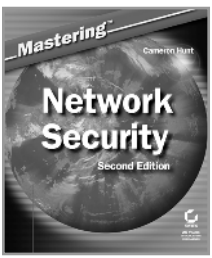
Chapter 1

A Systems Approach to Information Networks

We all probably have an idea about what the word *system* means. For those of us who work in information technology, the term has become a catch-all that covers everything from an operating system on a single computer to the Internet itself. Now, we know you're probably thinking that we're going to spend a whole chapter convincing you that because your network is complex, securing it will also be complex. You would be right!

But we're going to do more than that. We're going to give you a small tour through the idea of complexity and how it relates to anything that gets labeled a "system." The goal is to give you several principles that you can apply to any complex system—whether that system is a network, a data recovery procedure, or a security decision tree—in order to build one or more models. These models not only provide a common reference for author



Adapted from *Mastering™ Network Security*, Second Edition by Chris Brenton and Cameron Hunt
ISBN 0-7821-4142-0 $49.99

and reader, but are valuable tools in their own right in understanding, planning, implementing, and managing any complex group of interrelating, dynamically operating parts. And if anything fits that description, it's a computer network (the thing we're trying to secure, remember?).

An Introduction to Systems Analysis

Systems analysis is the formal term for the *process* (which we cover in Chapter 2) that uses systems principles to identify, reconstruct, optimize, and control a system. The trick is, you have to be able to walk through that process while taking into account multiple objectives, constraints, and resources. Simple, but what's the point? Well, ultimately you want to create possible courses of action, together with their risks, costs, and benefits. And that, in a nutshell, is what network security is all about—choosing among multiple security alternatives to find the best one for your system, given your constraints (technical or financial, typically).

The principles that make up systems analysis come from several theories of information and systems. Let's look at Information Theory first. In its broadest sense, the term *information* is interpreted to include any and all messages occurring in any medium, such as telegraphy, radio, or television, and the signals involved in electronic computers and other data-processing devices. Information Theory (as initially devised in 1948 by Claude E. Shannon, an American mathematician and computer scientist) regards information as *only* those symbols that are unknown (or uncertain) to the receiver.

What's the difference between symbols that are known and symbols that are unknown? First, think of long distance communication a little more than a century ago, in the days of Morse Code and the telegraph. Messages were sent leaving out nonessential (predictable or known) words such as *a* and *the*, while retaining words such as *baby* and *boy* (defined as unknown information in Information Theory). We see the same kind of behavior in today's text messaging—minimal words and abbreviations come to stand for entire phrases.

Shannon argued that unknown information was the only true information and that everything else was redundant and could be removed. As a result, the number of bits necessary to encode information was called the *entropy* of a system. This discovery was incredibly important because it

gave scientists a framework they could use to add more and more bandwidth (using compression, or the removal of redundant information) to the same medium. For example, modems increased their speed to the point they were transmitting 56,000 bits of information per second, even though the physical medium of the phone line could represent only 2400 changes (known as *bauds*) per second.

We point out Information Theory and Shannon's definition of information is to illustrate a central concept of understanding systems: You don't have to know *everything* about a system to model it; you only need to know the unknown or nonredundant parts of a system (the information) that can affect the operation of a system as a whole. You can ignore everything else; for all practical purposes, it doesn't exist.

System analysis also draws heavily from another discipline, Systems Theory. Traditional Systems Theory tends to focus on complex (from the Latin *complexus*, which means "entwined" or "twisted together") items such as biological organisms, ecologies, cultures, and machines. The more items that exist and are intertwined in a system, the more complex the system is. Newer studies of systems tend to look not only at items that are complex, but also at items that are also *adaptive*. The assumption is that underlying principles and laws are general to any type of complex adaptive system, principles that then can be used to create models of these systems. The following are some of these principles:

Complexity Systems are complex structures, with many different types of elements that influence one another. For example, a computer network encompasses software, different layers of protocols, multiple hardware types, and, of course, human users—all interacting with and influencing one another.

Mutuality The elements of a system operate at the same time (in real time) and cooperate (or not). This principle creates many simultaneous exchanges among the components. A negative example of this is a positive feedback loop! Imagine a computer that creates a log entry every time the CPU utilization is greater than 50 percent. Now, imagine the consequences that will occur if every time the system writes an error log, it forces the CPU to be used greater than—you guessed it—50 percent.

Complementarity Simultaneous exchanges among the elements create subsystems that interact within multiple processes and structures. The result is that multiple (hierarchical) models are needed to describe a single system.

