(the company's CSO) received quite a few apologies. Immediately afterwards, he began lobbying for stronger change controls and clearer separations of duties.

Three key metrics can help organizations understand the degree of change control an organization possesses. All of these assume the organization keeps track of changes to production systems:

- The number of production changes
- The number of exemptions
- The number of unauthorized changes (violations)

These metrics are typically tracked on a per-period basis, and for additional insights they can be sliced by business unit or technology area. The latter two, exemptions and violations, are related. An "exemption" represents a change that was granted for exceptional reasons and required implementation outside of normal maintenance hours. Emergency fixes and other out-of-cycle changes require exemptions.

An interesting variation on the exemption metric is one that divides the number of exemptions into the number of changes. This results in a percentage that shows how many changes are made out-of-cycle. When grouped by business unit, this metric provides evidence that helps IT organizations finger the twitchiest and most cowboy-like business units.

Unauthorized changes, also known as violations, measure the number of changes that were applied without approval. This number, obviously, should be as close to zero as possible.

APPLICATION SECURITY

Applications are the electronic engines that drive most businesses. Microsoft Office, web servers, order-management software, supply chain management, and ERP systems are all applications that businesses rely on every day. Applications automate firms' workforce activities, pay their bills, and serve customers. Applications come in many shapes and sizes: in-house developed, packaged, outsourced, and served on demand.

As important as applications are to the fortunes of most organizations, they also represent points of potential weakness. Application threat vectors, although they are less well understood than network-based threats, are just as important. As long ago as 2002, Garter Group stated that 75 percent of attacks tunneled through or used application-related threat vectors.[27]

[27] D. Verton, "Airline Web Sites Seen as Riddled with Security Holes," *Computerworld*, 4 Feb. 2002.

For companies with custom-developed applications, the manner in which the software was developed matters. Software written without sufficient attention to security issues carries much more risk than software that adheres to generally accepted principles for coding secure software—as much as five times more risk, based on my previous research.[28]

Measuring the relative security of application code is hard. The security industry has not arrived at a consensus about exactly what it means to build a "secure application." Although definitions vary, there are at least three potential ways to measure application security (see Table 3-5): by counting remotely and locally exploitable flaws without knowledge of the code (black-box metrics), by counting design and implementation flaws in the code (code security metrics), and by creating qualitative risk indices using a weighted scoring system (qualitative process metrics and indices).

Table 3-5 Application Security Metrics

Metric	Purpose	Sources
Black-Box Defect Metrics		
Defect counting	Shows externally identified defects due to implementation or design flaws	Black-box testing tools
Vulnerabilities per application (number [#]) • By business unit • By criticality • By proximity	Measures the number of vulnerabilities that a potential attacker without prior knowledge might find	Black-box assessments by security consultants
Qualitative Process Metrics and Indices		
Business-adjusted risk	Simple formula for scoring the business impact and criticality of vulnerabilities identified in security assessments	Security assessments Spreadsheets
Application conformance indices	Creates a score for ranking the overall security posture for an application or group of applications	Questionnaires Spreadsheets

[28] A. Jaquith, "The Security of Applications: Not All Are Created Equal," @stake, Inc., 2002.

Metric	Purpose	Sources
Code Security Metrics		
Assessment frequency for developed applications • % with design reviews • % with application assessments • % with code reviews (optional) of sensitive functions • % with go-live penetration tests	Measures how often security quality assurance "gates" are applied to the software development life cycle for custom-developed applications.	Manual tracking Lines of code (LOC)
Thousand lines of code (KLOC)	Shows the aggregate size of a developed application	Code analysis software
Defects per KLOC	Characterizes the incidence rate of security defects in developed code	Code analysis software
Vulnerability density (vulnerabilities per unit of code)	Characterizes the incidence rate of security defects in developed code	Code analysis software
Known vulnerability density (weighted sum of all known vulnerabilities per unit of code)	Characterizes the incidence rate of security defects in developed code, taking into account the seriousness of flaws	Code analysis software
Tool soundness	Estimates the degree of error intrinsic to code analysis tools	Code analysis software Spreadsheets
Cyclomatic complexity	Shows the relative complexity of developed code. Indicates potential maintainability issues and security trouble spots.	Code analysis software

BLACK-BOX DEFECT METRICS

Perhaps the most dramatic and headline-grabbing type of application security metric is of the black-box variety—that is, how many holes we can drill in one application compared to another. Black-box testing involves assessing an application, typically remotely via the web. The method of assessment varies. For high-volume testing, automated black-box testing tools from SPI Dynamics, Cenzic, and Watchfire allow companies (or consultants) to quickly scan a large number of deployed applications for potential vulnerabilities.

