Realtime publishers

The Definitive Guide™ To

Building Code Quality

sponsored by

Don Jones

Copyright Statement

Chapter 5: Addressing Performance and Security Problems

Performance and security: two aspects of your application that users are perhaps most likely to *perceive* as poor quality. Sure, users might be irritated by a recurring bug, but bugs can often be worked around, where poor performance becomes an unwanted and unwelcome part of daily life. Security is likewise a great way for your application to earn a reputation for poor quality: Either security is cumbersome and gets in users' way, or it isn't complete and ends up causing a major incident within users' organizations.

Too many development projects focus on performance and security last, if at all, but they are every bit as important to your application as making sure all the necessary end-user functionality is properly implemented. In fact, while security and performance are commonly referred to as *non-functional requirements,* you should definitely consider them to be a core part of your application's general functionality—things users will notice if they're not done properly.

In this chapter, we'll look at common sources of both security and performance problems, and examine specific ways to help eliminate those problems in a variety of scenarios. We'll also look at general techniques for measuring the results of your security and performance tuning efforts so that you can begin to develop a feel for the exact amount of effort required to generate a specific amount of additional value in your applications.

Common Sources of Performance Problems

"I'm sorry, sir, can I put you on hold for a minute? The computer is being slow today." Everyone who has ever called a customer service agent on the phone has heard this phrase more than once, and it's perhaps one of the things that end users most commonly perceive as "poor quality" in an application. Major software developers—such as Microsoft—spend millions to provide at least a better *perception* of performance, such as working to make the Windows desktop appear quickly, even though other bits of the operating system (OS) might still be starting in the background.

So where do performance problems originate?

Inefficient Coding or Algorithms

Inefficient coding is probably one of the most common performance problems I've seen in the projects I've worked on. One reason, I think, stems from the fact that developers over the years have become increasingly abstracted from the hard work that goes on to access various resources. For example, in the "old days," retrieving information from a database meant pretty laborious coding: You had to set up a database connection, make database library calls, wait for callbacks from the database as data began streaming over, read that data from input buffers, and so forth. Today, with technologies such as the .NET Framework's rich data classes and LINQ, retrieving data takes a couple of statements, as the hard work is all done under the hood. But the *hard work is still being performed,* and poor coding can easily result in performance problems.

For example, consider this pseudo-code:

```
Rows = Get_Rows_From_Database(query)
For Each (Row in Rows)
  If (Row.Column = Value)
    Update_Row_In_Database(query)
  End
Next
```

Inefficient? Absolutely. Pulling over a few thousand rows, enumerating them in a client application, and then sending single-row changes back to the database would be considered abusive by most experienced developers. A single database query could accomplish the same thing, but with less processing on the client and with far more efficient processing on the database server.

There's a joke amongst experienced software developers that systems administrators—folks who tend to write short scripts to help automate administrative tasks—will write five lines of code, run it, and if it works, call it a day and go home. Good developers, however, will spend two additional days trying to reduce those five lines to three. Why? Efficiency: Less code may mean less processing, and less processing tends to lead directly to better performance (obviously, "fewer lines" doesn't necessarily equal "faster," but it *is* a joke).

The point here is that nearly *any* working code can be optimized for better performance. Algorithms can be tuned to be more efficient, and code itself can be reduced or rewritten to work more effectively.

Note that Microsoft's own code, in the .NET Framework, is no exception—although you obviously can't make corrections to that code. Just be open to the possibility that portions of the built-in .NET Framework might not be as efficient, performance-wise, as some third-party replacements. Using alternative database drivers, for example, might gain you significant performance gains in an application that makes heavy use of database connections; using a different set of charting components might offer better performance than your existing charting components can provide.

