CHAPTER

APPLICATION MANAGEMENT SECURITY

8

I think computer viruses should count as life. I think it says something about human nature that the only form of life we have created so far is purely destructive. We've created life in our own image.
Stephen Hawking

Security has too often been an afterthought. Organizations have traditionally perceived the threat level to be very low, akin to winning the Powerball lottery or being struck by lightning. Believing the risk to be so low made doing little or nothing seem like a reasonable course of action. Of course, we now know that it was not a wise choice, but many executives do not understand that. Then came one high profile incident after another. Today, the situation is changing (albeit slowly), and there is a new paradigm emerging, one that looks at an organization and says, "Assume that you have been breached or will be breached soon." This shift is leading many organizations to take more stringent security measures. In other cases, regulatory bodies are forcing tighter security on them.

Application security is not the same as securing a server, a router, database, etc., nor is it completely separate from securing those parts of the infrastructure where the **applications** are developed, tested, and run. Applications are different. They enforce security policies, are a trusted source of data, and provide trusted access to data. Applications hold the veritable "keys to the kingdom." Whoever controls or compromises an application has almost unlimited power relative to the domain of that application. Thus, it is imperative that robust security be in place for all of an organization's applications. Broadly, this issue can be divided into two areas: (1) design and develop and (2) deploy and run. This chapter is going to look at measures to protect the applications themselves.

APPLICATION DEVELOPMENT

When it comes to protecting or compromising applications, it is the people who develop and maintain applications who sit at the top of the pecking order. It is their responsibility to design and build applications that are secure. However, the first step in secure application **development** is to create an environment where applications can be developed securely.

What is a secure environment? It is one where reasonable measures are taken to guard against access to the development and test environments by unauthorized parties. Also, in those environments, access to code (whether in development or in production) is limited to individuals who are responsible for development or review of those applications.

In a perfect world, the systems where applications are developed would have no connectivity to anything outside of the secured environment. Unfortunately, in today's world that is not going to happen except for the most extreme cases (e.g., the most sensitive national security applications). Some programmers work from home. Others need to access code in middle of the night when there are problems in production. Also,

Application Performance Management (APM) in the Digital Enterprise. http://dx.doi.org/10.1016/B978-0-12-804018-8.00008-5

it is generally desirable to be able to move application code from the development and test environments to a production environment by transmitting it electronically. Therefore, for these and other reasons, the total isolation of a development and test environment is simply not possible or at least not practical.

The security measures put in place for the design, development, and test environments need to be at least as stringent as, if not greater than, those for the production environments where those applications will run. The security measures are not limited to securing the system (server or mainframe) on which the application development takes place. That also includes any remote systems that may be used for development or to access the development and test environments. It must extend to each of the networks that connect to the application development and test environments. Code that is developed and stored on the application development and test environments must be secured. Generally, that will mean that the code will be encrypted. Similarly, any code that is stored on remote systems should be encrypted while at rest, as should the communications between the remote systems and the application development environment.

As noted earlier, the application developers (programmers) ultimately are the key to securely developing applications and to developing applications that are secure. Therefore, it is of utmost importance that the developers are vetted as much as possible. That last statement may bring a smile to the faces of some readers, for indeed the world of application development is filled with many "unique" individuals. Also, for decades, application development has been a space where there was a shortage of qualified personnel. Therefore, employers cannot always be as selective as they might like to be.

For decades, one of the authors (Sturm) has told audiences around the world that, "When it comes to money, sex, or power, trust no one." Application developers certainly have power and the potential for financial gain through the insertion of malicious code. We will leave it to the reader to decide whether sex is also part of the equation. However, it is clear that unless there are adequate safeguards in place, application developers have both the opportunity to introduce code that they can later exploit and some incentive to do so. This is not intended to suggest that all application developers are dishonest, for clearly not all are. However, in the case of application development, the law of large numbers definitely applies and there will be a few people who might succumb to temptation if they thought that they might not be detected. Therefore, maintaining a secure application development environment with adequate controls is essential.