Automated tools are best suited to testing web applications. A typical black-box web security testing tool "spiders" an application by starting at a known URL (http://www.foo.com/myapp) and following every related hyperlink until it has discovered all the website's pages. After the spider enumerates the application's pages, an automated "fuzzer" or "fault injector" examines the web forms on each page, looking for weaknesses. For example, the fuzzer might see an account registration form that contains a field into which a new user is meant to type her first name. The goal of the fault injector is to see what happens when it sets the field value to something the server-side logic won't expect—like 10,000 letter A's, SQL statements, or shell code.

In the nonautomated camp are security consultants who conduct tests as part of a formally scoped engagement. Consultants tend to be much more expensive than an automated black-box tool but can find issues that the tools cannot. They can also exercise their creativity to dig deeper and find root causes. On the other hand, the level of analytical rigor and degree of methodological consistency vary from consultant to consultant.

Regardless of the method used, the objective of testing is the same: to find vulnerabilities (defects) that can be exploited to compromise the application's integrity, confidentiality, or availability. The categories of flaws that black-box tools and consultants tend to find include:

- **SQL injection:** Manipulating submitted web form fields to trick databases into disgorging sensitive information
- **Command injection:** Executing native operating system commands on the web server
- **Parameter tampering:** Changing submitted web form fields to change the application state
- **Cross-site scripting:** Submitting malformed input that will cause subsequent users to execute malicious JavaScript commands that hijack their sessions or capture data
- **Buffer overflows:** Overfilling a server-side buffer in an effort to make the server crash, or to take it over remotely

At the end of the assessment, the tool or consultant adds up and summarizes any defects found for prioritization. Results of black-box tests are typically simple counts of what defects were found, the category to which they belong, and where. Security consultants generally, as part of the engagement, prioritize the vulnerabilities they find and assign them a criticality rating (high, medium, low).

Enterprises that rely on black-box testing techniques to provide application security metrics, in my experience, do not care too much about defects that aren't marked as

critical. They *do* fix severe issues that could lead to a remote compromise or disclose sensitive data. Thus, in Table 3-5 we recommend that organizations group vulnerabilities by criticality. Other cross sections that companies find useful include by business unit and by proximity. Was the defect remotely exploitable, or could the exploit succeed only when the attacker was logged in locally to the server?

QUALITATIVE PROCESS METRICS AND INDICES

Qualitative assessments earlier on in the application life cycle uncover issues before they become bona fide vulnerabilities in the field. Assessments go by many names. During my tenure at @stake, we performed all manner of application assessments at different stages in the application development life cycle (see Table 3-6):

- **Design reviews** at the midpoint of the design stage
- **Architecture assessments** at the midpoint of development
- **Code reviews** (optional) at the end of development for sensitive functions
- **Penetration tests** prior to deployment

Table 3-6 Qualitative Assessments by Phase of Software Development

	Design Review	Architecture Assessment	Code Review	Penetration Test
Test Type				
Goals	Validation of security engineering principles	Verification of implemented security standards	Focused examination of sensitive functions	Identification of deployment flaws
	Identifies gaps compared to security standards	Finds potential architectural weaknesses	Finds development flaws	Finds "real-world" vulnerabilities
Recommended Testing				
External public-facing	Yes	Yes	Yes	Yes
External partner-facing	Yes	Yes	Yes	Yes
Internal enterprise	Yes	Yes	Optional	Optional
Internal departmental	Optional	Yes	Optional	Optional

Enterprises that want to quantify the spread of secure development processes can measure the frequency with which they conduct these activities. Of these activities, penetration tests (also known as "ethical hacking" or "black-box testing") are the best known. Black-box testing uncovers issues in software that an organization has already deployed or has in the field. But not all applications that need testing are always in the field; indeed, one might argue that post-deployment black-box testing comes far too late to uncover important issues. It is always best to detect potential design flaws as early as possible, either through qualitative assessments or via automated code security tools (which I describe later in this chapter).