Efficient Runtime vs. Efficient Design-Time

Many developers tend to choose components for their ease-of-coding rather than for performance or even for intrinsic security. There's certainly value in working with components that are easy to code because they make the overall application easier to work with. But when selecting components—whether user interface (UI) components, I/O components, or anything else—do your homework and consider performance and security as well. A component that runs significantly slower than its competition will inflict that performance hit on every end user who uses the application; the better-performing competitor might have a more complex coding model, but that coding inefficiency will only be experienced during the project's development.

Excessive Code Path Length

Modularization—encapsulating pieces of code for easier re-use—is fantastic. So is database normalization, which is in essence a form of modularization. However, every good database designer recognizes the necessity of *de*-normalization—breaking the rules of normalization in order to achieve a better degree of performance or to realize some other important goal. Modularization should be viewed the same way: It's fantastic insofar as it encourages code reuse and helps make debugging easier, but as developers, we need to be willing to back off from modularization when it starts to affect other goals, such as performance.

The code path is the number and relationship of individual code objects, functions, and/or proxies involved in executing a method. Consider a deeply-modularized application where Module A calls Module B, which in turn calls Module C to perform some task. Module C relies on Module D, which needs to call Module E, and so forth. At some point, the application is spending more time traversing code boundaries and managing the call stack than it is doing any useful work, and you've created a performance problem. In these cases, it might be better to strategically flatten the application a bit to achieve better performance. The same kind of deeply-modularized structure is seen in applications that leverage object-oriented techniques such as inheritance: Object A inherits from Object B, which inherits from Object C, and so forth—creating a deep code path.

In the database world, for example, designers know that any query that has to join more than seven to nine tables is probably excessive, and that some de-normalization may be called for. In the broader world of software modularization, however, there's no hard-and-fast rule, which is why you'll need to become adept at using performance-tracing tools to determine where bottlenecks exist in your application, at examining those bottlenecks, and at deciding whether a bit less modularization might help improve performance.

Note

Chapter 1 spent some time looking at quality metrics, some of which focused on factors such as code depth and code complexity. Intelligently reducing the code path in your application can also realize other benefits, such as reducing debugging complexity.

Resource Bottlenecks

Developers often work in a perfect world: Their own, high-end development computer, where they're the only user and where there is little to no contention for resources. In the real world, where the application will run, however, users have less-powerful computers and are all trying to use the application at once—which can create unforeseen resource bottlenecks.

In modern computing, there are essentially four key system resources that will be a bottleneck for your application:

- Memory
- Disk
- Network
- Processor

On a 32-bit Windows-based computer, each application is normally assigned a 2GB chunk of memory, even though the computer may not have that much space available (Windows can be configured to offer more to applications using Address Windowing Extensions—AWE—and 64-bit editions of Windows can offer much more). Any difference between the amount of space offered and the amount of memory actually available is made up from virtual memory stored in a *page file* on disk—meaning that memory bottlenecks can also lead to disk bottlenecks. Also, because disk access times are typically much slower than memory access times, a heavy dependence on the page file *will* reduce application performance.

A *bottleneck* is essentially any condition where one of these resources (or other resources, for that matter) is present in insufficient quantity, and your application performance suffers because of it. Insufficient or inefficient use of memory, for example, may result in heavy use of the page file, which slows Windows as a whole. Insufficient or badly-managed network bandwidth may slow a network-dependent application.

There are two straightforward ways to address these bottlenecks once you find them: Add more of the resource or rewrite your application to need less of the resource. Typically, only the latter solution will be within your purview. Be aware that eliminating one bottleneck will often simply move the "choke point" to another resource: Adding more memory may speed things enough for you to realize that you now need more processor power, for example. The trick is in *finding* the bottlenecks in the first place.

