Chapter 4

Risk-Based Security Testing: Prioritizing Security Testing with Threat Modeling

Focus testing on areas where difficulty of attack is least and the impact is highest.

—Chris Wysopal

Time and resources are constrained in the business world, and the software development process is no exception. Time to market is critical in software development, whether the end result is a state-of-the-art media player or a new interactive customer Web site. It is necessary to prioritize the tests to be performed to maximize the quantity and severity of the security flaws uncovered given a fixed amount of time and resources.

A technique for prioritizing security testing is *threat modeling*. With threat modeling, potential security threats are hypothesized and evaluated based on an understanding of the application's design. The threats are then ranked and mitigated according to the ease of attack and the seriousness of the attack's impact. Security testers can then focus their testing on the areas where the difficulty of attack is least and the impact is highest.

Wysopal_Rev-Chapter 04.qxp 10/5/2006 12:28 PM Page 74

Some security professionals think the term *threat modeling* is a misnomer and that the term *risk modeling* is more apt. Risk-based testing is a well-known approach to functional software testing. The term *threat modeling* as used by application security experts can be considered another type of risk-based testing related to security.

Information Gathering

To perform threat modeling, you need a solid understanding of the program's design. You need to know where all the entry points are so that you can understand and diagram the program's *attack surface*. The Chapter 1 section "The Fault Injection Model of Testing: Testers as Detectives" mentioned that, to be effective, security testers need to start thinking like an attacker. Evaluating the attack surface lets you target the areas where an attacker might find an entry point.

You also need to understand the valuable functions that the program performs and its assets. Assets are resources that the program must protect from misuse by an attacker. This is important for the understanding of the high-risk areas of a business application.

Typically this requires reviewing the program's design documents, including data flow diagrams if they exist, and interviewing the program's designers or architects. Access to the designers is the most efficient way to get the necessary design information to perform threat modeling because often the type of detailed security information needed is missing from typical design documents. Additional attack surface information can be gathered by runtime analysis—executing the program and analyzing it with debugging and diagnostic tools. (Chapter 11, "Local Fault Injection," discusses this in more detail.)

Meeting with the Architects

You should start with an overview or kickoff meeting with the development manager and as many of the architects (if there is more than one) as you can. A small application may have just one architect, but a large application usually has several architects or "component owners." This is the time to get an overall view of the system. Having all the component owners in the room ensures that no major systems get missed and that the interactions and dependencies are all accounted for.

For a large system with multiple components, it is best to schedule one meeting per architect after the kickoff meeting to drill down on each component.

On a whiteboard, have the architect start with a block diagram of the system, sketching the major components and the major data flows between them. Of special importance are any data flows that come from outside the system's process

(or processes) space. Data that comes from outside the program must be considered untrusted. These are the inputs to the system where an attacker can strike.

Here are some external data flows:

- Network I/O
- Remote procedure calls (RPCs)
- Querying external systems: domain name system (DNS), database lookups, Lightweight Directory Access Protocol (LDAP)
- File I/O
- Registry
- Named pipes, mutexes, shared memory, any OS object
- Windows messages
- Other operating system calls

Have the architect describe the data flow's purpose and what processing occurs on the data after it enters the system.

- Is there any input validation? Is it white list or black list? White list validation checks the input against a known "good input" list and accepts it if it is in the list. Black list validation checks the input against a known "bad input" list and rejects it if it is in the list.
- Does any authentication or session management operate in conjunction with the data flow? Processing that occurs after authentication is lower-risk.
- Is there any anti-denial-of-service protection, such as throttling or resource protection?

Draw detailed diagrams containing all the information you learn during the architect session. If design documentation contains additional information, you can certainly use that, but you should check with the architect to make sure the information isn't out of date. If it is a large system, it is best to schedule the architect sessions over many days. This gives you a chance to follow up on the information you learn in one session by looking at design documentation, reading source code, or doing runtime inspection.

Runtime Inspection

What the architects describe doesn't always match reality. It is always a good idea to verify the expected implementation with a real look under the hood. Many developers working with high-level system APIs may not even realize that they are accessing information over the network or creating local system objects.

Wysopal_Rev-Chapter 04.qxp 10/5/2006 12:28 PM Page 76

Application footprinting is the process of discovering what system calls and system objects an application uses.

You may be familiar with network footprinting, in which you discover a network's topology, what devices are connected to the network, what operating systems (OSs) the devices are running, what OS versions are in use, what network ports are open, and what applications are communicating over those ports.

Application footprinting uses similar inspection techniques but focuses on just one application. You want to know how that application receives input from its environment via operating system calls. You want to find out what OS objects the application is using, such as

- Network ports
- Files
- Registry keys

You can use the application footprinting process to validate and add to the information gathered by interview and documentation inspection. If you do find some discrepancies, it is a good idea to tell the component architect immediately. You should then do some research to understand where that input fits into the design.