The first two activities, design reviews and architecture assessments and code, provide qualitative measures of application security. When an organization embarks on a substantive effort to assess applications qualitatively, it must possess a defensible methodology for evaluating and scoring defects. If it does not, the results of different assessments will vary wildly, giving management an excuse not to trust the numbers they see.

Frankly, there is no easy solution for guaranteeing that all team members involved in assessing applications will use the same methodology each time. Every person offers different experiences, creative urges, biases, and interpretations. However, organizations can and should implement standard definitions for terms like "risk" and "impact" that are as unambiguous as possible. In addition, managers should ask team members to agree to use a standard formula for "scoring" application assessments. If an organization can standardize on definitions and scoring formulas, it can partially mitigate the risk of inconsistency.

Business-Adjusted Risk

A common scoring technique is to define an index formula that assigns an overall risk number to defects. Here, I will discuss two that I am familiar with: the @stake business-adjusted risk (BAR) formula for scoring vulnerabilities, and a broader, more general application security index suitable for scoring applications as a whole.

BAR is a technique I invented along with colleagues at @stake, an Internet security consultancy. During its life span from 1999 to 2004, @stake conducted hundreds of application assessments using the exact same formula. BAR classifies security defects by their vulnerability type, degree of risk, and potential business impact. When assessing an application, for each security defect we calculated a BAR score as follows:

BAR (1 to 25) = business impact (1 to 5) × risk of exploit
(1 to 5, depending on business context)

Risk of exploit indicates how easily an attacker can exploit a given defect. A score of 5 denotes high-risk, well-known defects an attacker can exploit with off-the-shelf tools or

canned attack scripts. A score of 3 indicates that exploiting the defect requires intermediate skills and knowledge, such as the ability to write simple scripts. Finally, only a professional-caliber malicious attacker can exploit certain classes of defects; we give these defects a score of 1.

Business impact indicates the damage that would be sustained if the defect were exploited. An impact score of 5 represents a flaw that could cause significant financial impact, negative media exposure, and damage to a firm's reputation. A score of 3 indicates that a successful exploit could cause limited or quantifiable financial impact, and possible negative media exposure. Defects that would have no significant impact (monetary or otherwise) receive a score of 1.

BAR is a simple tool for scoring applications: the higher the score, the higher the risk. Because BAR includes relative ratings for both likelihood of occurrence and business impact, it moves in the same direction as insurers' annual loss expectancy calculations.

BAR suffers from several defects. First, its estimation method is fast and light rather than precise, and it does not quantify risk in terms of time or money. Second, scores are heavily biased by the availability (or lack thereof) of attack scripts and exploit code. BAR scores, therefore, are necessarily temporal—when a hacker or researcher releases exploit code, it changes the score. We believed that this quality was (and is) true to the way the world works, but in practice it causes BAR scores to understate risks over time as new exploits become available. Newer metrics like the Common Vulnerability Scoring System (CVSS) explicitly support temporal adjustments and as such represent an improvement over BAR.

That said, at @stake we were able to successfully and consistently replicate the BAR method over hundreds of engagements. To give you an idea of how BAR works in practice, in 2002 I released a paper called "The Security of Applications: Not All Are Created Equal," which analyzed 45 e-business applications (commercial packages, middleware platforms, and end-user e-commerce applications). We used outlier analysis on 23 of the assessments in our survey. For each engagement, we calculated an overall business risk index, based on the sum of the individual BAR scores. We ranked engagements by their index scores (highest to lowest) and divided them into quartiles. Engagements with the lowest business risk index formed the first quartile; those with the highest formed the fourth. The most-secure applications in our analysis contained, on average, one-quarter of the defects found in the least-secure. The top performers' reduced defect rates also translated into much lower risk scores. The least-secure applications had a BAR score six times that of the most-secure: the fourth quartile had an average BAR score of 332, and the first had an average score of 60. (You can see a graphical depiction of the BAR scores in Figure 6-8 in Chapter 6, "Visualization.")