Poor Contention Management

Closely related to resource bottlenecks is the idea of contention, although it actually goes far beyond physical resources. For example, an application that modifies a database may find itself waiting while other users modify the same portion of the database—a wait that users perceive as poor performance. This type of contention can't be resolved simply by adding more resources, so you'll have to focus on coding techniques that help reduce the contention itself. For example, optimizing database access so that less of the database is locked at any one time, optimizing changes to the database so that they occur more quickly, or modifying the application so that changes to highly-contested resources can happen asynchronously are all possible approaches to a problem. To give more detail for this particular example:

- Databases typically try to lock as little data as possible, to avoid contention. However, managing locks on data takes up server resources, so databases will lock more data than is strictly needed if doing so helps cut back on server utilization. So, for example, a user attempting to change a couple of columns in a table might wind up locking an entire row, creating more opportunity for contention. Writing your application to recognize how the server manages resource contention can help you write code that triggers less-broad locking, creating fewer opportunities for contention.
- If you need to lock data in the database, releasing those locks quickly can help reduce the opportunity for contention. You can write code that gets its lock, does its thing, and gets out as quickly as possible so that the resource can be freed up for others to use.
- Asynchronous changes can also help. Rather than everyone attempting to lock and change the same resource at the same time, submit changes to a queue and have a middle- or data-tier component process items from that queue in order. This obviously doesn't work in every data-manipulation scenario, but when it does work, it can improve client application performance because those applications aren't sitting around contesting for access to the same data.

Databases are one of the most commonly-used examples of resource contention simply because they're where so much contention occurs. That doesn't mean other resources can't come into contention, though: Shared files, two threads within the same application waiting on a specific condition, and so forth are all examples of resource contention that can slow applications unnecessarily.

Detect and Resolve Problems—Don't Hide Them

Microsoft's technique of getting the Windows desktop to appear quickly—even while the rest of Windows is still getting itself up and running—has inspired a few application developers to similar efforts. A notable one in my experience was a developer who knew his application was taking a long time to populate a particular screen with information from a database. Rather than working to correct the underlying problem, however, he decided to distract users from the performance problem using what he named "the Mesmerizing Eye." This animated icon would float around the screen while the application churned away, allegedly occupying users' attention and making them ignore the significant lag time. Bad idea: After a few hours with the "Mesmerizing Eye," users disliked the application even more than they had disliked previous versions that simply sat there while retrieving data.

The moral is that poor performance is a sign of poor application quality, whether you choose to try and mask the performance problem or not. Instead of coding your own "Mesmerizing Eye," spend the time analyzing your application's performance problems and correcting them.

Detecting Performance Problems: Single-Tier

In this section, we'll discuss techniques for detecting performance problems in single-tier applications—that is, applications that run entirely on a client or server computer and don't rely heavily on another application tier located elsewhere. However, note that these techniques can also be useful in multi-tier applications for resolving performance problems that are constrained to a single one of the tiers in that application—such as problems entirely within a client application.

Trace Application Operations

Tracing your application's operations—that is, keeping a detailed log of how long each element of your application takes to execute—is the first step to finding where performance problems occur. By logging a "start operation" and an "end operation" timestamp around key tasks within the application, you can detect those tasks that are taking an excessive amount of time to complete, and start thinking about ways to speed those tasks. In other words, tracing helps you focus on the areas where you can achieve the biggest performance improvement by highlighting those areas of the application that are contributing the most to poor performance.

Manually tracing an application typically requires you to add your own trace logging functionality: perhaps a StartOp() and CompleteOp() pair of functions or just a Trace() function that logs a text message and a timestamp to some external log file.

> **Note**
>
> The very act of tracing your application will negatively impact its performance. Be sure to remove or disable trace code prior to deploying the application.

Other, more general trace information can be obtained from performance counters that .NET applications register to report thread utilization, time spent on specific methods or garbage collection, and so forth. You can obtain this information from Windows' Performance Monitor utility and analyze it on your own. Table 5.1 highlights key performance counters.

