels.

Software Assurance Maturity Model

The OWASP *Software Assurance Maturity Model (SAMM)* is specifically focused on secure software development and allows organizations of any size to decide their target maturity levels within each of the five critical business functions: Governance, Design, Implementation, Verification, and Operations, as shown in Figure 24-11. One of the premises on which SAMM is built is that any organization that is involved in software development must perform these five functions.

Each business function, in turn, is divided into three security practices, which are sets of security-related activities that provide assurance for the function. For example, if you want to ensure that your Design business function is done right, you need to perform activities related to threat assessment, identification of security requirements, and securely architecting the software. Each of these 15 practices can be independently

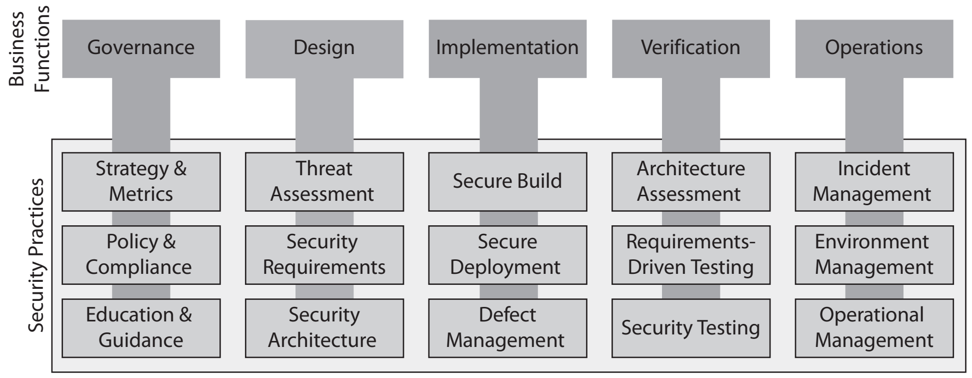


Figure 24-11 Software Assurance Maturity Model

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

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

Chapter Review

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

Quick Review

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

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

Questions

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

1. The software development life cycle has several phases. Which of the following lists these phases in the correct order?
   - **A.** Requirements gathering, design, development, maintenance, testing, release
   - **B.** Requirements gathering, design, development, testing, operations and maintenance
   - **C.** Prototyping, build and fix, increment, test, maintenance
   - **D.** Prototyping, testing, requirements gathering, integration, testing
2. John is a manager of the application development department within his company. He needs to make sure his team is carrying out all of the correct testing types and at the right times of the development stages. Which of the following accurately describe types of software testing that should be carried out?
   - **i.** **Unit testing** Testing individual components in a controlled environment where programmers validate data structure, logic, and boundary conditions
   - **ii.** **Integration testing** Verifying that components work together as outlined in design specifications
   - **iii.** **Acceptance testing** Ensuring that the code meets customer requirements
   - **iv.** **Regression testing** After a change to a system takes place, retesting to ensure functionality, performance, and protection
   - **A.** i, ii
   - **B.** ii, iii
   - **C.** i, ii, iv
   - **D.** i, ii, iii, iv
3. Marge has to choose a software development methodology that her team should follow. The application that her team is responsible for developing is a critical application that can have few to no errors. Which of the following best describes the type of methodology her team should follow?
   - **A.** Cleanroom
   - **B.** Joint Application Development (JAD)
   - **C.** Rapid Application Development (RAD)
   - **D.** Reuse methodology

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

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

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

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

Answers

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

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

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

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

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

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

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

CHAPTER 25

Secure Software

This chapter presents the following:

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

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

—Doug Linder

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

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

Programming Languages and Concepts

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

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

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

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

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

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

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

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

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

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

Language Levels

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

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

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

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

The following lists the basic software programming language generations:

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

Assemblers, Compilers, Interpreters

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

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

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

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

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

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

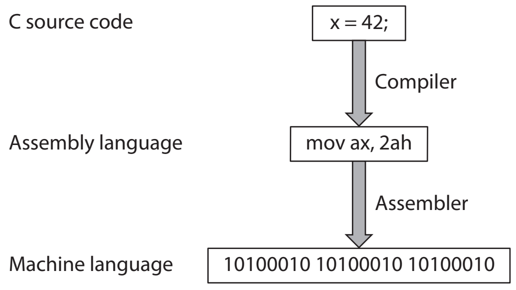


Figure 25-1 Converting a high-level language statement into machine language code