Another important factor that warrants serious consideration is outsourcing, particularly when application development is outsourced to a company that develops the applications in another country (more so in some countries than others). The client company has control over who is actually writing the code. Also, the client cannot be certain that the development and test environments are adequately secured. Therefore, any code that is developed outside of the company must be analyzed thoroughly to ensure that there are not any backdoors or other weaknesses embedded for someone to exploit in the future. While this is true of applications that are developed in-house, it is much more important with applications that were developed by an outsourcer.

Once a secure environment is established and a team of developers hired, the next step in realizing application security is in the design of the applications. Security must be a priority when designing applications. In fact, security must be a priority throughout the application lifecycle beginning with requirements gathering and continuing through the design, development, testing (**quality assurance**), and deployment phases. It is essential that in the testing phase (perhaps as a separate step in the testing process) the security of the application is tested. It is interesting to note that the actual development phase (i.e., the coding of the application) is not where security is designed. That happens earlier. It is in development that the design is transformed into an executable program. Security for an application is created in the development phase only to the extent that the programmers adhere to the design. However, in the development phase, vulnerabilities can be introduced through errors or by a failure to adhere to the original design.

TOP 25 MOST DANGEROUS SOFTWARE ERRORS

Through a collaborative effort of SANS Institute/MITRE Corporation, many of the leading security experts around the world and with support from the U.S. Department of Homeland Security developed a concise list of the 25 most dangerous software errors, as listed in Tables 8.1, 8.2, and 8.3.

INSECURE INTERACTION BETWEEN COMPONENTS

These weaknesses are related to insecure ways in which data are sent and received between separate components, modules, programs, processes, threads, or systems.

Table 8.1 Interaction Between Components[a]

CWE ID	Name
CWE-89	Improper neutralization of special elements used in an SQL command (“SQL injection”)
CWE-78	Improper neutralization of special elements used in an OS command (“OS command injection”)
CWE-79	Improper neutralization of input during web page generation (“cross-site scripting”)
CWE-434	Unrestricted upload of file with dangerous type
CWE-352	Cross-site request forgery (CSRF)
CWE-601	URL redirection to untrusted site (“open redirect”)

[a]*2011 CWE/SANS “Top 25 Most Dangerous Software Errors” http://cwe.mitre.org/top25/.*

RISKY RESOURCE MANAGEMENT

The weaknesses in this category are related to ways in which software does not properly manage the creation, usage, transfer, or destruction of important system resources.

Table 8.2 Risky Resource Management[a]

CWE ID	Name
CWE-120	Buffer copy without checking size of input (“classic buffer overflow”)
CWE-22	Improper limitation of a pathname to a restricted directory (“path traversal”)
CWE-494	Download of code without integrity check
CWE-829	Inclusion of functionality from untrusted control sphere
CWE-676	Use of potentially dangerous function
CWE-131	Incorrect calculation of buffer size
CWE-134	Uncontrolled format string
CWE-190	Integer overflow or wraparound

[a]*2011 CWE/SANS “Top 25 Most Dangerous Software Errors” http://cwe.mitre.org/top25/.*

POROUS DEFENSES

The weaknesses in this category are related to defensive techniques that are often misused, abused, or just plain ignored.

Table 8.3 Porous Defenses[a]

CWE ID	Name
CWE-306	Missing authentication for critical function
CWE-862	Missing authorization
CWE-798	Use of hard-coded credentials
CWE-311	Missing encryption of sensitive data
CWE-807	Reliance on untrusted inputs in a security decision
CWE-250	Execution with unnecessary privileges
CWE-863	Incorrect authorization
CWE-732	Incorrect permission assignment for critical resource
CWE-327	Use of a broken or risky cryptographic algorithm
CWE-307	Improper restriction of excessive authentication attempts
CWE-759	Use of a one-way hash without a salt

[a] *2011 CWE/SANS "Top 25 Most Dangerous Software Errors" http://cwe.mitre.org/top25/.*

In addition to developing the list of most dangerous software errors, the team also developed a list of mitigations and general practices that can be used to defend against the consequences of those errors. That excellent work is included next and is set apart by being formatted in italics.

2011 CWE/SANS Top 25: Monster Mitigations

These mitigations will be effective in eliminating or reducing the severity of the Top 25. These mitigations will also address many weaknesses that are not even on the Top 25. If you adopt these mitigations, you are well on your way to making more secure software.