Windows Platform

Several excellent free Windows inspection tools are available from Sysinternals (http://www.sysinternals.com). The most important one for your needs is Process Explorer, shown in Figure 4-1. Process Explorer shows you what DLLs and handles a process has open. A handle is created when a process opens an OS object. This list is all-inclusive for any moment in time. As the program executes, handles are created and destroyed, such as when a configuration file is opened, read, and then closed. So Process Explorer is good at discovering the OS objects that the process uses for long periods of time. For shorter-lived objects or object accesses, you use other tools.

Process Explorer can view the following handles:

- Desktop
- Directory
- Event
- File
- Key
- KeyedEvent
- Mutant

Wysopal_Rev-Chapter 04.qxp 10/5/2006 12:28 PM Page 77

- Port
- Process
- Section
- Semaphore
- SymbolicLink
- Thread
- Timer
- Token
- WindowStation

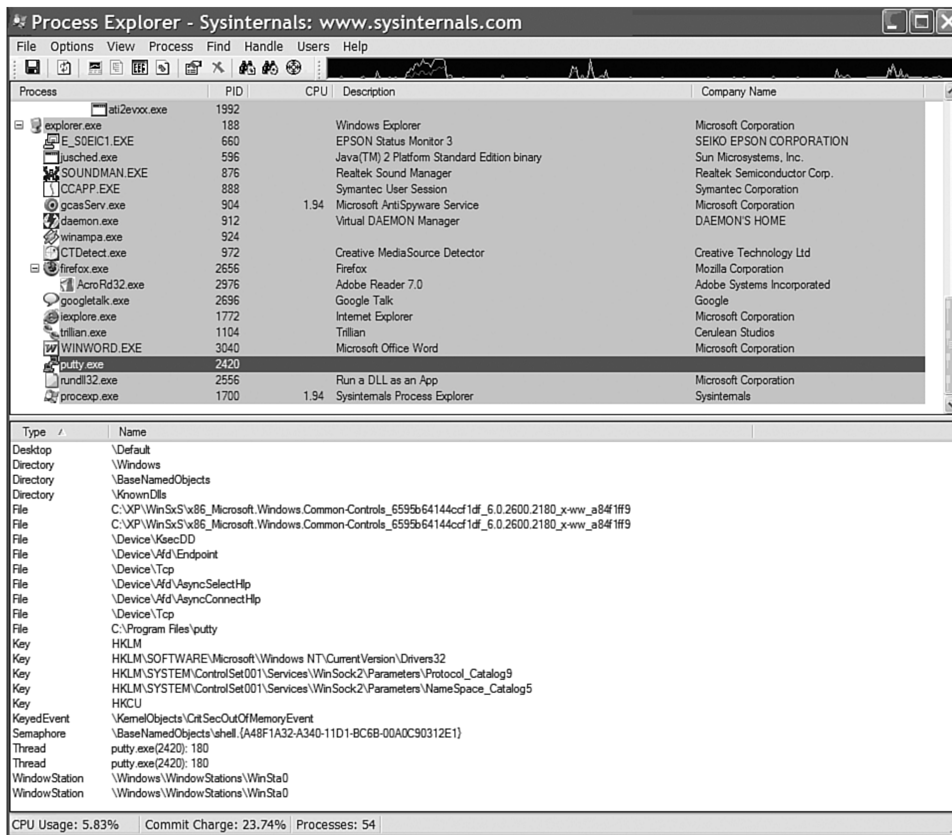

Figure 4-1 Process Explorer

Pay special attention to the File handles. Not only does this include files in the file system, it also includes network devices such as \Device\Rawip\1,\Device\

Wysopal_Rev-Chapter 04.qxp 10/5/2006 12:28 PM Page 78

Tcp, and \Device\Ip. These network device handles tell you that the process is performing network I/O. This is risky behavior that you need to scrutinize during your threat modeling. All the other handles are objects local to the system. These are important if the process is running at a higher privilege level than a normal user and you need to evaluate the risk of a privilege escalation attack.