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

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

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

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

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

Runtime Environments

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

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

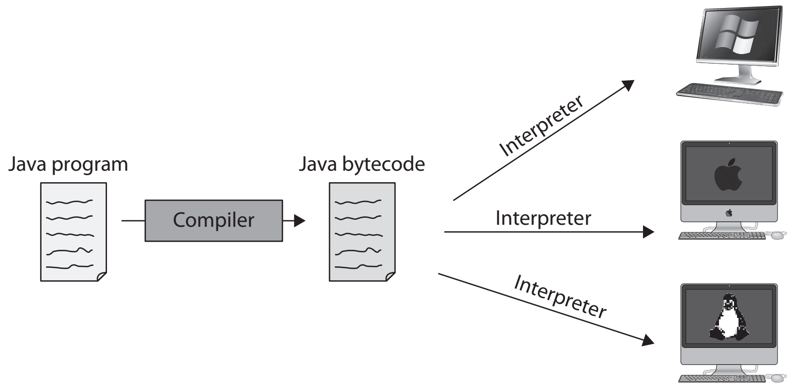


Figure 25-2 The JVM interprets bytecode to machine code for that specific platform.

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

Let's quickly walk through these steps:

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

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

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

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

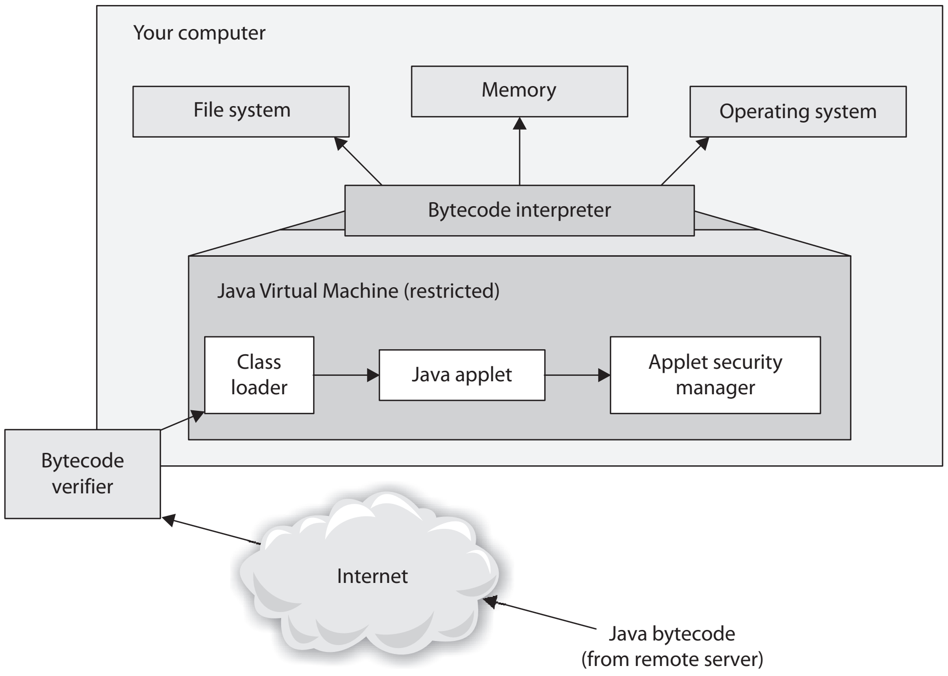
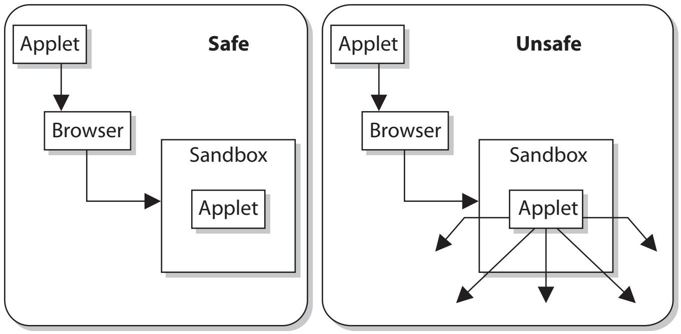