Monster Mitigation Index

ID	*Name*
M1	*Establish and maintain control over all of your inputs.*
M2	*Establish and maintain control over all of your outputs.*
M3	*Lock down your environment.*
M4	*Assume that external components can be subverted, and your code can be read by anyone.*
M5	*Use industry-accepted security features instead of inventing your own.*
GP1	*(general) Use libraries and frameworks that make it easier to avoid introducing weaknesses.*
GP2	*(general) Integrate security into the entire software development lifecycle.*
GP3	*(general) Use a broad mix of methods to comprehensively find and prevent weaknesses.*
GP4	*(general) Allow locked-down clients to interact with your software.*

Monster Mitigation Matrix

The following table maps CWEs to the recommended monster mitigations, along with a brief summary of the mitigation's effectiveness.

Effectiveness ratings include:

- ***High**: The mitigation has well-known, well-understood strengths and limitations; there is good coverage with respect to variations of the weakness.*
- ***Moderate**: The mitigation will prevent the weakness in multiple forms, but it does not have complete coverage of the weakness.*
- ***Limited**: The mitigation may be useful in limited circumstances, only be applicable to a subset of this weakness type, require extensive training/customization, or give limited visibility.*
- ***Defense in Depth (DiD)**: The mitigation may not necessarily prevent the weakness, but it may help to minimize the potential impact when an attacker exploits the weakness.*

Within the matrix, the following mitigations are identified:

- *M1: Establish and maintain control over all of your inputs.*
- *M2: Establish and maintain control over all of your outputs.*
- *M3: Lock down your environment.*
- *M4: Assume that external components can be subverted, and your code can be read by anyone.*
- *M5: Use industry-accepted security features instead of inventing your own.*

The following general practices are omitted from the matrix:

- *GP1: Use libraries and frameworks that make it easier to avoid introducing weaknesses.*
- *GP2: Integrate security into the entire software development lifecycle.*
- *GP3: Use a broad mix of methods to comprehensively find and prevent weaknesses.*
- *GP4: Allow locked-down clients to interact with your software.*

M1	*M2*	*M3*	*M4*	*M5*	*CWE*
High		*DiD*	*Mod*		*CWE-22: Improper Limitation of a Pathname to a Restricted Directory ("Path Traversal")*
Mod	*High*	*DiD*	*Ltd*		*CWE-78: Improper Neutralization of Special Elements used in an OS Command ("OS Command Injection")*
Mod	*High*		*Ltd*		*CWE-79: Improper Neutralization of Input During Web Page Generation ("Cross-site Scripting")*
Mod	*High*	*DiD*	*Ltd*		*CWE-89: Improper Neutralization of Special Elements used in an SQL Command ("SQL Injection")*
Mod		*DiD*	*Ltd*		*CWE-120: Buffer Copy without Checking Size of Input ("Classic Buffer Overflow")*
Mod		*DiD*	*Ltd*		*CWE-131: Incorrect Calculation of Buffer Size*
High		*DiD*	*Mod*		*CWE-134: Uncontrolled Format String*
Mod		*DiD*	*Ltd*		*CWE-190: Integer Overflow or Wraparound*
		High			*CWE-250: Execution with Unnecessary Privileges*
		Mod		*Mod*	*CWE-306: Missing Authentication for Critical Function*
				Mod	*CWE-307: Improper Restriction of Excessive Authentication Attempts*
		DiD			*CWE-311: Missing Encryption of Sensitive Data*
				High	*CWE-327: Use of a Broken or Risky Cryptographic Algorithm*
			Ltd		*CWE-352: Cross-Site Request Forgery (CSRF)*
Mod		*DiD*	*Mod*		*CWE-434: Unrestricted Upload of File with Dangerous Type*
		DiD			*CWE-494: Download of Code Without Integrity Check*