Evolvability Complex adaptive systems tend to evolve and grow as the opportunity arises, as opposed to being designed and implemented in an ideal manner. Now, this definitely sounds like most computer networks we've been privy too—patchworks of various brands, capabilities, and complexities, implemented in pieces as time and resources allow.

Constructivity Systems tend to grow (to scale), and as they do so, they become bound (in the sense of heritage) to their previous configurations (or models) while gaining new features. Anyone who has worked at an organization over an extended period of time has seen this happen. No matter how large the network grows (unless there was a major overhaul somewhere), it still seems to fundamentally reflect the small, original network it originated from, even with additional capabilities and features added over the life of the system.

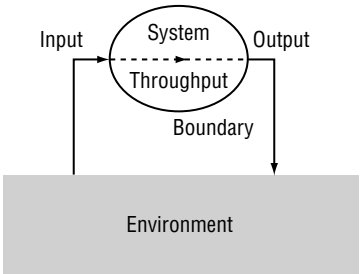
Reflexivity Both positive and negative feedback are at work. Because this feedback affects both static entities and dynamic processes, the system as a whole begins to reflect internal patterns. You'll notice that the physical network begins to reflect the way you use that network.

The original Systems Theory was developed in the 1940s by Ludwig von Bertalanffy (1901–72), a biologist, who realized that there were no effective models to describe how biological organisms worked in their environment. Physicists at the time could make a small model of the solar system (through a process of both analysis and reductionism, breaking the components and functions down to their smallest, simplest parts) that would accurately predict planetary orbits while ignoring the universe at large. Biologists, however, could not completely separate an organism from its environment and still study it; it would die of starvation, cold, or boredom. As a result, the systems approach tries to combine the analytic and the synthetic methods, using both a holistic and reductionist view.

Again, think of how this concept applies to a computer network. Systems inside and outside the network make up the environment of the network itself. Although we can break down the parts and functions of a network, we can truly understand it only by looking at the dynamic interaction of the network with other components, whether those components are other networks or human beings.

To identify a system means to identify a boundary. The reality, especially in our connected world, is that boundaries are often arbitrarily dictated and defined, not necessarily created through physical reality. Placing a firewall between your business LAN and the Internet may or may not establish a boundary between two systems. It all depends on the model—the way in which you view your network.

Assuming that we *have* defined a boundary between the system and its environment, we can add some concepts that define how a system interacts with that environment. In the following illustration, *input* is defined as any information added into the system from the environment. *Throughput* is defined as those changes made to the input by the system. *Output,* of course, is what leaves the system and crosses the boundary back into the environment.



Of course, the environment itself is made up of one or more systems, and we rapidly reach the conclusion that defining a system (which really means defining a boundary between one system and its environment) is really a matter of scale and perspective—a concept that lets us begin to see systems in a hierarchical order (more on this later). Whereas you might view the Internet and your internal LAN as two separate systems, that model no longer functions as effectively when you consider remote workers accessing your network through a VPN (virtual private network) or even a web mail session secured through SSL (Secure Sockets Layer).

If you look at a system as a whole, you don't necessarily need to be aware of all its parts. This perspective is called the *black box view*—seeing a system as something that takes in input and produces output, with us being ignorant of the throughput. (Seeing the innards would then be called a *white box view.*) Although the black box view doesn't necessarily satisfy your inner control freak, it's not always necessary to see the innards of a process in order to implement and maintain it. (Remember the definition of *information* according to Information Theory?) This approach is common in the complex world of information technology, where we often work with black box abstractions of data operations.

In the realm of object-oriented programming languages such as C# and Java, reducing a code object to a black box is considered a primary strength. You are able to use the functionality of a code object (written by another programmer) in programs of your own without knowing *how* the object does the work. As long as the methods used to access the capabilities of the object or its accessible properties don't change, the authors can change, update, or rework their object in any way they desire. Your code can stay the same!

The challenge in dealing with information technology, as well as in dealing with *any* complex adaptive system, is to identify when you should use black or white approaches. And the capability of using both black and white approaches illustrates another principle that we mentioned earlier: Systems are hierarchical. At the higher (or unified) level, you get an abstraction of the system as a whole. At the lower (reduced) level, you see many interrelating components, but you don't necessarily know how they fit together.