Figure 25-3 Java's security model

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



Object-Oriented Programming Concepts

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

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

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

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

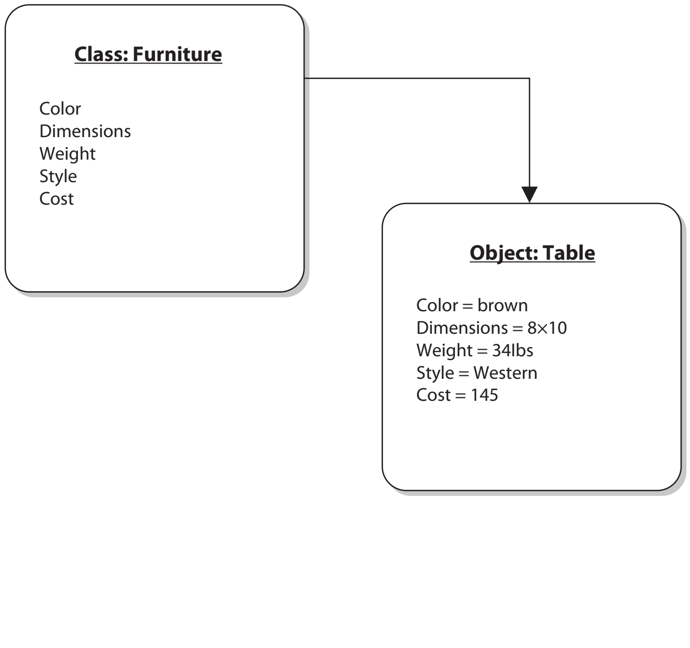
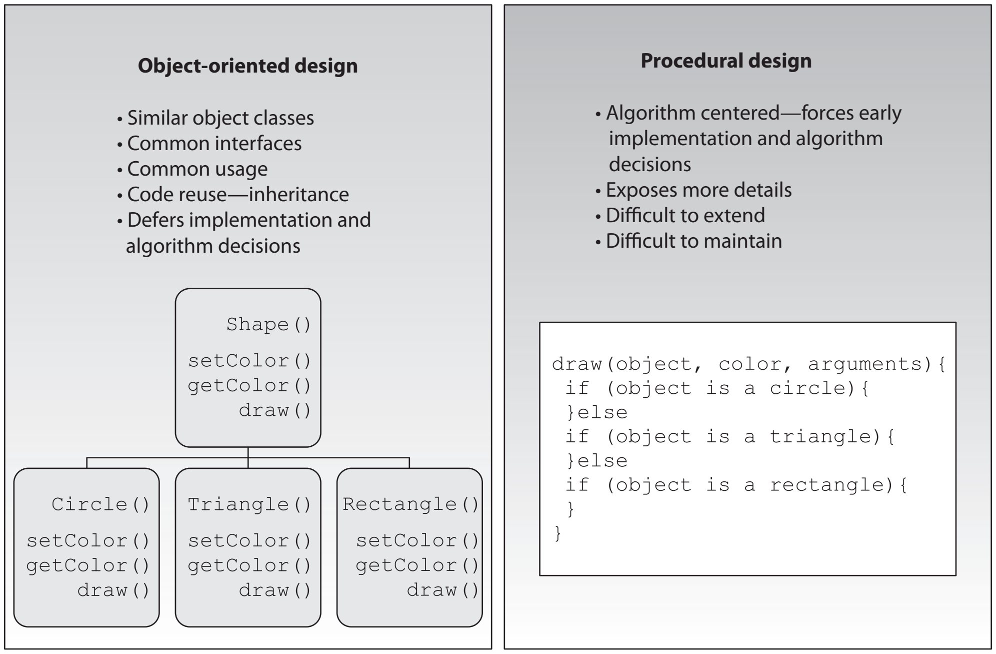


Figure 25-4 In object-oriented inheritance, each object belongs to a class and takes on the attributes of that class

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

TIP An object is an instance of a class.

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


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

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

Figure 25-5 Procedural vs. object-oriented programming

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

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

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

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

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

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

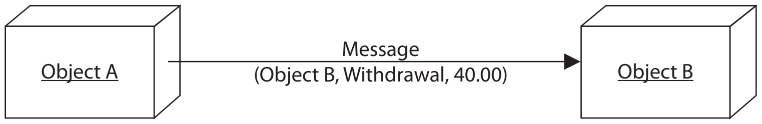


Figure 25-6 Objects communicate via messages.