Continued

M1	*M2*	*M3*	*M4*	*M5*	*CWE*
Mod	*Mod*		*Ltd*		*CWE-601: URL Redirection to Untrusted Site ("Open Redirect")*
Mod	*High*	*DiD*			*CWE-676: Use of Potentially Dangerous Function*
	Ltd	*DiD*		*Mod*	*CWE-732: Incorrect Permission Assignment for Critical Resource*
				High	*CWE-759: Use of a One-Way Hash without a Salt*
		DiD	*High*	*Mod*	*CWE-798: Use of Hard-coded Credentials*
Mod		*DiD*	*Mod*	*Mod*	*CWE-807: Reliance on Untrusted Inputs in a Security Decision*
High		*High*	*High*		*CWE-829: Inclusion of Functionality from Untrusted Control Sphere*
DiD		*Mod*		*Mod*	*CWE-862: Missing Authorization*
		DiD		*Mod*	*CWE-863: Incorrect Authorization*

Mitigation Details
M1: Establish and maintain control over all of your inputs.
Target Audience: *Programmers.*
Associated CWEs:

CWE-20 Improper Input Validation

Improper input validation is the number one killer of healthy software, so you're just asking for trouble if you don't ensure that your input conforms to expectations. For example, an identifier that you expect to be numeric shouldn't ever contain letters. Nor should the price of a new car be allowed to be a dollar, not even in today's economy. Of course, applications often have more complex validation requirements than these simple examples. Incorrect input validation can lead to vulnerabilities when attackers can modify their inputs in unexpected ways. Many of today's most common vulnerabilities can be eliminated, or at least reduced, using proper input validation.

Use a standard input validation mechanism to validate all input for:

- *length*
- *type of input*
- *syntax*
- *missing or extra inputs*
- *consistency across related fields*
- *business rules*

As an example of business rule logic, "boat" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if you are expecting color names such as "red" or "blue."

Where possible, use stringent whitelists that limit the character set based on the expected value of the parameter in the request. This can have indirect benefits, such as reducing or eliminating weaknesses that may exist elsewhere in the product.

Do not accept any inputs that violate these rules, or convert the inputs to safe values.

Understand all the potential areas where untrusted inputs can enter your software: parameters or arguments, cookies, anything read from the network, environment variables, reverse DNS lookups, query results, request headers, URL components, e-mail, files, databases, and any external systems that provide data to the application. Remember that such inputs may be obtained indirectly through API calls.

Be careful to properly decode the inputs and convert them to your internal representation before performing validation.

Applicable Top 25 CWEs:
CWE-120, CWE-129, CWE-131, CWE-134, CWE-190, CWE-209, CWE-22, CWE-285, CWE-306, CWE-311, CWE-327, CWE-352, CWE-362, CWE-434, CWE-494, CWE-601, CWE-676, CWE-732, CWE-754, CWE-770, CWE-78, CWE-79, CWE-798, CWE-805, CWE-807, CWE-829, CWE-862, CWE-89, CWE-98.

M2: Establish and maintain control over all of your outputs.
Target Audience: *Programmers, Designers*

Associated CWEs:

CWE-116 Improper Encoding or Escaping of Output

Computers have a strange habit of doing what you say, not what you mean. Insufficient output encoding is the often-ignored sibling to improper input validation, but it is at the root of most injection-based attacks, which are all the rage these days. An attacker can modify the commands that you intend to send to other components, possibly leading to a complete compromise of your application - not to mention exposing the other components to exploits that the attacker would not be able to launch directly. This turns "do what I mean" into "do what the attacker says." When your program generates outputs to other components in the form of structured messages such as queries or requests, it needs to separate control information and metadata from the actual data. This is easy to forget, because many paradigms carry data and commands bundled together in the same stream, with only a few special characters enforcing the boundaries. An example is Web 2.0 and other frameworks that work by blurring these lines. This further exposes them to attack.

If available, use structured mechanisms that automatically enforce the separation between data and code. These mechanisms may be able to provide the relevant quoting, encoding, and validation automatically, instead of relying on the developer to provide this capability at every point where output is generated.

For example, stored procedures can enforce database query structure and reduce the likelihood of SQL injection.

Understand the context in which your data will be used and the encoding that will be expected. This is especially important when transmitting data between different components, or when generating outputs that can contain multiple encodings at the same time, such as web pages or multi-part mail messages. Study all expected communication protocols and data representations to determine the required encoding strategies.