Application Scoring Indices

Variations on indices for counting and rating specific application defects, such as the BAR technique I described, are common ways to "score" application security. As I mentioned, however, any risk index technique that relies on humans to discern between qualitative levels of risk is prone to inconsistencies.

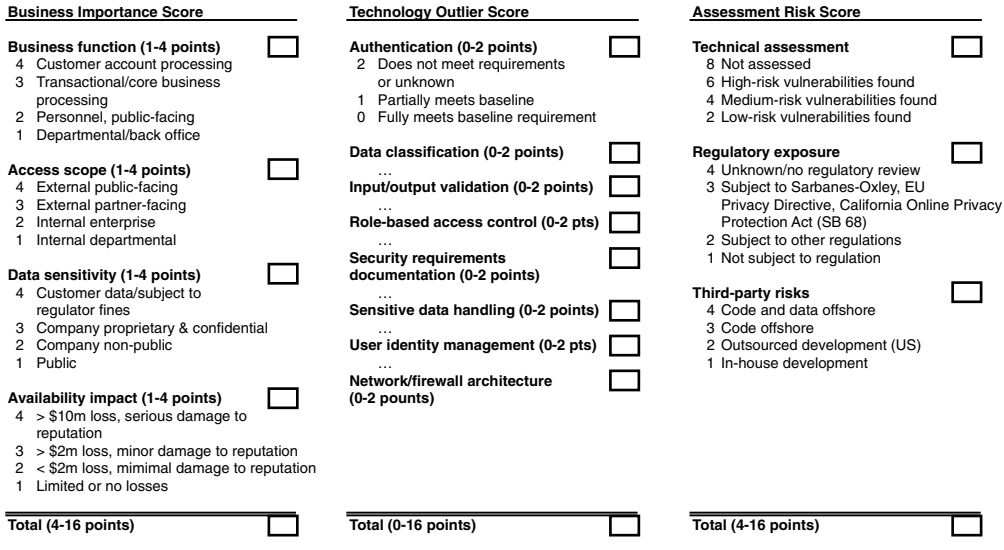
An alternative technique to the various vulnerability-rating methods is a scoring technique that eliminates considerations for things that *might* happen (such as a vulnerability that "could result in financial damage to multiple business units") and replaces them with simple, declarative statements about things that *do* happen ("the server encrypts sensitive data"). Although some subjectivity remains, it is easier to debate about facts instead of hypothetical outcomes. Scoring systems do not necessarily linearly relate to risk; they sacrifice a certain amount of precision for speed and repeatability.

A sample scoring technique that focuses on factual questions is something I have loosely called the Application Insecurity Index (AII).[29] The idea is to create a fast and lightweight application scoring method that assigns points based on whether particular applications meet (or do not meet) specific guidelines and practices. Fact-based questions that result in binary yes/no answers serve as the basis of the score. Figure 3-1 shows the Application Insecurity Index components. The potential score ranges from 8 to 48; lower scores are better.

The AII contains three primary areas: business importance, technology alignment (or lack thereof), and assessment oversight activities:

- **Business importance** scores consider the application's importance to the organization: whether the application faces the Internet, contains sensitive data, costs the organization money when down, or processes business transactions.
- **Technology outlier** scores put a number on the degree to which the application follows prescribed organizational guidelines for eight security topics, including authentication, data classification, validation of user input and output, role-based access control, and identity management.
- **Assessed risk** scores highlight the application's relative riskiness based on whether the application might be considered subject to regulatory inspection or review, such as Sarbanes-Oxley or the European Union Privacy Directive. It also scores whether the application carries any risks associated with third-party code development or data storage, and whether the application has received a technical security assessment.

[29] Feel free to scream.