According to the traditional analytic approach (the one that existed before von Bertalanffy came along), that low-level view is all that is necessary to understand a system. In other words, if you know the precise state of every component in the system, you should be able to understand how the system functions. Anyone who has ever tried to optimize an operating system for a given task (such as a web server or a database server) knows how limiting this model can be, simply because performance rarely scales in a linear fashion. In other words, increasing the number of users by a specific amount doesn't always guarantee the same rate (proportional or not) of resource utilization.

In the same fashion, doubling the amount of RAM doesn't automatically increase RAM-based performance by the same percentage. Computer components don't (often) exist in simple, linear, cause-and-effect relationships; rather they exist in complex networks of interdependencies that can only be understood by their common purpose: creating the functionality of the system as a whole. Looking at RAM or disk I/O or a CPU as individual elements isn't sufficient to understand resource utilization until you understand the relationship each of these elements has to the others—something not readily apparent by simply dissecting their design.

All this might seem like common sense, and if you've spent any time dealing with computer networks, you've probably come to the belief that these ideas are true, even if you haven't known *why* they are true. Most IT workers have an emotional reaction (and not necessarily a positive one)

to the overwhelming complexity and unpredictability normally experienced by trying to understand, let alone manage, a complex system. Add in the feedback (in the *systems* sense of the word) of human users (each with their own method of interacting and altering that system), and we now have to struggle with a complex *adaptive* system.

But we're not done yet. Remember that we said systems have hierarchies? Understanding that systems can affect the structure and functionality of subsystems and, likewise, that subsystems can influence the behavior of a parent system or systems (both directions of influence occurring simultaneously and repeatedly over a period of time) is crucial in Systems Theory. This theory also states that systems tend to mimic (in a general sense) the structures and the functions of their parent systems.

Let's look at a biological example. The cells in your body have boundaries (the cell wall), inputs (the structures on the cell wall that bind to proteins and usher them into the body of the cell), and outputs (internal cell structures that eject waste through the cell wall to the outside). Your body as a whole has inputs (your mouth and nose), outputs, and a boundary (the skin). Both your body, as a parent system, and your cells, as subsystems, have to take in nourishment. Both transform that input into an output. Although the specifics are different, the functions are the same: to allow sustenance, growth, and repair.

Similar structures also exist in the hierarchy of systems. The inputs on your body serve not only to transport nourishment, but also to analyze and prepare it for the body. Likewise, the inputs on the cell (located on the cellular wall itself) identify and "format" the proteins for the use of the cell. Systems Theory, ultimately, asserts that there are universal principles of organization that hold for all systems (biological, social, or informational) and that we can use these principles to understand, build, and manipulate those systems.

Now that you have a better understanding of the theory of information and systems, you need a practical way not just to understand a complex system, but to predict how the system will respond to changes. Such a method allows you not only to understand the security risks a computer network might face, but the consequences (especially the unforeseen ones) of trying to mitigate that risk. The name given to this practical method of managing systems is *systems analysis, decision analysis,* or even *policy analysis.* We'll use the traditional term *systems analysis*.

The systems analysis model is a multidisciplinary field that includes programming, probability and statistics, mathematics, software engineering, and operations research. Although you don't need a background in any

of these areas to use the model, understanding the background will help you use the tools. The typical systems analysis process goes something like this:

1. Define the scope of a problem.
2. Determine the objectives, constraints, risks, and costs.
3. Identify alternative courses of action.
4. Evaluate the alternatives according to the constraints (feasibility), the fixed costs (cost-effectiveness), the ratio of benefits to cost (cost-benefit), or the ratio of benefits to risk (risk-benefit).
5. Recommend an alternative that will meet the needs of a decision maker (without violating the constraints of the system).

Sounds easy enough, right? You're creating a model of the system. This model allows you to apply metrics (measurable behaviors of the components, their behaviors, and relationships) in order to make decisions about what courses of actions will allow you to meet your objectives.

Two major challenges are associated with systems analysis of network security. The first is to assign realistic values to the frequency of threats. As we'll illustrate later, the frequency of a threat is one of the primary ways you determine the actual risk to a system. The second challenge is to decide which evaluation criteria to use (the items from step 4). Traditional computer network security has attempted to use all the criteria in the decision-making process, while giving the greatest weight to cost-benefit.

Now that we have a list of the steps in the systems analysis process, let's walk through each of them in more detail.

Define the Scope of the Problem

In systems analysis, a *problem* is something in the system or its environment that requires the system to change. The *scope* of network security includes protecting the system from data corruption and ensuring the availability of data, no matter where the threat originates. The result of this definition of scope is that even if you have no external environmental threat from hackers, you still need to determine if the design of a network itself could put your data at risk—for example, by not providing sufficient levels of data redundancy.