Performance Counter	Description
Processor (_Total)\% Processor Time	Gives an indicator of the total processor utilization in the machine. A value of 80-85% is acceptable. Lower the better.
.NET CLR Exception\# of Excep Thrown /sec (_Global_)	This is the number of managed code exceptions thrown per second. More exceptions mean more performance degradation. This has to be 0 under normal circumstances.
.NET CLR Interop\# of Stubs	Indicates the number of stubs created by the CLR. This value has to be as low as possible. However, for calls being made from managed to unmanaged code and vice-versa, this counter gives an indication to the interaction between unmanaged and managed code. There is always a performance penalty for such interactions.
.NET CLR Loading\Current Assemblies (_Global_)	Indicates and records the number of assemblies that are loaded in the process. It includes all the AppDomains in the system.
.NET CLR Loading\Rate of Assemblies	Rate at which assemblies are loaded into the memory per second.
.NET CLR Loading\Bytes in Loader Heap	Indicates the number of bytes committed by the class loader across all AppDomains. This counter has to be in a steady state; large fluctuations in this counter indicate there are too many assemblies loaded per AppDomain.
.NET CLR Memory\# Bytes in all Heaps	This counter indicates the number of bytes committed by managed objects. This should be less than the Process\Private Bytes counter. The difference between the same is the number of bytes committed by unmanaged objects.

Performance Counter	Description
.NET CLR Memory\# Induced GC	This counter records the explicit calls to the GC.Collect method. This value should be close to 0.
.NET CLR Memory\% Time in GC	This indicates the percentage of time spent performing the last garbage collect. Values in the range of 5-10% are acceptable. There can be spikes in this counter, but those are acceptable. There is a temporary suspension of all the threads during this activity, so there is a performance overhead. Allocating large strings to cache, heavy string operations, and so forth leave a lot of memory spaces that the GC has to clean up, creating a performance hit.
.NET Data Provider for SQLServer\NumberOfActiveConnections	Number of active connections in the pool that are currently in use.
.NET Data Provider for SQLServer\NumberOfFreeConnections	Number of connections free for use in the pool.
.NET Data Provider for SQLServer\SoftConnectsPerSecond	Indicates the number of connections that are being received from the pool every second.

Table 5.1: A sampling of Performance Monitor counters.

Manually tracing application performance can be a real nightmare, though, which is why there's a robust third-party market in tools that can collect this information automatically, analyze it for you, and provide reports that help you focus on the parts of your application that need improvement. Tools include dynaTrace (dynatrace.com), SpeedTrace (ipcas.com), and VantageAnalyzer (Compuware.com), along with literally dozens of others. Note that application tracing isn't the only capability you're going to want to automate, so give special consideration to products that come as part of a suite or family of application analysis products, as the integration between family members often means less work and more accurate results for you.

Differentiate Application from OS Calls

When analyzing performance, be sure to differentiate between calls made within your own application and calls made to the underlying OS. Although it's entirely possible for the OS to create a performance bottleneck, there is not much you can do about it except to stop using that OS call and to implement an alternative—either something of your own or a different, more efficient OS call.