Applicable Top 25 CWEs:

CWE-120, CWE-129, CWE-131, CWE-190, CWE-209, CWE-22, CWE-285, CWE-306, CWE-311, CWE-327, CWE-352, CWE-362, CWE-434, CWE-494, CWE-601, CWE-676, CWE-732, CWE-754, CWE-770, CWE-78, CWE-79, CWE-798, CWE-805, CWE-807, CWE-89, CWE-98.

M3: Lock down your environment.

Target Audience: *Project Managers, Designers*

Associated CWEs:

CWE-250 Execution with Unnecessary Privileges

It's an unfortunate fact of life: weaknesses happen, even with your best efforts. By using defense in depth, you can limit the scope and severity of a successful attack against an unexpected weakness, sometimes even reducing the problems to an occasional annoyance. The core aspect of defense in depth is to avoid dependence on a single feature, solution, mitigation or barrier (e.g., firewall) to provide all the security. Locking down your environment helps to achieve this.

Run your software in a restricted environment, using available security features to limit the scope of what your software can do to its host systems and adjacent networks. Even if an attacker manages to find and exploit a vulnerability, an environmental lockdown can limit what the attacker can do, reducing the overall severity of the problem and making the attacker work harder. In some cases, the lockdown may prevent the problem from becoming a vulnerability in the first place.

Some of the aspects of environment lockdown are:

- *Run as an unprivileged user. Run your software with the lowest privileges possible. Even if your software is compromised via a weakness that you were not aware of, it will make it more difficult for an attacker to cause further damage. Spider Man, the well-known comic superhero, lives by the motto "With great power comes great responsibility." Your software may need special privileges to perform certain operations, but wielding those privileges longer than necessary can be extremely risky. When running with extra privileges, your application has access to resources that the application's user can't directly reach. For example, you might intentionally launch a separate program, and that program allows its user to specify a file to open; this feature is frequently present in help utilities or editors. The user can access unauthorized files through the launched program, thanks to those extra privileges. Command execution can happen in a similar fashion. Even if you don't launch other programs, additional vulnerabilities in your software could have more serious consequences than if it were running at a lower privilege level.*
- *Consider disabling verbose error messages for messages that are displayed to users without special privileges. If you use chatty error messages, then they could disclose secrets to any attacker who dares to misuse your software. The secrets could cover a wide range of valuable data, including personally identifiable information (PII), authentication credentials, and server configuration. Sometimes, they might seem like harmless secrets that are convenient for your*

Continued

users and admins, such as the full installation path of your software. Even these little secrets can greatly simplify a more concerted attack that yields much bigger rewards, which is done in real-world attacks all the time. This is a concern whether you send temporary error messages back to the user or if you permanently record them in a log file.

- *Wherever possible, if you must third-party libraries, use best-of-breed libraries with an established record of good security.*
- *Build or compile your code using security-related options. For example, some compilers provide built-in protection against buffer overflows.*
- *Use operating system and hardware features, when available, that limit the impact of a successful attack. For example, some chips, OS features, and libraries make it more difficult to exploit buffer overflows. Some OS configurations allow you to specify quotas for disk usage, CPU, memory, open files, and others.*
- *Use virtualization, sandboxes, jails, or similar technologies that limit access to the environment. Even if an attacker completely compromises your software, the damage to the rest of the environment may be minimized. For example, an application may intend to write to files within a limited subdirectory, but path traversal weaknesses typically allow an attacker to navigate outside of the directory and through the entire file system.*
- *Consider adopting secure configurations using security benchmarks or hardening guides such as the Federal Desktop Core Configuration (FDCC).*
- *Use vulnerability scanners and regularly apply security patches to ensure that your system does not have any known vulnerabilities. Even if your application does not have any obvious problems, an attacker may be able to compromise the application by attacking the system itself.*

Applicable Top 25 CWEs:
CWE-120, CWE-129, CWE-131, CWE-134, CWE-190, CWE-209, CWE-22, CWE-250, CWE-285, CWE-306, CWE-311, CWE-327, CWE-352, CWE-362, CWE-434, CWE-494, CWE-601, CWE-676, CWE-732, CWE-754, CWE-770, CWE-78, CWE-79, CWE-798, CWE-805, CWE-807, CWE-829, CWE-862, CWE-863, CWE-89, CWE-98.