Business Importance Score

Business function (1-4 points) ☐
- 4 Customer account processing
- 3 Transactional/core business processing
- 2 Personnel, public-facing
- 1 Departmental/back office

Access scope (1-4 points) ☐
- 4 External public-facing
- 3 External partner-facing
- 2 Internal enterprise
- 1 Internal departmental

Data sensitivity (1-4 points) ☐
- 4 Customer data/subject to regulator fines
- 3 Company proprietary & confidential
- 2 Company non-public
- 1 Public

Availability impact (1-4 points) ☐
- 4 > $10m loss, serious damage to reputation
- 3 > $2m loss, minor damage to reputation
- 2 < $2m loss, mimimal damage to reputation
- 1 Limited or no losses

Total (4-16 points) ☐

Technology Outlier Score

Authentication (0-2 points) ☐
- 2 Does not meet requirements or unknown
- 1 Partially meets baseline
- 0 Fully meets baseline requirement

Data classification (0-2 points) ☐
…

Input/output validation (0-2 points) ☐
…

Role-based access control (0-2 pts) ☐
…

Security requirements documentation (0-2 points) ☐
…

Sensitive data handling (0-2 points) ☐
…

User identity management (0-2 pts) ☐
…

Network/firewall architecture (0-2 pounts) ☐

Total (0-16 points) ☐

Assessment Risk Score

Technical assessment ☐
- 8 Not assessed
- 6 High-risk vulnerabilities found
- 4 Medium-risk vulnerabilities found
- 2 Low-risk vulnerabilities found

Regulatory exposure ☐
- 4 Unknown/no regulatory review
- 3 Subject to Sarbanes-Oxley, EU Privacy Directive, California Online Privacy Protection Act (SB 68)
- 2 Subject to other regulations
- 1 Not subject to regulation

Third-party risks ☐
- 4 Code and data offshore
- 3 Code offshore
- 2 Outsourced development (US)
- 1 In-house development

Total (4-16 points) ☐

Figure 3-1 Application Insecurity Index

By design, AII gives the highest scores to applications that serve the most critical business functions, deviate the most from appropriate technical security standards, and have the highest exposures to regulations and identified security vulnerabilities. Also by design, AII focuses on facts that can be quickly ascertained through a lightweight interview process or questionnaire. An application:

- Processes customer transaction data—or not
- Faces the Internet—or not
- Meets requirements for role-based access control to govern authorization—or not
- Handles sensitive data properly—or not
- Has been assessed for application vulnerabilities—or not
- Contains code developed offshore—or not

Criteria for judging standards compliance are likewise straightforward (see Table 3-7). For example, an organization's identity management standard might specify that applications must authenticate users against a centralized LDAP directory, and that external applications used by contractors and partners must be authenticated against a separate directory replica. Scoring compliance for these criteria is easy—the application architects and system administrators will know the answers, and they will not be subject to ambiguity or interpretation.

Lightweight scoring systems like AII should be just that: lightweight. They should take only a few minutes to complete and need not exhaustively score every possible technical or business criterion. When designing such a system, be careful not to rate everything—just high-priority items. It is also important to keep baseline criteria concise and objective. Speed trumps precision.

Table 3-7 Technology Outlier Criteria

Criterion	Standard
Authentication	Sensitive applications require multifactor authentication (username, password, token). Web-based applications require form-based authentication over SSL.
Data classification	The application provides controls for managing different classes of information sensitivity, as appropriate.
Input/output validation	User-supplied input is sanitized before use by the application. Data sent to users are cleansed of malformed or malicious output.
Network/firewall environment	Externally facing Internet applications use a DMZ for web servers, with a separate zone for application servers and databases. Internal core business application servers reside in protected internal subnetworks.
Role-based access control	The application provides separate roles for general users, administrators, and line-of-business roles. Access control rules are expressed as appropriate for roles rather than for named users.
Security requirements documentation	Requirements for authentication, data classification, and so on are explicitly defined in the work order or design documentation.
Sensitive data handling	User credentials are encrypted when in transit. Passwords and sensitive data are encrypted when in storage.
User identity management	For corporate applications, user identities are stored in corporate LDAP or Active Directory. External/partner identities are stored in a dedicated LDAP/AD replica.

AII is not a standard—it is my own interpretation of a method organizations can use to quickly get a handle on potential trouble spots in their application portfolios. More formal methods for assessing application risk exist, of course.

More formal methods for assessing and scoring application risk include the Relative Attack Surface Quotient (RASQ) methodology popularized by Microsoft's Michael Howard and two researchers from Carnegie-Mellon, Jon Pincus and Jeannette Wing. Carnegie-Mellon's OCTAVE method is another.