In a practical sense for any individual involved in network security, defining the scope of the problem comes down to two questions: what and why? The first question is essentially about responsibility: What assets (or systems) are you in charge of protecting? This quickly moves beyond a technical arena into the specifics of your business, job, or role within a security effort. Once you clarify the *what* of your work, you can start to define the *why*. In other words, you can evaluate the current state of the system (*state* being formally defined as the current value of any variable element in a system) and decide what needs to be changed.

We'll introduce the formal security process in Chapter 2, but you can probably guess that in an ideal, formal setting, you receive a document that clarifies the areas (or systems) of your responsibility. You then attempt to determine the current state of the system, followed by an analysis of the problem. In network security specifically, this process means identifying and quantifying the risks to your data, including the systems that process, store, and retrieve that data.

Determine Objectives, Constraints, Risks, and Cost

In systems analysis, an *objective* is simply the outcome desired after a course of action is followed. Because objectives (like systems) usually exist in a hierarchy (descending from general to specific, nonquantified to highly quantified), we usually refer to higher-level, abstract objectives as *goals*. Specific quantifiable objectives are referred to as *targets*.

An example of a goal/target combination is a corporate website. The *goal* is to maintain the functionality and integrity of the website as a whole. A *subgoal* is to protect against web page defacement. Two specific *targets* that support that subgoal (which, in turn, supports the overall goal) are to apply vendor security patches to the web server within five hours of release and to create a secure, mirrored content server that overwrites the master website with correct content every five minutes.

Nice and easy, right? The problem comes when you have multiple objectives that are contradictory or competitive (also known as conflicting objectives). You usually see conflicting objectives when more than one party is responsible for the state (remember the definition of state?) of a system. You probably already know how rampant conflicting objectives are in the security world, because implementing security almost always comes down to restricting behavior or capability (or increasing cost).

Unfortunately, restricting system capability tends to conflict with the central purpose of information systems, which is to enable and ease behavior or capability. For example, think of how quickly you find notes hidden under keyboards when password complexity and length requirements are enforced in an organization!

Fortunately, systems analysis gives you a method for resolving conflicts by providing hierarchical decision makers. Because the means of achieving your goal of system security might conflict with the accountant's goal of maintaining a low cost of the system, a decision maker at a higher level is usually required to determine either which goal takes precedence or (more commonly in the real world) how to change the constraints of each goal so that they are no longer in opposition. In other words, executive-level decisions are often required to reach a compromise between two competing goals.

A byproduct of conflict resolution is the creation of *proxy* objectives—replacing generalized objectives with those that can be measured in some quantifiable way. An example of proxy objectives is illustrated by multiple security plans, each with a quantified cost/benefit ratio, that are presented to senior management who make the final decision based on how much risk they are willing to accept (the greater the risk, the lower the *initial* cost).

So what are constraints? According to systems analysis, a *constraint* is a limit in the environment or the system that prevents certain actions, alternatives, consequences, and objectives from being applied to a system. A simple, but limited way to understand this idea is to think of the difference between what is *possible* to do in a system and what is *practical*. Thinking of a constraint this way makes it easier to identify the consequences of any given course of action on a system.

A good example is a requirement mandating that biometric security devices (such as a fingerprint scanner) be used on every desktop computer in an organization. Although using a fingerprint scanner would achieve a major goal of network security (and is technically possible in most cases), you could easily run into constraints—initial equipment cost, client enrollment (storing authenticated copies of every employee's fingerprints), and non-biometric capable access devices (such as a Palm Pilot or other PDA) that make the solution unworkable according to other goals (such as maintaining your security within a certain budget).

We know this sounds complicated, and it is. Using systems analysis to guide you in your security process has great rewards, but it also requires you to have a thorough knowledge of your network inventory (hardware, software, and configuration), business procedures and policies, and even some accounting. Using formal worksheets and checklists to guide you

through the process is highly recommended, as is hiring a consultant who specializes in systems analysis in a security context.

Once you identify your objectives and initial constraints (additional constraints usually show up when you are defining various courses of action), you need to identify risk. *Risk*, in systems analysis, can mean several things. For our purpose, we'll choose *risk assessment*, which is a two-part process. The first part is identifying the impact (measured, from a security perspective, in cost) of a threat (defined as a successful attack, penetration, corruption, or loss of service); the second part is quantifying the probability of a threat.