M4: Assume that external components can be subverted, and your code can be read by anyone.
Target Audience: *Project Managers, Programmers, Designers*
Associated CWEs:
CWE-602 Client-Side Enforcement of Server-Side Security
CWE-649 Reliance on Obfuscation or Encryption of Security-Relevant Inputs without Integrity Checking
CWE-656 Reliance on Security Through Obscurity
If it's out of your hands, it's out of your control.

When it comes to security, your code operates in a hostile environment. Attackers can analyze your software - even if it's obfuscated - and figure out what's going on. They can modify external components to ignore security controls. They can try any type of attack against every component in the system. If your code is distributed to your consumers, even better - they can run your software and attack it in their own environment, forever, without you ever knowing about it.

Many developers assume that their compiled code, or custom protocols, cannot be figured out by attackers. However, modern reverse engineering techniques are very mature, and hackers can figure out the inner workings of software quickly, even with proprietary protocols, lack of access to specifications, and no source code. When software's protection mechanisms rely on this secrecy for protection, those mechanisms can often be defeated. You might try to encrypt your own code, but even so, it's a stalling tactic in most environments.

You also can't trust a client to perform security checks on behalf of your server, even if you wrote the client yourself. Remember that underneath that fancy GUI, it's just code. Attackers can reverse engineer your client and write their own custom clients that leave out certain inconvenient features like all those pesky security controls. The consequences will vary depending on what your security checks are protecting, but some of the more common targets are authentication, authorization, and input validation. If you've implemented security in your servers, then you need to make sure that you're not solely relying on the clients to enforce that security.

Applicable Top 25 CWEs:
CWE-120, CWE-129, CWE-131, CWE-134, CWE-190, CWE-209, CWE-22, CWE-285, CWE-306, CWE-311, CWE-327, CWE-352, CWE-362, CWE-434, CWE-494, CWE-601, CWE-732, CWE-754, CWE-770, CWE-78, CWE-79, CWE-798, CWE-805, CWE-807, CWE-829, CWE-89, CWE-98.

M5: Use industry-accepted security features instead of inventing your own.
Target Audience: *Project Managers, Designers, Programmers*
Associated CWEs:

CWE-693 Protection Mechanism Failure.

You might be tempted to create your own security algorithms, but seriously, this is tough to do correctly. You might think you created a brand-new algorithm that nobody will figure out, but it's more likely that you're reinventing a wheel that falls off just before the parade is about to start.

This applies to features including cryptography, authentication, authorization, randomness, session management, logging, and others.

- *Investigate which of the security algorithms available to you is the strongest for cryptography, authentication, and authorization, and use it by default. Use well-vetted, industry-standard algorithms that are currently considered to be strong by experts in the field, and select well-tested implementations. Stay away from proprietary and secret algorithms. Note that the number of key bits often isn't a reliable indication; when in doubt, seek advice.*
- *Use languages, libraries, or frameworks that make it easier to use these features.*
- *When you use industry-approved techniques, you need to use them correctly. Don't cut corners by skipping resource-intensive steps (CWE-325). These steps are often essential for preventing common attacks.*

Applicable Top 25 CWEs:
CWE-120, CWE-129, CWE-131, CWE-190, CWE-209, CWE-22, CWE-285, CWE-306, CWE-307, CWE-311, CWE-327, CWE-352, CWE-362, CWE-434, CWE-494, CWE-601, CWE-732, CWE-754, CWE-759, CWE-770, CWE-78, CWE-79, CWE-798, CWE-805, CWE-807, CWE-862, CWE-863, CWE-89, CWE-98.

GP1: Use libraries and frameworks that make it easier to avoid introducing weaknesses.
Target Audience: *Project Managers, Designers, Programmers*
In recent years, various frameworks and libraries have been developed that make it easier for you to introduce security features and avoid accidentally introducing weaknesses. Use of solid, security-relevant libraries may save development effort and establish a well-understood level of security. If a security problem is found, the fix might be local to the library, instead of requiring wholesale changes throughout the code.