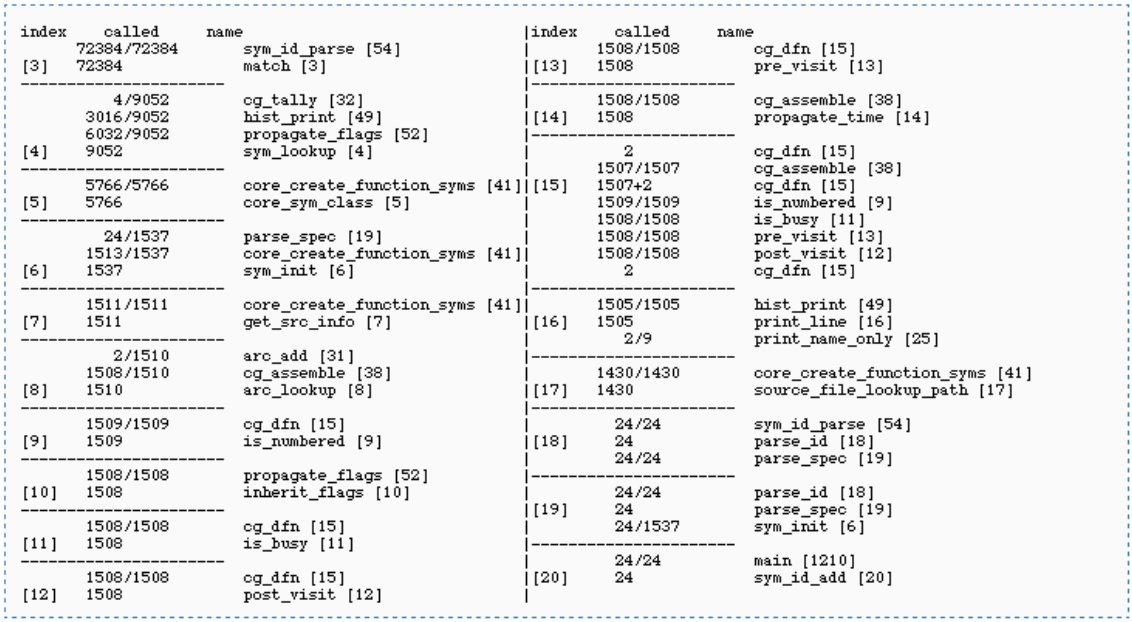
Tip

One root cause of slow OS calls is simply using outdated, deprecated calls. Read Microsoft's documentation carefully, and make sure that the OS calls you're using are of the current generation—rather than using calls that might have been around for years and are comfortable to you but are deprecated and perhaps supported only through underlying "hacks" that keep them working for a bit longer.

Commercial application profiling tools can typically include this information in application traces, making it easy to detect and differentiate between internal and OS calls.

Use of Dynamic Call Graphs to Navigate App Components

A *call graph* (also called a *multigraph*) is a graph or chart that depicts the relationships between subroutines in an application. Each node in the graph represents a procedure and each edge in the graph represents the procedures called. These graphs can be static or dynamic. A dynamic one is a record of the program's actual execution, as monitored by some tool. A static graph is simply a theoretical depiction of every possible run of the program, as generated by some static analysis tool. Both are useful, but for performance purposes, the dynamic call graph helps you trace code depth, detect modules that are making excessive calls (and slowing execution), and detecting the use of specific modules. Figure 5.1 shows a simplistic example of a dynamic call graph.



```
index    called     name                                |index    called     name
      72384/72384       sym_id_parse [54]              |       1508/1508       cg_dfn [15]
[3]   72384             match [3]                      |[13]   1508            pre_visit [13]
----------------------                                 |-----------------------
         4/9052         cg_tally [32]                  |       1508/1508       cg_assemble [38]
      3016/9052         hist_print [49]                |[14]   1508            propagate_time [14]
      6032/9052         propagate_flags [52]           |-----------------------
[4]   9052              sym_lookup [4]                 |          2            cg_dfn [15]
----------------------                                 |       1507/1507       cg_assemble [38]
      5766/5766         core_create_function_syms [41]|[15]   1507+2          cg_dfn [15]
[5]   5766              core_sym_class [5]             |       1509/1509       is_numbered [9]
----------------------                                 |       1508/1508       is_busy [11]
        24/1537         parse_spec [19]                |       1508/1508       pre_visit [13]
      1513/1537         core_create_function_syms [41]|       1508/1508       post_visit [12]
[6]   1537              sym_init [6]                   |          2            cg_dfn [15]
----------------------                                 |-----------------------
      1511/1511         core_create_function_syms [41]|       1505/1505       hist_print [49]
[7]   1511              get_src_info [7]               |[16]   1505            print_line [16]
----------------------                                 |          2/9          print_name_only [25]
         2/1510         arc_add [31]                   |-----------------------
      1508/1510         cg_assemble [38]               |       1430/1430       core_create_function_syms [41]
[8]   1510              arc_lookup [8]                 |[17]   1430            source_file_lookup_path [17]
----------------------                                 |-----------------------
      1509/1509         cg_dfn [15]                    |         24/24         sym_id_parse [54]
[9]   1509              is_numbered [9]                |[18]     24            parse_id [18]
----------------------                                 |         24/24         parse_spec [19]
      1508/1508         propagate_flags [52]           |-----------------------
[10]  1508              inherit_flags [10]             |         24/24         parse_id [18]
----------------------                                 |[19]     24            parse_spec [19]
      1508/1508         cg_dfn [15]                    |         24/1537       sym_init [6]
[11]  1508              is_busy [11]                   |-----------------------
----------------------                                 |         24/24         main [1210]
      1508/1508         cg_dfn [15]                    |[20]     24            sym_id_add [20]
[12]  1508              post_visit [12]                |
```