To see what privilege level a process is running in Windows, use the Task Manager. Under the Processes tab you will see a username associated with each process. If the username is SYSTEM or LOCAL SERVICE, the process is running with more privileges than a normal user. For UNIX systems you can use the `ps -aux` command to display the list of processes and the associated usernames to see the privileges of the running programs. On UNIX the root user has extra privileges.

```
vikki:weld {138} ps -aux
USER     PID   %CPU %MEM VSZ  RSS  TT  STAT STARTED  TIME     COMMAND
weld     23340 0.0  0.1  328  204  p2  R+   4:59PM  0:00.00  ps -aux
root     3816  0.0  0.1  148  460  ??  Is   6Oct05  0:03.19  syslogd: [pri
_syslogd 28917 0.0  0.1  172  480  ??  S    6Oct05  17:47.85 syslogd -a /v
root     21459 0.0  0.1  388  284  ??  Ss   6Oct05  7:55.38  pflogd
www      1990  0.0  0.5  996  1564 ??  Ss   6Oct05  5:22.98  httpd: parent
root     13100 0.0  0.3  280  932  ??  Is   6Oct05  1:39.89  /usr/sbin/ssh
root     22198 0.0  0.1  420  304  C0- I    6Oct05  0:00.01  /bin/sh /comm
root     28060 0.0  0.2  268  556  ??  Is   6Oct05  0:31.88  cron
root     14920 0.0  0.1  60   464  C0  Is+  6Oct05  0:00.01  /usr/libexec/
root     10081 0.0  0.2  88   484  C1  Is+  6Oct05  0:00.01  /usr/libexec/
root     31239 0.0  0.1  104  472  C2  Is+  6Oct05  0:00.01  /usr/libexec/
root     29542 0.0  0.1  112  476  C3  Is+  6Oct05  0:00.01  /usr/libexec/
root     27973 0.0  0.1  52   468  C5  Is+  6Oct05  0:00.01  /usr/libexec/
root     14639 0.0  0.1  100  396  ??  S    6Oct05  9:23.28  svscan /servi
root     31448 0.0  0.1  36   320  ??  I    6Oct05  0:00.01  readproctitle
root     8158  0.0  0.1  92   392  ??  I    6Oct05  0:00.65  supervise qma
root     22141 0.0  0.1  44   376  ??  I    6Oct05  0:00.94  supervise log
root     4273  0.0  0.1  96   376  ??  I    6Oct05  0:00.66  supervise qma
root     32755 0.0  0.1  100  376  ??  I    6Oct05  0:00.60  supervise log
qmaill   4751  0.0  0.1  56   368  ??  I    6Oct05  0:00.01  /usr/local/bi
qmails   4004  0.0  0.2  188  548  ??  S    6Oct05  47:10.08 qmail-send
qmaill   7754  0.0  0.1  104  412  ??  I    6Oct05  1:13.57  /usr/local/bi
qmaild   25150 0.0  0.1  116  468  ??  I    6Oct05  2:21.00  /usr/local/bi
```

Wysopal_Rev-Chapter 04.qxp 10/5/2006 12:28 PM Page 79


```
qmaill    26241 0.0 0.1 68  428  ?? S  6Oct05  20:55.36 splogger qmai
root      24711 0.0 0.1 116 416  ?? I  6Oct05  5:10.66  qmail-lspawn
qmailr    8474  0.0 0.1 132 404  ?? S  6Oct05  21:11.60 qmail-rspawn
qmailq    30726 0.0 0.1 68  412  ?? I  6Oct05  4:01.28  qmail-clean
root      14547 0.0 0.4 352 1288 ?? Is Sun05PM 0:00.08  sshd: weld [p
weld      8927  0.0 0.4 328 1152 ?? I  Sun05PM 0:00.93  sshd: weld@tt
weld      1196  0.0 0.1 348 316  p0 Is Sun05PM 0:00.03  -csh (csh)
```

If an attacker finds a vulnerability in a program running with privileges higher than a normal user, he can run code and gain the privileges of the higher-privileged program. This is why programs running with enhanced privileges demand extra scrutiny. Another angle on this is finding resources on the system that will be used by other users running with different privileges. The attacker tampers with that resource and waits for a different or higher-privileged user to consume that resource.

Process Explorer tells you only if the process you are footprinting is using the network. To see the details of what TCP/IP ports are being used, whether any of them are listening on the network, and where any active connections are going, you need another program—TCPView (see Figure 4-2).

Other useful tools for Windows footprinting are Filemon and Regmon from Sysinternals. These tools intercept calls to the Windows file and Registry APIs and record the parameters a program sends to the OS. This enables you to see every open, read, write, and delete an application makes. Anytime a program reads from a file or Registry, a threat occurs that you need to model, evaluate, and possibly mitigate.

OLEview lets you browse any of an application's COM interfaces. COM objects marked "Safely Scriptable" are the riskiest because Internet Explorer opens an HTML page that can instantiate and use JavaScript/VBScript to send commands to that COM object. This can allow a malicious Web site to run commands on a client machine. Chapter 11 shows you how to use OLEview to identify safely scriptable COM objects and how to enumerate and test their interfaces.

UNIX Footprinting

You should know about a few tools for application footprinting a UNIX program. lsof (list open files) is similar to ProcessExplorer. For each process, it lists all the file handles open across the entire OS. They are sorted by process, so you can quickly zoom in on the file handles you are interested in.