We can use the web page defacement example to illustrate risk assessment. You begin by identifying the threat (a successful web page defacement) in terms of the cost to the organization. Now things become difficult to quantify. How much money does an organization lose when investor and customer confidence is lowered (or lost) when a page is defaced? What if the particular page was interactive and the defacement breaks or inhibits commercial interactions?

You could even break the threat down to finer details, assigning cost to each individual defaced page, varied by the amount of time the page was defaced; the time of day, month, and year the defacement took place; the amount of publicity received; and the functionality that was broken.

You must also consider another type of impact: Does the system state change after a threat? In other words, the process needed to deface your web page most likely results in the attacker having some level of control over your system. This, according to strictly defined systems analysis, has changed the state of your system, especially if you extend your concept of your system to include those individuals who are authorized to use your system. Once your system state has changed (for better or worse), new threats are possible, requiring a repeat of the entire risk assessment process. This recursive, hierarchical analysis helps you to establish a multilayered defense—something we'll refer to as *defense in depth*.

Once impact is quantified, you have to assess the probability of a threat (remembering our definition of a threat as an *actual occurrence* of a specific negative event, not just a possible one). You can begin to assess the probability of a threat against a system by using simple comparison: Identify all the known characteristics of systems that have succumbed to that threat in the past. In our web page defacement example, you compare your system (including operating system, web server configuration, level of dynamic code, public exposure and opinion of the website, and so on) to those that have been defaced before.

This process sounds relatively straightforward, but you can't stop here. You also need to attempt to weigh those system characteristics in susceptibility to the threat. Using an example from history, which factor contributed more to the Department of Justice website defacement (the one that left then-Attorney General Janet Reno with a Hitler-like mustache!)—the operating system or the type of web server running on the operating system? The answer is a little more difficult to determine and takes experience along with knowing *how* the threat was carried out. We quickly come to the conclusion that we need another hierarchy: a hierarchy of threats. Although the end threat is still web page defacement, an attacker could use multiple methods to deface a page. The threat probability is then a combination of which methods of attack are the most popular, along with which system configurations are most susceptible to those popular attacks.

Remember that we're speaking of system configuration in the systems analysis sense of the term. Part of your website system is the *environment* in which it operates, including the popularity and publicity associated with that site. If you are, say, a U.S. military organization, the state of your system guarantees a higher level of interest. That raised interest can translate into a higher frequency and sophistication of attack. In this example, not only has the *frequency* of the threat possibly changed, but also the nature of the threat *itself*.

Once you identify your objectives, constraints, and risks, you're ready to decide on *courses of action*—nothing more than the ways in which an objective will be met. Multiple courses of action are defined as *alternatives*, and then only if they are mutually exclusive. For example, an objective requires a standard biometric authentication device across an organization. If the decision makers in the organization are trying to decide between fingerprint scanners and iris scanners, they are said to be selecting from two alternatives. If the organization decides that it could use both alternatives together in a standardized fashion—fingerprint scanners for desktops, iris scanners for server rooms (or combine elements from two mutually exclusive alternatives)—a new, distinct alternative has been created (and possibly a new objective, depending on how strictly that objective was originally defined).

Defining an alternative means to establish the feasibility, costs, benefits, and risks associated with a course of action—a process that usually occurs repeatedly, starting with a multitude of alternatives that are gradually integrated and combined until at last you reach a

small collection of alternatives. At this point, the process usually stops for a couple of reasons:

- You don't have sufficient information to continue an evaluation. Perhaps no one has yet conducted a TCO (Total Cost of Ownership) study comparing biometric authentication with centrally stored user profiles against a system using smart dongles that store an encrypted copy of the user's profile in an embedded chip.
- All the alternatives that could meet the objective are greater than the budget constraint (an objective/constraint conflict), which (as we mentioned earlier) usually requires the intervention of a higher-level decision maker.

And, after all your hard work, all that is left to do is present your proposal to the decision makers. Although it can be bad enough if someone questions your results, it's worse when your decision makers don't understand your methodology.

Applying Systems Analysis to Information Technology

Now that we've covered the general theory of systems analysis, let's apply it to IT systems specifically. This approach might seem like unnecessary repetition, but it's actually an attempt to reinforce important concepts while adding details that are specific to issues you'll face in dealing with security.