Figure 5.1: A dynamic call graph example.

Different applications generate different-looking call graphs; better ones of course are easier to read and provide more information. For example, although the graph in Figure 5.1 was generated by a freeware script, commercial applications such as Micro Focus DevPartner offer visual call graphs, which utilize tool tips and other graphical elements to bring more information to you (including details such as memory consumption). Figure 5.2 shows an example of this functionality, which as you can see, makes it much clearer about which module is calling what, and how much of the application's performance and time is spent in each module.

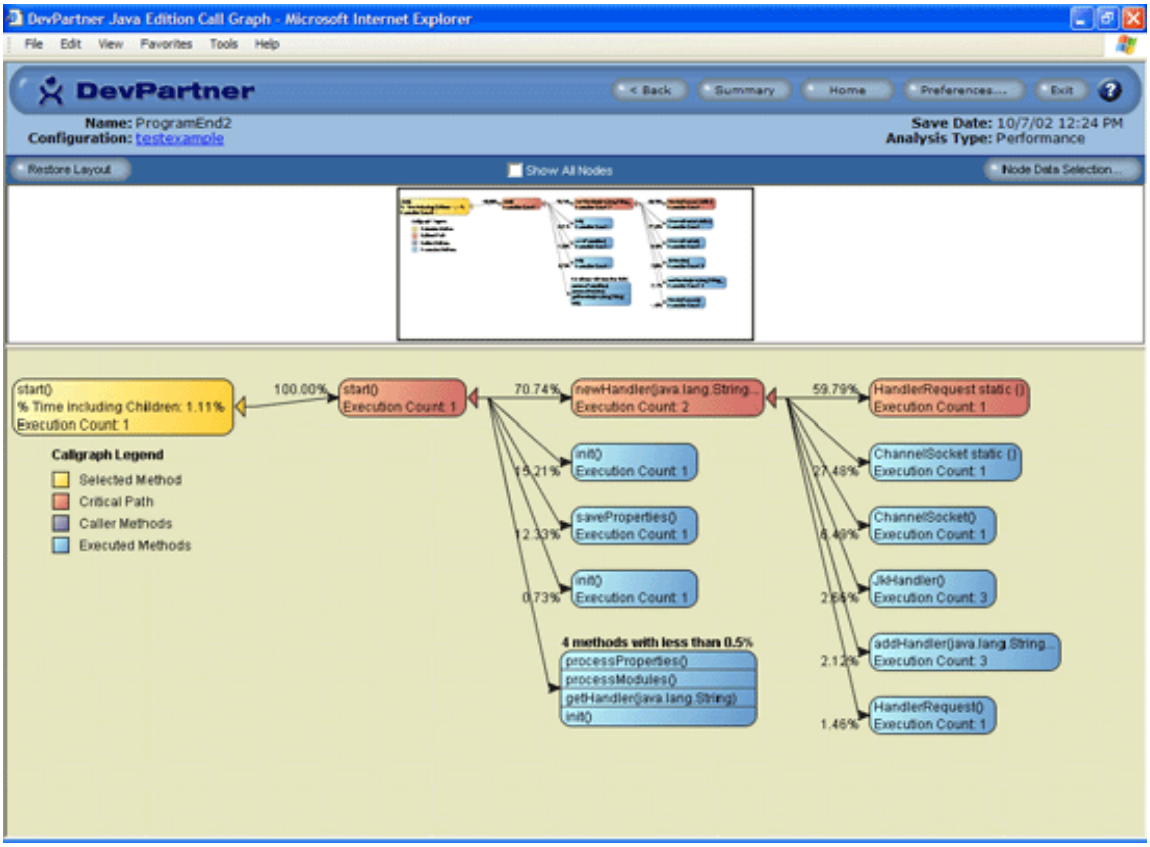


Figure 5.2: An example visual call graph.

Other commercial code analysis tools provide similar functionality. As with most code analysis and profiling tools, these tend to be specific to a given programming language or environment, so if you're working with the .NET Framework and Visual Studio, you'll need to select compatible tools.

Monitor and Analyze Performance Data

Carefully reviewing performance data from your application—using performance counters such as those described earlier in this chapter—can also help spot performance problems. Even better than those raw performance counters, however, are application profilers, which can produce information like the dynamic call charts just discussed.

Profiling tools can provide a number of useful functions:

- Object lifetimes. Continually creating and destroying objects consumes a certain amount of overhead and can fragment memory, leading to more frequent garbage collection in the runtime engine.
- Memory analysis. Although use of memory does not necessarily equal poor performance, using a lot of memory may result in poor performance if users' computers don't have enough memory to go around.
- Application traces. These can identify external calls, including OS calls, as well as list all internal calls and display call graphs.
- Performance bottleneck isolation. This feature can help spot problems across an entire computer, for a given process, in specific code modules, or even at specific lines of source code.

Figure 5.3 shows an example of a full profile output, including a call chart. This example is taken from Micro Focus DevPartner Studio. In it, you get a complete text call list, a call graph, access to run-time information and performance details, and much more.