Here is part of the output for lsof from a system running Apache 1.3:

Wysopal_Rev-Chapter 04.qxp 10/5/2006 12:28 PM Page 80

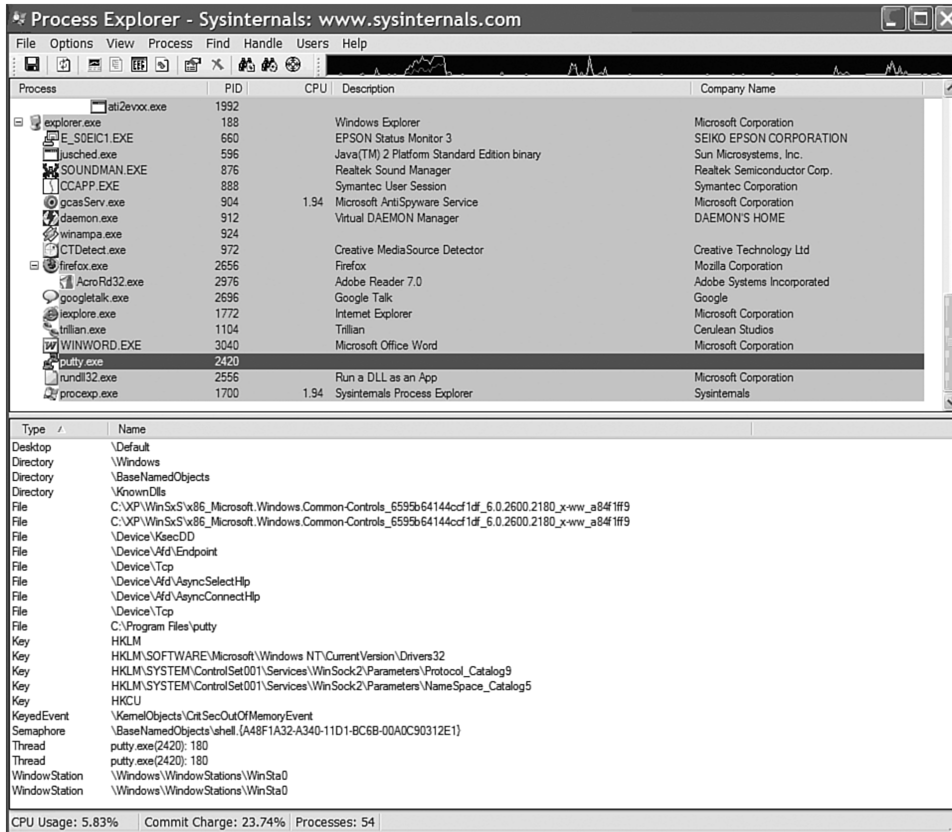

Figure 4-2 Sysinternals TCPView

These are the open file handles for the httpd process 2785. Apache (`httpd`) spawns multiple processes to handle requests, but you need to look at only one of them. The NAME field tells you what file, device, or sockets the httpd process is using. The most important part is the last two lines, where you see that httpd is listening for TCP connections on ports 81 and 80 (www). You definitely want to threat-model what is coming in over those ports.

strace, ktrace, and truss are debugging tools that print a trace of all the system calls made by an application. strace is available on Linux. ktrace is available on FreeBSD, NetBSD, OpenBSD, and OS X. truss is available on Solaris. From the tracing tool output, you can determine what files and network sockets a program is reading from and even the content of the read calls. As you might imagine, the output is very verbose, and most programs make thousands of system calls. As an example, we show you how to use ktrace on MacOS X to log system calls.

Wysopal_Rev-Chapter 04.qxp 10/5/2006 12:28 PM Page 81


```
COMMAND  PID   USER    FD   TYPE      DEVICE    SIZE/OFF       NODE    NAME
httpd   2785   www   cwd   VDIR          0,0         512    123051   /     — htdocs/misc
httpd   2785   www   rtd   VDIR          0,0         512      1920   / (/dev/wd0a)
httpd   2785   www   txt   VREG          0,0      656616    714656   /usr/lib/libc.so.30.1
httpd   2785   www   txt   VREG          0,0      226445    714660   / (/dev/wd0a)
httpd   2785   www   txt   VREG          0,0       30835    737624   /usr/libexec/ld.so
httpd   2785   www   txt   VREG          0,0     1033529    714657   / (/dev/wd0a)
httpd   2785   www   txt   VREG          0,0       89479    714292   / (/dev/wd0a)
httpd   2785   www   txt   VREG          0,0      606168    746931   / (/dev/wd0a)
httpd   2785   www   txt   VREG          0,0        4154   1741463   /var/run/ld.so.hints
httpd   2785   www    0r   VCHR          2,2         0t0   1383509   /dev/null
httpd   2785   www    1w   VCHR          2,2         0t0   1383509   /dev/null
httpd   2785   www    2w   VREG          0,0    22578954      5834   / (/dev/wd0a)
httpd   2785   www    3w   VREG          0,0   261063135      5850   / (/dev/wd0a)
httpd   2785   www    4u   VCHR         70,0         0t0   1383507   /dev/crypto
httpd   2785   www   15w   VREG          0,0    22578954      5834   / (/dev/wd0a)
httpd   2785   www   16u   IPv4   0xd0c93000         0t0       TCP   www.vulnwatch.org:81 (LISTEN)
httpd   2785   www   17u   IPv4   0xd0c93164         0t0       TCP   www.vulnwatch.org:www (LISTEN)
```