RASQ attempts to quantify the number and kinds of vectors available to an attacker.[30] At a system level, analysts identify the relative "attackability" of an application by modeling the potential targets, channels facilitating the attack, and access rights. The net result is a single number capturing the relative size of the attack surface. The advantage of RASQ is that it benefits from a formally developed threat model. Howard, Pincus, and Wing's papers contain good discussion and labeling of generic attack vectors, which helps practitioners.

That said, certain aspects of RASQ make it less suitable as an enterprise scoring method for a portfolio of applications. Critically, RASQ works best when it analyzes a single application over time. But its metrics are not comparable across multiple applications (hence the "Relative" part of the name). In short, RASQ has lots of promise, but it is not a useful metric—yet—that can be consistently measured and cheaply gathered. Research continues, however.

CODE SECURITY METRICS

So far I have discussed qualitative metrics for measuring the security of applications. These qualitative metrics are best used to estimate the risk exposure of applications as a whole. They do not assume any particular inside knowledge about the stuff that applications are actually made of: software code.

Code security metrics tackle measurement of software quality directly. A great body of software development metrics, generally speaking, has sprung up over the last twenty years, and many of these are finding their way into security.

First, consider "code volume" metrics. These count the number of lines or code in an application, or the discrete number of features and functions it provides. One of the most common code-volume metrics is "lines of code" (LOC); many people prefer to measure code by thousands of lines as well (KLOC). As an alternative to LOC counting, some methods simply count "statements" instead of lines of code.[31]

---

[30] M. Howard, J. Pincus, and J. Wing, "Measuring Relative Attack Surfaces," 2003, http://www.cs.cmu.edu/~wing/publications/Howard-Wing03.pdf. Manadhata and Wing, "Measuring a System's Attack Surface," 2004, http://reports-archive.adm.cs.cmu.edu/anon/2004/CMU-CS-04-102.pdf.

[31] Methodology issues occasionally arise with respect to pure LOC counting. Specifically, what constitutes a "line of code"? Does a single statement, split for readability, count for multiple lines? For this reason, the popular Java tool CheckStyle measures code volume by counting statements rather than lines of code. http://checkstyle.sourceforge.net/.

A more subjective metric is "Use Case Points," which counts the number of "use cases" an application supports by analyzing functions with respect to who uses them and under what scenarios. Use Case Points suffer from methodological inconsistencies and the relatively high amount of effort needed to count them, whereas KLOC can be counted easily by machine.

Code volume metrics are not directly related to security, but they provide texture, depth, and context. They become much more interesting when combined with statistics about security defects. By "defect," we mean a flaw in the code as detected by an automated code-scanning program like RATS, ITS4 (open-source) or Klocwork, Coverity, Ounce Labs, or Fortify Software. Typical flaws detected by these programs include unsafe memory handling, lack of validation of user input, and dead (unreachable) code blocks. Thus, a common code security metric is "defect density," defined as the number of defects per unit of code (such as KLOC). In August 2005, The Software Assurance Metrics and Tool Evaluation (SAMATE) working group, a project of NIST, identified defect density as an important metric.[32]

Defect density is not perfect, because it characterizes only raw ratios of flaws in applications. Density numbers do not prioritize issues or difficulty of exploit, as colleague Adam Shostack explains:

> *"I think defect density is hard. I'll trade just about anything for a* `gets` *on a socket. So if program A has a* `strcpy` *of some data, and program B has a* `gets` *on that data, I think there's a qualitative difference. Is it quantifiable? Hard to say. You could say either will get you* `root`*, but maybe one is easier to exploit. Five minutes versus ten is a two-to-one difference."*[33]

Other variations on defect density include "known vulnerability density," defined as the number of *known vulnerabilities* per unit of code, and Ounce Labs' "V-density" metric, which is a weighted sum of all known vulnerabilities. V-density corrects for the rawness of pure defect density numbers by giving heavier weights to the most serious security flaws.