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

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

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

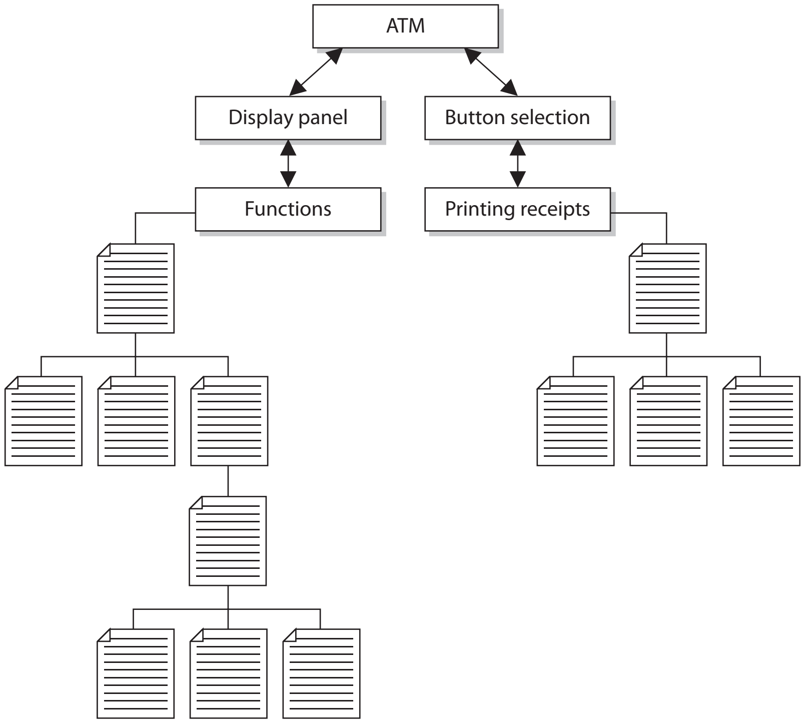


Figure 25-7 Object relationships within a program

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

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

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

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

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

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

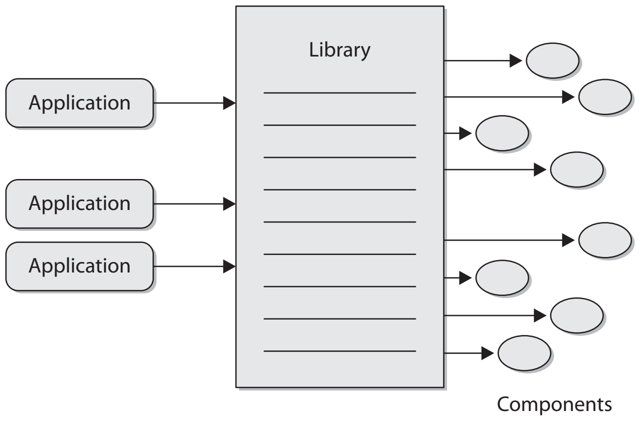


Figure 25-8 Applications locate the necessary objects through a library index.

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

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

Cohesion and Coupling

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

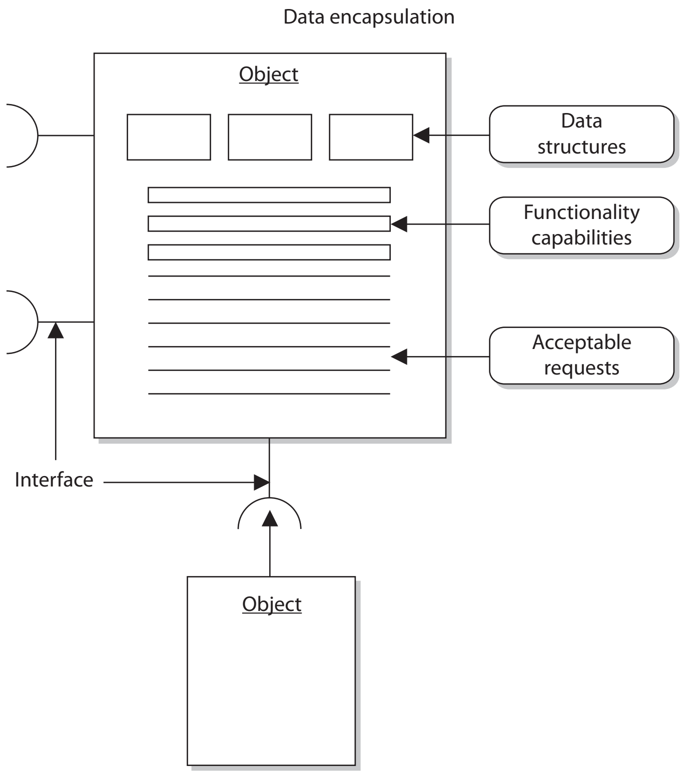


Figure 25-9 The different components of an object and the way it works are hidden from other objects.

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

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

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

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

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

Application Programming Interfaces

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

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

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

Software Libraries

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

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

Secure Software Development

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

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

Source Code Vulnerabilities

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

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

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

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

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

Secure Coding Practices

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

Coding Standards

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

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

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

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

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

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

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

Software-Defined Security

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

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

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

Coding Guidelines

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

Security Controls for Software Development

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

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

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

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

Development Platforms

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

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

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

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

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

Tool Sets

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

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

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

Application Security Testing

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

Static Application Security Testing

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

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

Dynamic Application Security Testing

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

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

Fuzzing

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

Manual Penetration Testing

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

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

Continuous Integration and Delivery

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

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

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

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

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

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

Security Orchestration, Automation, and Response

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

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

Software Configuration Management

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

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

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

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

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

Code Repositories

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

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

Software Escrow

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

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

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

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

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

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

Software Security Assessments

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

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

Risk Analysis and Mitigation

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

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

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

Change Management

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

Assessing the Security of Acquired Software

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

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

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

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

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

Commercial Software

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

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

Open-Source Software

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

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

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

Third-Party Software

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

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

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

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