Wysopal_Rev-Chapter 04.qxp 10/5/2006 12:28 PM Page 82

To trace a program, you run `ktrace` with the process ID of the application you want to analyze and specify what type of data you want logged. For this example, we logged the system calls, I/O, and nami for the main sshd process (7564) on a system. The command line is

```
ktrace -p 7564 -t cin
```

When you are done tracing, you stop the trace with this command:

```
ktrace -C
```

and then you look at the results with

```
kdump
```

The following is small section of `ktrace` output from a MacOS X system running the sshd process:

```
7564 sshd     CALL  open(0x20ea5640,0,0x1b6)
7564 sshd     NAMI  "/etc/hosts.allow"
7564 sshd     RET   open -1 errno 2 No such file or directory
7564 sshd     CALL  open(0x20ea5651,0,0x1b6)
7564 sshd     NAMI  "/etc/hosts.deny"
7564 sshd     RET   open -1 errno 2 No such file or directory

<some lines deleted>

7564 sshd     CALL  write(0x6,0x3c01a600,0x17)
7564 sshd     GIO   fd 6 wrote 23 bytes
      "SSH-1.99-OpenSSH_3.7.1
      "
7564 sshd     RET   write 23/0x17
7564 sshd     CALL  read(0x6,0xcfbfb36c,0x1)
7564 sshd     GIO   fd 6 read 1 bytes
      "S"
7564 sshd     RET   read 1
7564 sshd     CALL  read(0x6,0xcfbfb36d,0x1)
7564 sshd     GIO   fd 6 read 1 bytes
      "S"
7564 sshd     RET   read 1
```

Wysopal_Rev-Chapter 04.qxp 10/5/2006 12:28 PM Page 83


```
7564 sshd      CALL   read(0x6,0xcfbfb36e,0x1)
7564 sshd      GIO    fd 6 read 1 bytes
      "H"
```

The first column is the process ID that we are tracing, the second column is the process name, the third is the type of trace logged, and the rest is the data for that trace item. CALL is a system call, NAMI is the conversion of a system vnode to a human-readable pathname, and GIO is an input/output operation.

You can see that the first operation in this output is a call to open a file named /etc/hosts.allow. In this case the return code specifies that the file doesn't exist. The same operation is done for the file /etc/hosts.deny. Then the process writes 23 bytes to file descriptor (`fd`) 6. This happens to be a network socket. The 23 bytes are `SSH-1.99-OpenSSH_3.7.1\n` (`\n` is the newline character). Then the process loops, reading 1 byte at a time from `fd 6`. You can see the contents it is reading. The first 3 bytes are SSH. This looks like the protocol handshake when an SSH client first connects to the sshd server.

As you can see, ktrace is very verbose, but it doesn't miss anything. Searching through the output for the string `NAMI` can be a useful way to see all the files a process uses.

Part II discusses additional tools and techniques for dynamic application footprinting.

Finalizing Information Gathering

At this point you should have enumerated all the application's entry points. Note whether the entry point is local or remote, if it is encrypted, what protocol is used (if any), what type of interface it is (HTTP, RPC, COM, file I/O), and whether the interface has authentication and session management.

The Modeling Process

After you have collected the design data and discovered the application's attack surface, you can move on to trace the data flows through the application's components to uncover areas of the program that are the highest risk. This is where you will later focus your testing. You may also uncover security weaknesses that are obvious by inspection, such as an administrative interface that doesn't perform authentication or a session identifier that is not encrypted.

The modeling process described here is a process optimized for prioritizing testing and discovering areas to test. It is shorter than a full threat-modeling

process that should be used during program design to determine design flaws. A good source for a full threat-modeling process is the book *Threat Modeling* by Frank Swiderski and Window Snyder of Microsoft. Microsoft has followed this up with a new process called Application Consulting and Engineering (ACE) Threat Analysis and Modeling (version 2.0).[1]

The goal of their new process is to make it easier for people who are not security experts to perform threat modeling. The threat-modeling process has four main steps:

1. Identify threat paths.
2. Identify threats.
3. Identify vulnerabilities.
4. Rank/prioritize the vulnerabilities.

Identifying Threat Paths

The first stage of the process is to identify the highest-level risks and protection mechanisms for the application you will test. First, overall security strengths of the application platform and implementation language are noted because these will be relevant throughout the threat-modeling process. Sample strengths are the use of a managed code language such as Java or C#, running on a multiuser OS platform, or the use of encrypted network communications.

Next, you need to identify the different user access categories. Every application has at least anonymous access—even those that perform authentication. You mustn't forget that although the user is authenticating, he is interacting anonymously with the application. Table 4-1 shows sample user access categories ranked from highest-risk to lowest-risk.