Ultimately, you may choose to develop your own custom frameworks or libraries; but in the meantime, there are some resources available to give you a head start. Options range from safe string handling libraries for C/C++ code, parameterized query mechanisms such as those available for SQL, input validation frameworks such as Struts, API schemes such as ESAPI, and so on.

While it is often advocated that you could choose a safer language, this is rarely feasible in the middle of an ongoing project. In addition, each language has features that could be misused in ways that introduce weaknesses. Finally, as seen in the "Weaknesses by Language" focus profile, very few weaknesses are language-specific - i.e., most weaknesses apply to a broad array of languages.

GP2: Integrate security into the entire software development lifecycle.
Target Audience: *Project Managers*
The Top 25 is only the first step on the road to more secure software. Weaknesses and vulnerabilities can only be avoided when secure development practices are integrated into all phases of the software lifecycle, including (but not limited to) requirements and specification, design and architecture, implementation, build, testing, deployment, operation, and maintenance. Creating a software security group (SSG) may be the most critical step.

Projects such as BSIMM, SAFEcode, and OpenSAMM can help you understand what leading organizations are doing for secure development.

GP3: Use a broad mix of methods to comprehensively find and prevent weaknesses.
Target Audience: *Project Managers*
There is no single tool, technique, or process that will guarantee that your software is secure. Each approach has its own strengths and limitations, so an appropriate combination will improve your security posture beyond any single technique.

Continued

Approaches include, but are not limited to:

- *Automated static code analysis: whether for source code or binary code. Modern automated static code analysis can provide extensive code coverage, with high hit rates for certain types of weaknesses, especially for implementation errors. However, in some contexts, these techniques can report a large number of issues that a developer may not wish to address. They also have difficulty with various design-level problems, which often require human analysis to recognize.*
- *Manual static code analysis: Can be useful for finding subtle errors and design flaws, and for giving an overall assessment of development practices. Useful for detecting business logic flaws. However, it can be expensive to conduct, code coverage is a challenge, and it may not catch all attack vectors.*
- *Automated dynamic code analysis: fuzzers, scanners. These can find some issues that are difficult to detect with static analysis, such as environmental problems, and generate a large number of inputs or interactions that can find subtle flaws that may be missed by static analysis. However, code coverage is a problem.*
- *Manual dynamic code analysis ("pen testing," etc.): This can be effective for identifying flaws in business logic, for quickly finding certain types of issues that are not easy to automate, and for understanding the overall security posture of the software.*
- *Threat modeling: these are useful for finding problems before any code is developed. While techniques are still under development, currently this approach is difficult to teach, available tools have limited power, and it benefits significantly from expert participation.*
- *Application firewalls and external monitoring/control frameworks (web application firewalls, proxies, IPS, Struts, etc.): using application firewalls and similar mechanisms may be appropriate for protecting software that is known to contain weaknesses, but cannot be modified. They may also be useful for preventing attacks against latent weaknesses that have not yet been discovered yet. However, deployment could affect legitimate functionality, customization may be needed, and not all attacks can be automatically detected or prevented.*
- *Education and training: Appropriate training will help programmers avoid introducing errors into code in the first place, reducing downstream costs. However, training itself may be expensive, face cultural challenges for adoption, and require updates as the software and detection methods evolve.*
- *Architecture and design reviews: these can be important for detecting problems that would be too difficult, time-consuming, or expensive to fix after the product has been deployed. They may require expert-level effort.*
- *Coding standards: following appropriate coding standards can reduce the number of weaknesses that are introduced into code, which may save on maintenance costs. However, these are difficult to integrate into existing projects, especially large ones.*

Additional methods and practices are provided in the Software Security Framework (SSF), which is associated with BSIMM.

GP4: Allow locked-down clients to interact with your software.
Target Audience: *Project Managers, Designers, Programmers*
If a user has a locked-down client that does not support all the features used by your application, consider allowing that user some access, instead of completely denying access, which may force the user to degrade the client's security stance. This may help protect users and slow the propagation of malware. For example, in a web environment, some users may have Javascript or browser plug-ins disabled, and requiring these features exposes them to attacks from other web sites.[1]

[1]2011 CWE/SANS "Top 25: Monster Mitigations" http://cwe.mitre.org/top25/mitigations.html.