Vulnerability density metrics provide rough quantitative scores of software quality. The primary benefit of density metrics is that they can be calculated consistently across an organization's application development portfolio. On the flip side, critics of vulnerability density metrics contend that the state of the art in code scanning is not very good

[32] "Metrics for Software and Metrics for Tools," 10 August 2005, SAMATE, Paul Black (PPT), as reported and retyped by Chris Wysopal. A majority of the metrics in the Code Security section are derived from SAMATE's excellent summation.

[33] Adam Shostack, e-mail to securitymetrics.org mailing list, 19 September 2005

and suffers from two problems: accuracy and lack of knowledge of certain classes of flaws. With respect to the first problem, most tools tend to overstate the number of vulnerabilities present in code. Fred Cohen explains why accuracy has long been an issue:

> *"The automated code base checking market is still pretty immature and problematic in several ways, most notably the large number of false positives for most tools. The tools really don't seek to understand the code, they tend to be syntactic-based or based on following paths through execution, but they fail to handle lots of things well. . . . Folks at big software companies have programmers that intentionally avoid the stronger solutions in favor of what they understand or what is easier—seat of the pants, so to speak. But still, [the tools] do successfully find the really obvious ones that cause most of the currently exploited errors."*[34]

Chris Wysopal makes the same point more concisely, and wonders whether automated tools will ever be good enough to detect all the serious types of flaws:

> *"The challenge with vulnerability density scores is this: how do you work in false positive and false negative rates, especially the fact that the tool may not have the capability of detecting many classes of security flaws?"*

Both of these gentlemen, whose opinions I respect greatly, have a point. If you cannot trust the tools, can you trust the metric? Generally, the answer is no.

Thus, to complement (and correct for) vulnerability density metrics, SAMATE recommends a metric called "soundness," which denotes the number of correct defect classifications minus the false positives and false negatives, divided by the total number of defects detected. This metric helps put the defect numbers in perspective by assigning a potential range to the tools' error rate. No tool can calculate soundness metrics automatically. Users must work out these numbers on their own, based on their experiences with the tools.

Sampling methods work well for calculating soundness. Users should select several "typical" code modules and compare the results of a manual code review against an automated scan. To calculate soundness, users count the number of defects the scanner missed and add to this result the number of things the scans flagged incorrectly as defects. Dividing this figure by the number of defects identified by the scan yields the "unsoundness" metric; subtracting this percentage from 100% returns the soundness figure.

---

[34] Fred Cohen, e-mail to securitymetrics.org mailing list, 23 December 2005.

Beyond the direct security issues that scanning products can find and enumerate in code modules, organizations should also consider the broader issue of code complexity. Both Bruce Schneier and Dan Geer are fond of pointing out that "complex systems fail complexly." Modern applications are typically complex code edifices constructed with care, built for extensibility, and possessed of more layers than a Herman Melville novel. This is not necessarily a bad thing, but it makes it harder to find and eliminate the root causes of security problems. Thus, if complexity contributes to insecurity, we ought to devise methods for measuring code complexity as a leading indicator of future security problems. Fortunately, the academy is way ahead of us.

The last twenty years have witnessed quite a bit of prior academic research into the relationship between software complexity, defect rates, and reliability. Researchers have theorized that complexity metrics ought to be a good predictor of reliability and the degree to which a module is prone to faults. For example, in a study of a large telecommunications software project, Koshgoftaar, Allen, Kalaichelvan, and Goel concluded that "design product metrics based on call graphs and control-flow graphs can be useful indicators of fault-prone modules. . . . The study provided empirical evidence that in a software-maintenance context, a large system's quality can be predicted from measurements of early development products and reuse indicators."[35] Other researchers have arrived at similar conclusions.[36]

Although different schools of thought each have their favorites, many researchers feel McCabe's "cyclomatic complexity" metric provides an effective measure of complexity in code. Cyclomatic complexity for a code module is defined as the minimum number of paths that in linear combination generate all possible paths through the module.[37] Modules with a high number of branching instructions, excessive nesting, or `if/then` statements generate higher scores than simpler modules do. A cyclomatic complexity of 1 to 4 denotes relatively low complexity, 5 to 7 suggests moderate complexity, 8 to 10 denotes high complexity, and anything over 10 is considered highly complex.