Table 4-1

Access Categories

Risk	Access Category
Very high	Anonymous remote user
High	Authenticated remote user with file manipulation capability
Medium	Authenticated remote user
Low	Local user with execute privileges
Very low	Administrative local user

Most client/server applications have similar user access categories. Every application needs security testing performed on the threat paths that anonymous

remote users and authenticated remote users can access. Where higher-security assurance is required, the threat paths that local users and even administrative users can access need to be tested.

Even programs that don't listen on the network for their data input can have anonymous remote user access. Consider a media player that plays MP3 files. An input file may be an MP3 file that you created from a CD, or it could be a file that is downloaded from the Internet. In the Internet download case, the file input comes over a threat path that has an anonymous remote user access category. Any program that opens files that are commonly transferred over the Internet needs security testing.

The next step in identifying the threat paths starts with a data flow diagram. Data is how the user (or, in this case, the attacker) interacts with the application. For this reason, it is important to think in terms of a data flow to do threat analysis.

A data flow diagram is a block diagram. The squares are the users or other external systems, called *entities*; the circles are the different software components that do processing, called *processes*; and the lines between them are the *data flows*. Figure 4-3 is a simple high-level data flow describing an instant messenger (IM) server.

The two IM users are anonymous until they send their credentials to the Authenticate process. Then they can interact as authenticated users with the Manage Account and IM Chat Engine processes. Even with this high-level view, you can start to see the processes within the program that are at higher risk because they can be attacked anonymously by anyone on the network. When a process in the data flow diagram is many different data flows, it should be separated into subprocesses and sub-data flows. The Manage Account process in the example is a prime candidate for that.

To highlight the attack entry points, the data flow diagram is next marked with process and machine boundaries and the user access categories. Whenever data crosses a process or machine boundary, it moves from a lower-privileged security domain to a higher-privileged domain. This is noted as threat path of high risk.

In Figure 4-4, dashed lines have been added to show the boundaries between machines that communicate as part of an IM chat. The three circles in the middle are the processes that make up the IM chat server. It is communicating with two entities that are outside the machine executing the chat server processes.

For some programs where a threat is a local privilege escalation vulnerability, it is important to understand where components interact across process boundaries. In those cases, use lines to show where process boundaries are crossed.

The marked-up data flow diagram is boiled down to a list of threat paths (see Table 4-2). Components in the anonymous remote user's threat path are at the top

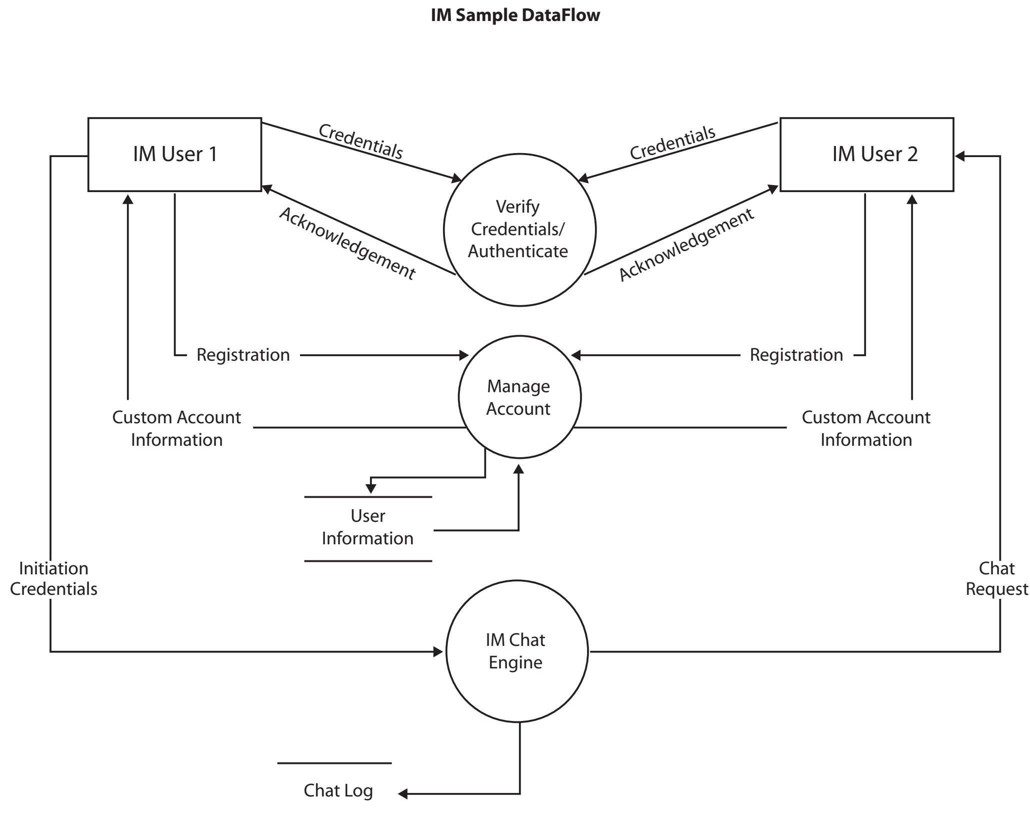

Figure 4-3 IM sample data flow diagram

of the list. Components behind multiple layers of authentication are toward the bottom. This list of threat paths is then processed one by one for the individual threats along that path.