When you begin to analyze your network (in preparation to secure it), you'll break it down into four general areas:

Data The nature of the information stored and processed on the system

Technology The different types of technology used in the system

Organization How the organization as a whole uses the system

Individuals Key decision makers and personalities that use the system

The Nature of the Data

Understanding your organization and the type of work it does goes a long way toward understanding the type of data stored and processed on your system. Translating this knowledge into specifics follows a task orientation: What is your system used for? Some smaller organizations primarily process and store groupware—common address books, shared or centrally stored files, and simple databases, along with e-mail and a website or two. Larger organizations tend to break down their network segments, and the network technical divisions begin to mirror the network logical divisions. For example, a company places its Internet-accessible resources (web servers, mail servers, and so on) on a network that is separate from its internal network (for security and performance reasons, among others). In this case, a parent system (the functionality) is driving a change in a subsystem (the technology).

The Types of Technology

Technology itself helps define the structure of the system, but primarily as background. In other words, understanding the technological topology of your system will help you formulate your constraints and identify and quantify your threats, and will ultimately play a big part in formulating your risk assessment.

How the Organization Uses the System

Understanding how your organization uses the system can be easier in larger organizations, in which the network tends to follow organizational lines along centers of power or divisions of labor. However, even in smaller organizations, understanding how the network is used and *perceived* by the organization becomes critical to projecting the consequences of your various courses of action. Those consequences play a primary role in determining which alternatives you choose to solve a problem.

How Individuals Use the System

This task is not just about evaluating the technical ability of individuals in an organization or simply identifying those with the most influence. It also concerns determining the relationship those individuals have with the system and determining their knowledge of how the system as a whole works, even to the point of the organization's relationship with the system.

Models and Terminology

Once you look at your network through these types of filters, you're ready to begin defining the subsystems. Selecting where to draw the border of a subsystem is always difficult, but, again, systems analysis gives you some direction. Object-Oriented Systems Analysis (OSA) takes the concept of the black box discussed earlier and uses it to create object-based models using the components of the network, using three model types:

ORM (Object-Relationship Model) Defines objects (and classes of objects), how objects relate to classes, and how objects map to real-world components

OBM (Object-Behavior Model) Defines the actions of objects (used to define how and why an object changes state)

OIM (Object-Interaction Model) Defines how objects influence one another

Object-Relationship Model

An *object* is a label you apply to a single thing that has a unique identity either physically or conceptually. Here are a few IT objects:

- Router #13
- `www.go-sos.com`
- An inventory database
- The first primary partition of the second hard drive of the web server

In OSA, objects are represented with a lowercase labeled dot:

router #13

Objects can be grouped into one or more *object classes*, such as the following:

- Router
- URL
- Database
- Partition

Object classes are represented with a cylinder and a capitalized label:

For an object to be a member of an object class, it has to meet the constraints. That requirement might seem obvious, but think about the Routers class in the previous example. It's easy to think of a dedicated router as belonging to the Router class, but what about a Windows 2000 server that shares files, hosts e-mail, and provides a VPN connection between a small office and corporate headquarters? Although we don't necessarily think of this machine as a router, it acts in that capacity (by routing traffic over the VPN). Including it in the Router class would depend on the constraints of the Router class; in other words, how you define the class determines what objects qualify for membership.

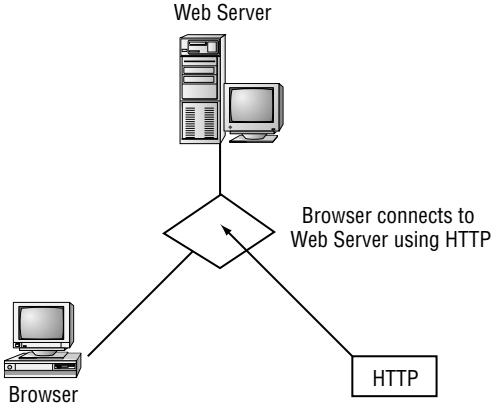
NOTE

Objects can migrate from class to class as class constraints change or as the state of the object changes.