SECURING APPLICATIONS IN PRODUCTION

The key to securing applications in a production environment lies in implementing a thorough, unified, multilayered approach to security for the environment [often referred to as "**defense in depth**" (DiD)]. That begins with making the production environment physically secure. Physical security is achieved by placing servers in a protected computer room with controlled access.

The next step is to establish **perimeter security**. That begins with putting in place **firewalls** for the production environment. Also, personal firewalls should be installed in client computers that are used

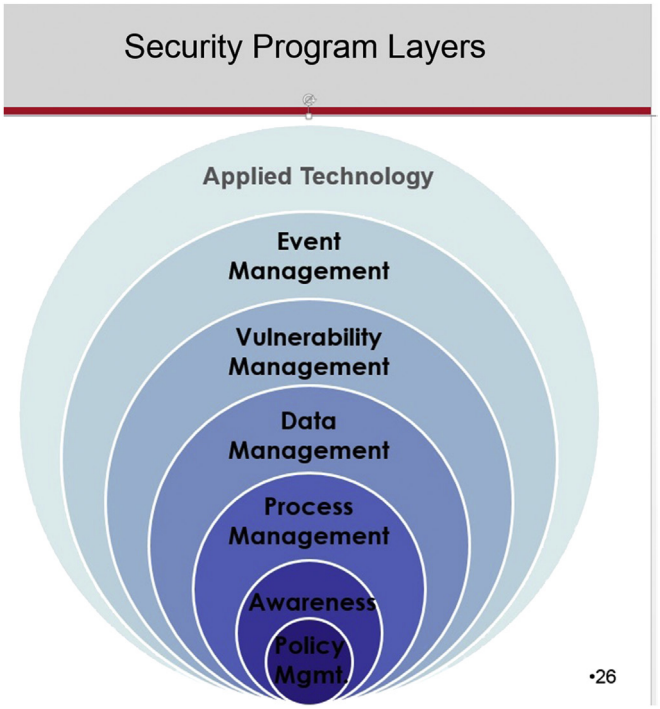

FIGURE 8.1

Defense in depth (DiD).

outside of the organization's secure facilities. Regularly scan for intrusion using **intrusion detection (IDS)**, **intrusion prevention (IPS)**, or **next generation firewalls (NGFWs)** that include such functionality. Other techniques might also be used (e.g., SSL and SSH interception, website filtering, **QoS/** bandwidth management, and antivirus inspection) (Fig. 8.1).

The development, test, and production environments must be in sync in terms of **operating system** version levels, patch levels, etc. This seems obvious but every year there are numerous breaches that result because the production environment is not at the same level as the other environments.

Additional measures that can be taken to ensure the security of applications in the production environment include access control, enforcement of strong and regularly changed **passwords**, **encryption** of application code when at rest, and of data and of network traffic.

Characteristics of very strong passwords:
Passwords must be 15 characters and contain at least three of the following four items:
- uppercase letter
- lowercase letter
- number
- special character

Expire after 30–60 days.
Passwords cannot be reused for at least 1 year.
Cannot contain any information that can be associated with the user (user's name, name of family members, dates, etc.)
Cannot contain easily guessable passwords (e.g, "password," "1234567890," etc.)

SUMMARY

Securing applications throughout their lifecycle (development, test, production) is an essential part of **application management**. In technical terms, it is not particularly difficult to do (at least within the limits of available technology). However, successfully securing applications is highly dependent on organization and personal discipline in formulating and enforcing the policies that make that security possible.

KEY TAKEAWAYS

- Application security begins the design and coding of applications that incorporate security best practices and avoid common coding errors that can create exploitable vulnerabilities.
- There are 25 common coding errors that are the most egregious.
- There are steps that can be taken to eliminate or minimize the seriousness of those common errors.
- There are general practices that can be put in place to improve security.
- The production environment, the test environment, and the production environments must be synchronized.
- The best way to secure applications is through a unified, multilayered approach that is described as DiD.