Table 4-2

Threat Paths

Access Category	Threat Path
Anonymous remote user	Credentials
Anonymous remote user	Registration
Authenticated remote user	Initiation
Local user	User information

You now have a prioritized list of threat paths to investigate. The Credentials and Registration processes should be tested first because they are the program's highest-risk areas. Next the Initiation process, which is part of the IM Chat Engine

Wysopal_Rev-Chapter 04.qxp 10/5/2006 12:28 PM Page 87

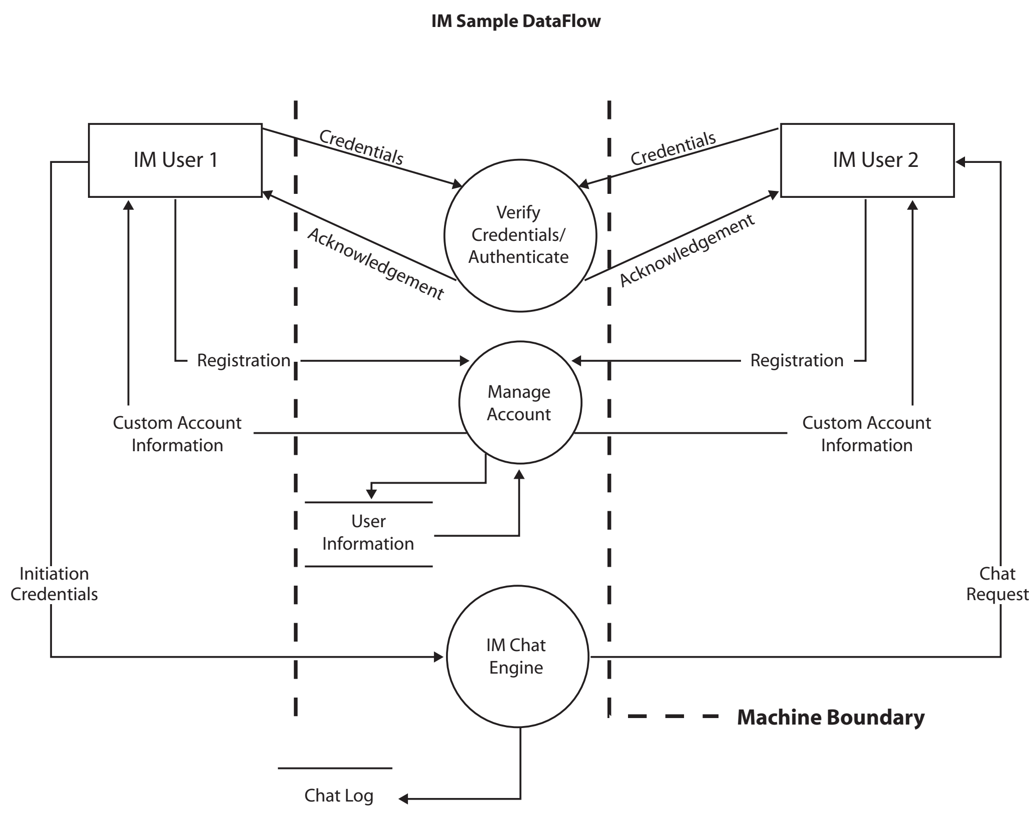

Figure 4-4 IM sample data flow diagram with trust boundaries

process, is tested. The final threat path, where the User Information is read by the Manage Account process, is low-risk because the access category is a local user. It is local user because an attacker would need to be on the machine to affect this user input. You can see this in Figure 4-4 because the data flow from the User Information to the Manage Account process doesn't cross a machine boundary. It is probably better to spend more time worrying about the high-risk threat paths than to bother with this one.

Identifying Threats

For each threat path, the next step is to go a level deeper to identify the processing that is performed along the threat path and enumerate the individual threats to that processing. Starting with the highest-risk threat path, the processing performed along the path is analyzed. The following is a list of questions to ask for each processing component along the path:

Wysopal_Rev-Chapter 04.qxp 10/5/2006 12:28 PM Page 88

- What processing does the component perform?
- How does it determine identity?
- Does it trust data or other components?
- What data does it modify?
- What external connections does it have?