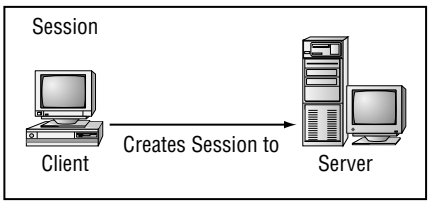
Objects can have a relationship, which is represented by a simple line between them labeled with a sentence that describes the relationship. Usually, however, this relationship reflects a relationship *set* between object classes. Relationships are grouped into sets when they connect to the same object classes and represent the same *logical* connection among objects. To illustrate a relationship set, you draw the two object classes as boxes and connect them:

When a relationship set has multiple connections, these connections are referred to as the *arity* of the set. Two connections make a set binary, three connections make a set ternary, and four connections make a set quaternary. Relationships with five or more connections are referred to as *x*-ary, with *x* reflecting the number of connections. When illustrating a relationship set with more than one connection, a diamond is

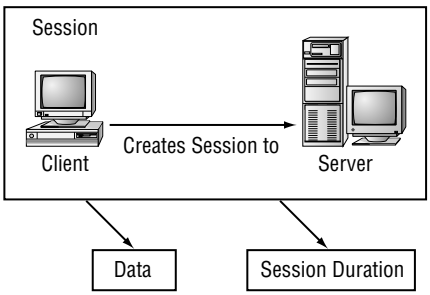
used to interconnect the lines:



To give an even better level of detail, you can treat the relationship set as an object, which in this case is called Session:



Treating the relationship set as an object allows you to link the Session object to other objects or object classes (in this cases, byproducts or characteristics of the session):



This graphic illustrates that a Session object (really a relationship set) has a relationship between a Data object class and a Session Duration class, much as any network session in the real world has data associated with it, along with an amount of time the session existed.

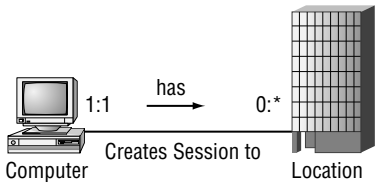
But something is still missing. When working with relationship sets, you need to clarify the constraints. There are three types:

Participation Defines (for every connection) how many times an object class or object can participate in the relationship set.

Co-occurrence Similar to participation, co-occurrence specifies how many times an object can participate in a relationship set with another object. This constraint can also apply to object collections.

General Defines what is allowed or not allowed in a relationship. This constraint can be expressed as a formal math/logic statement or as a simple statement.

Let's look at an illustrated example of a Participation constraint:

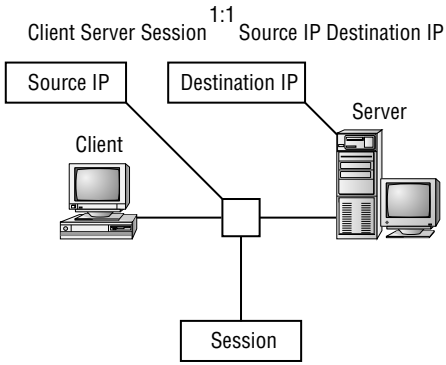


This illustration tells you that a Computer object (belonging to the Computer object class) must have one (but only one) Location object (again, of the Location object class). A Location object, however, doesn't even have to have a single corresponding Computer object, but *can* have an infinite number of Computer objects. For example, if you define a Location object to have a value of Corporate, it is tied to all the Computer objects that are mapped to physical machines at the corporate office (a one-to-many relationship, for all you database programmers).

This makes sense, but you could quickly run into a problem. What happens if you decide to map a laptop (the term *map* in OSA denotes an association between a physical item and a logical object) to the Computer class? Because a Computer object can have only one location (as defined by the Participation constraint), you could have difficulty if you are analyzing objects over time with an expectation that the Location value won't change! You can solve that problem by simply mapping laptops to a Laptop class that doesn't have the same constraint.

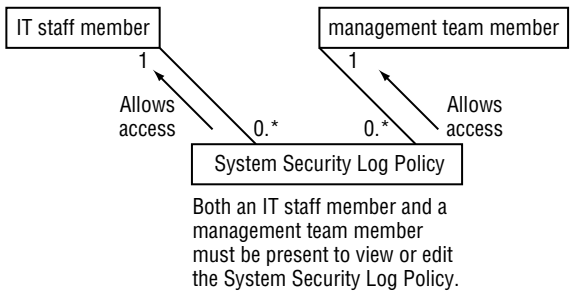
The following illustration shows how Co-occurrence constraints can limit the number of objects that can be associated with an object (or a

group of objects) in a relationship set:



This example is also fairly straightforward. When Client, Server, and Session objects are in a relationship set, there can be only one Source IP and one Destination IP object in that set. This arrangement is similar to real-life network communications, in which the source and destination IP addresses don't change for the duration of a session between a client and a server.