Many common development tools can create cyclomatic complexity metrics. An open-source toolkit that works well for Java code, for example, is the Project Mess

[35] T. Koshgoftaar, E. Allen, K. Kalaichelvan, and N. Goel, "Early Quality Prediction: A Case Study in Telecommunications," *IEEE Software*, January 1996.

[36] See http://hissa.ncsl.nist.gov/HHRFdata/Artifacts/ITLdoc/235/appendix.htm#418907 for a list of empirical studies investigating the relationship between complexity metrics and reliability. For a contrasting view, see F. Lanubile and G. Visaggio, "Evaluating Predictive Quality Models Derived from Software Measures: Lessons Learned," January 1996, http://www2.umassd.edu/SWPI/ISERN/isern-96-03.pdf.

[37] See NASA's "IV&V Facility Metrics Data Program - Complexity Metrics" web page, at http://mdp.ivv.nasa.gov/complexity_metrics.html.

Detector, aka PMD.[38] The project's slogan, amusingly enough, is "Don't Shoot the Messenger." My own often over-engineered code—in case you were wondering—typically scores between a 4 and 6.

Cyclomatic complexity metrics have the advantage of being relatively easy to calculate. This means that these metrics can be automated and compared across projects. Cyclomatic complexity is the right metric for measuring control flow density on a per-method and per-entity (or per-class) basis. Because security flaws are, at least some of the time, implementation-related, cyclomatic complexity metrics can help predict which classes/methods in an application might experience flaws. That said, the relationship of complexity metrics to security is hypothetical rather than proven. As with any code volume metrics, organizations should consider cyclomatic complexity in the context of additional environmental metrics to produce a true picture of risk.

Code security metrics is one of the most vibrant areas of security metrics. The metrics I suggested in the "Code Security Metrics" section of Table 3-5 represent just a smattering of the ones that groups like SAMATE have been discussing lately. As the software security industry comes closer to consensus on effective code security metrics, more will emerge.

Summary

Security analysts use metrics for many purposes, particularly for diagnosing problems with their organizations' security programs. Diagnostic security metrics borrow from management consulting techniques by asking two questions: what hypothesis can be formed about the efficiency or effectiveness of security controls, and what evidence can be marshaled to support or disprove that hypothesis?

Technical security activities provide a wide variety of metrics that analysts can use as diagnostics. Technical metrics include those that measure:

- **Perimeter defenses:** E-mail, antivirus software, antispam systems, firewalls, and intrusion detection systems
- **Coverage and control:** The extent and reach of controls such as configuration, patching, and vulnerability management systems
- **Availability and reliability:** Systems that ensure continuity and allow recovery from unexpected security incidents

[38] "Project Mess Detector" is just one variation on the PMD acronym. See http://pmd.sourceforge.net/rules/codesize.html for the tools—and for more amusing variants on the name.

- **Application risks:** Defects, complexity, and risk indices for custom and packaged line-of-business applications

Technical security metrics should not simply be "fun facts" or "happy metrics" that tell the CSO what a great job the security team is doing, such as the sheer number of spam messages blocked by the firewall. They should reveal more interesting insights, such as gaps in coverage (ratio of inbound to outbound viruses, patch latency), environmental stability (firewall turbulence, net changes in vulnerability incidence, patches within SLA windows), or problems with change controls (number of change control exemptions per period, unauthorized production changes). For all these metrics, historical measurement allows analysts to trace performance over time and detect outliers.

Application security metrics represent an entirely separate measurement domain with its own diagnostics. Black-box metrics count defects detected by scans. Process metrics about the frequency of security reviews and go-live assessments help measure the spread of secure development processes. Application security indices allow organizations to quickly "score" application security risk across an entire portfolio of applications. Code metrics such as vulnerability density and cyclomatic complexity provide raw measures of how secure and reliable code modules are likely to be.

Technical activity metrics are ideally suited to serve as diagnostic metrics because they fulfill many of the qualities that good metrics should have. They are expressed as numbers, incorporate clear units of measure, and can generally be computed on a frequent basis because their data flow from deployed IT and security software packages.

Technical metrics are not the only ones worth considering, of course. Security program metrics incorporate the overall processes that technical systems participate in. We turn to these in the next chapter, "Measuring Program Effectiveness."