Then high-risk activities need to be marked along the path:

- Data parsing
- File access
- Database access
- Spawning of processes
- Authentication
- Authorization
- Synchronization or session management
- Handling private data
- Network access

Boil down this information to write out the threats along each threat path. This list of threats will be used to drive the vulnerability-finding process. An example of a threat would be if the authentication process accepts data from anonymous users and passes it to the SQL server. This is not necessarily a vulnerability. If the program properly validates input, it has properly mitigated this threat. The next step is to find out whether this is the case.

For example, the threats to the IM server diagrammed in Figure 4-4 are as follows:

- IM User Credentials data could be malformed, causing processing errors in the Verify Credentials/Authenticate process. The processing errors could lead to

  Corruption of the credential data

  Remote execution of code

  Denial of service (DoS) to the Authentication process

- IM User Registration data could be malformed, causing processing errors in the Manage Account process. The processing errors could lead to

  Corruption of the user information

  Remote execution of code

  DoS of the Registration process

Wysopal_Rev-Chapter 04.qxp 10/5/2006 12:28 PM Page 89

- IM User Initialization Credentials data could be malformed, causing processing errors in the IM Chat Engine process. The processing errors could lead to
  - Corruption of the chat log
  - Remote execution of code
  - DoS of the Chat Engine process
  - Malformed data being sent to another IM user
- User Information data could be malformed, causing processing errors in the Manage Account process. The processing errors could lead to
  - Corruption of the user information
  - Remote execution of code
  - DoS of the Registration process

Identifying Vulnerabilities

As soon as you know the threats to the high-risk components, the next step is to find any actual vulnerabilities that may exist in those components. A threat becomes a vulnerability when the designers fail to build any security features into the application that mitigate that threat. Some of the security mitigations to look for are data validation testing, resource monitoring, and access control for critical functions.

The vulnerability hunt can branch in several directions at this point: detailed security design review, security code review, or security testing. The security design review is best at finding design-level vulnerabilities. The security code review and security testing are best at finding coding errors where the programmer did not follow secure coding practices. These three methods of security analysis each benefit from the prioritization done during the threat-modeling procedure.

Because our focus is security testing, you will find the vulnerabilities by testing. Part II of this book gives you recipes to test for vulnerabilities. The sample threat identified earlier, "the authentication process accepts data from anonymous users and passes it to the SQL server," prompts you to test for SQL injection vulnerabilities during the Verify Credentials process of the sample IM program.

Ranking the Risk Associated with a Vulnerability

A useful technique for ranking a threat's severity is to use the DREAD model. It was introduced by Michael Howard and David Leblanc in *Writing Secure Code*, Second

Edition. DREAD stands for Damage potential, Reproducibility, Exploitability, Affected users, Discoverability. You rank a threat using DREAD as follows:

- Damage potential

  The extent of the damage if a vulnerability is exploited.
- Reproducibility

  How often an attempt at exploiting a vulnerability works.
- Exploitability

  How much effort is required? Is authentication required? 1—authentication required, 2—no authentication but hard-to-determine knowledge required, 3—no authentication and no special knowledge required.
- Affected users

  How widespread could the exploit become? 1—only rare configurations, 2—a common case, 3—the default or most users.
- Discoverability (you might not want to use this one if you believe all vulnerabilities are eventually discovered)

  The likelihood that the researcher or hacker will find it.

These are subjective, so just use low (1), medium (2), or high (3) for each category. Add up the numbers assigned to each category to create the DREAD rating.

When it's time to fix vulnerabilities, higher DREAD ratings should come first because they have a higher likelihood of damaging the system.

Chapter 12, "Determining Exploitability," describes another model for ranking vulnerabilities that uses the following factors: time, reliability/reproducibility, access, and positioning.

Determining Exploitability

If a system is deployed in production or is a product in the hands of many customers, it is expensive to remediate the vulnerabilities, regression test, and redeploy a new version or issue a patch. Because of this cost, you should be sure that the vulnerability can be exploited. When software is in development, it is typically easier to just fix an issue that looks likely to be exploitable than to take the time to determine if it actually is. This is because determining exploitability can be difficult and time-consuming. Chapter 12 is devoted to the topic of determining exploitability. If you find vulnerabilities in deployed code, you will want to examine that chapter.

Wysopal_Rev-Chapter 04.qxp 10/5/2006 12:28 PM Page 91

Threat-Modeling Summary

1. Conduct information gathering.
   a. Schedule a kickoff with the team.
   b. Meet with the architects.
2. Review the block-level high-level diagram.
3. Interview the component designers.
4. Review or develop component-level diagrams.
5. Conduct a runtime inspection.
6. Finalize information gathering.
7. Conduct the modeling process.
   a. Identify threat paths.
   b. Identify threats.
   c. Identify vulnerabilities.
   d. Prioritize vulnerabilities.
8. Determine exploitability (see Chapter 12).

Endnote

1. http://msdn.microsoft.com/security/securecode/threatmodeling/acetm/.