Both of the previous examples illustrate constraints that arise from the workings of the objects themselves (or, in other words, the interaction of the objects *determines* the constraint). However, General constraints often represent constraints imposed on the objects. An example of General constraints is often seen in IT systems in which policies exist about how the system *should* be used, as shown in the following illustration:



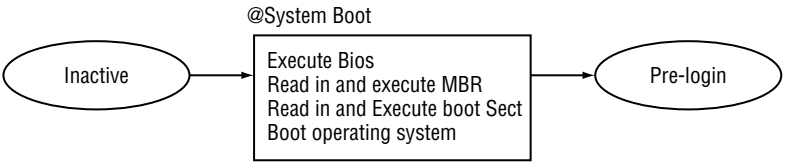
This example shows how a simple text sentence is used to create the General constraint. Because IT security policies are all about limiting the use of a system, General constraints are used frequently.

Once you model the objects and their relationships in the system, you're ready to examine their state, defined in OSA through the Object-Behavior

Model as the activity or status of an object. You illustrate state by using an oval and writing the name of the state inside that rectangle:

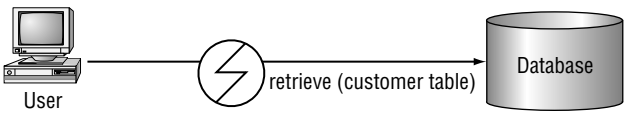
How do you determine all the states an object can have? That's really up to you; you use your experience and understanding of the object you are representing. States are binary—either on or off—and they are activated and deactivated when control (or flow) transitions to another state. Exceptions to this rule are threads (you can turn on multiple threads), prior conjunction (turning off multiple states when one is turned on), and subsequent conjunction (turning multiple states on when one is turned off).

A transition is also formally diagrammed in OSA, sometimes with an identifier, but always with a trigger and an action. The trigger defines the conditions under which a transition fires, along with the resulting action, as shown in the following illustration:



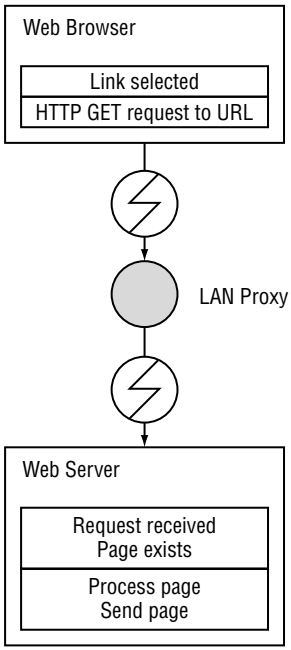
This example uses an event-based trigger (hence the @ sign). Event-triggered transitions execute their action the moment the event becomes true. Conditional triggers (informal statements not preceded with @) cause the transition the *entire time* they are true.

Now that you can illustrate objects and relationships using ORM and use multiple state transitions (known as *state nets*) to show how objects behave, you can use the Object-Interaction Model to show how objects and states are changed through interaction. Here is a basic example of an interaction:



Both User and Database are object classes, with the interaction defined by a circle with a zigzag; *retrieve* is the action, and *customer table* is the object transferred in the transaction. You can use a more complete

illustration to show how object class, state changes, and interactions work together:



This is a complete (although simplified) example of putting together all the pieces of the OSA to represent your network as a system. Remember also that this is just an introduction to familiarize you with the concepts and symbols. Many specialized security consultants will use these diagrams to document your network, analyze courses of action, and present you with alternatives that meet your criteria and solve your problems. Knowing how to read their documentation will aid you in making better decisions. Although you might not go to the same lengths in analyzing or illustrating your own network, the principles remain the same.

Formal systems analysis makes understanding the environment of a system as important as understanding the system itself. This can be more difficult—especially determining which elements of an environment actually influence the system (particularly those items that remain unknown). Once you begin making a list of items that *might* influence or affect your system, you identify the techniques to determine if they do. Here are some items to look for:

Data The flow of information streaming into a system

Technology The limitations and capabilities of other systems that interact with your own

Competition The capabilities of other organizations

Individuals People whose activities could influence your system, such as crackers

Capital The quantity of resources held by systems outside your own

Regulations The legal limitations faced by all systems

Opportunities The innovations and capabilities not yet integrated into your own system

What's Next

By formally documenting your system, you can better understand the vulnerabilities and threats it faces. By using the same theory and techniques to document your security alternatives (including their consequences), you can make better choices about *how* to secure your system. In the next chapter, we'll take these techniques and place them in a dynamic context of the security process—an important reminder that security is a constant effort, not just a project to complete!