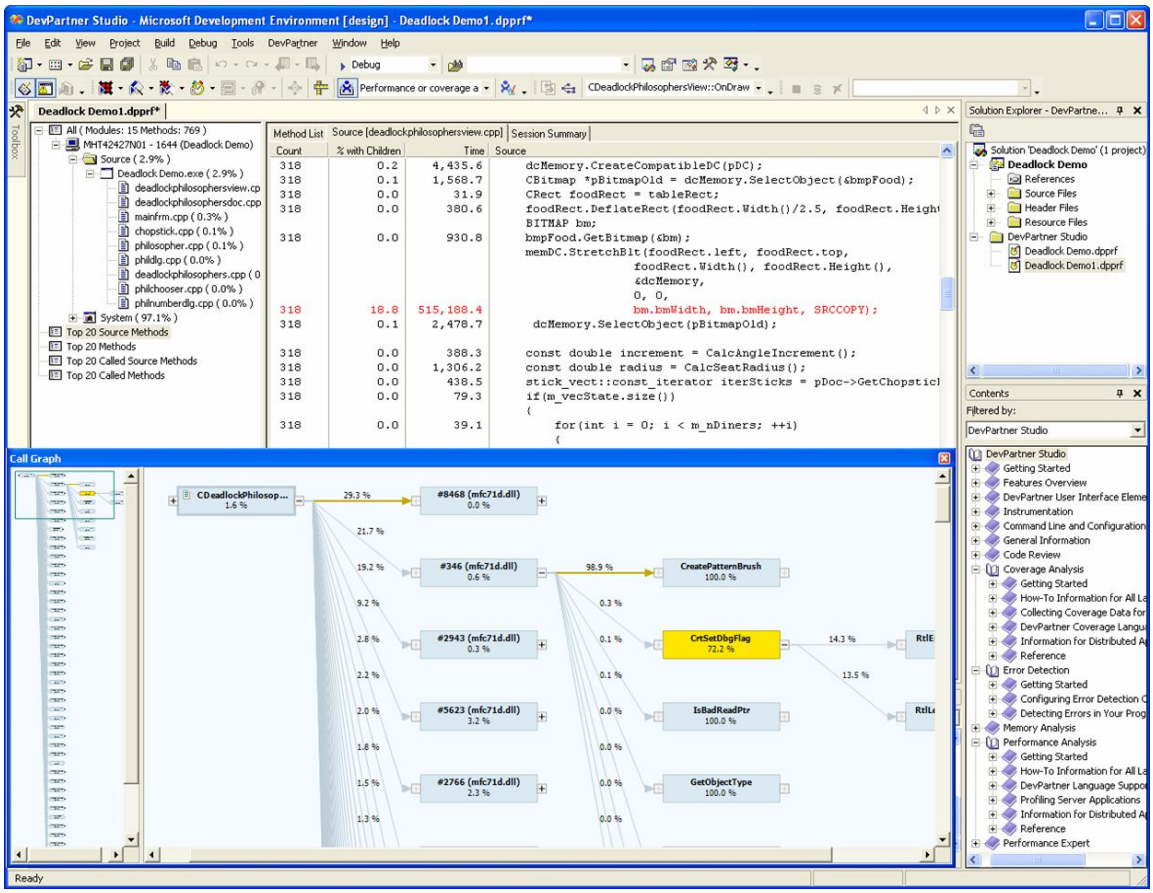


Figure 5.3: An application profiling example.

Some application performance tools—notably, commercial ones—can even help provide "what if" performance scenarios. For example, some features can help analyze your application's performance under more real-world conditions by simulating the load of multiple end users, real-world network conditions to constrict your application, and even less-powerful computer hardware.

Detecting Performance Problems: Multi-Tier

An application allows a user to set parameters and display a chart that the application receives from SQL Reporting Services. The report takes 15 seconds to return once the request is sent (quite a long time to make a CFO sit there waiting). Where is the bottleneck? Is it in the database, trying to pull together a complex query? Is it the Reporting Server, trying to render a complex graphic? Is it the ReportViewer control, trying to draw the resulting chart on the client application screen? How can a developer know where to focus his or her efforts to make this application more responsive?

Multi-tier performance problems are inherently more complicated to track, primarily because the components of the application are so distributed. You also get the added complexity of more physical infrastructure, meaning things like the physical network components can contribute to performance issues, as can supporting applications such as Active Directory (AD). Many of the same techniques used in troubleshooting single-tier performance problems apply but grow more complex and difficult. With multi-tier applications, you're nearly always going to want to rely on a toolset specifically designed for multi-tier performance analysis, such as Micro Focus' DevPartner Server, Crescendo Networks' AppBeat Suite (crescendonetworks.com), or Precise (precise.com).

Multi-tier tools are designed to collect data from multiple remote systems to enable analysis of memory, performance, and other key factors across a large, multi-tier system. In some cases—DevPartner is an example—the main developer product (DevPartner Studio) still does all the work; the "Server License" simply enables new functionality for collecting remote data. Each toolset works differently in this regard, so you should consider this when evaluating solutions and discuss your specific requirements in some detail with potential vendors.

Optimize Tier Performance

A first step in optimizing multi-tier application performance is to simply treat each tier of the application as a standalone application and apply single-tier performance techniques. Doing so will help remove any of the "local" performance issues, which are often easier to detect and resolve than the specific issues brought about by the multi-tier aspect of the application.

You do have to be a bit careful, especially on server tiers that are designed to eventually communicate with multiple clients at once. Some techniques that work well for performance optimization in a single-tier application don't work so well in a server tier; the server can't be "jealous" and simply devote all its resources to one task; be sure to always keep in mind its essentially multi-user nature and troubleshoot and optimize performance accordingly.

Analyze Multi-Tier Performance

Although factors such as disk I/O or CPU utilization are important in single-tier applications, they become even more important in multi-tier applications that have multiple computers—and thus multiple disk subsystems and CPUs—contributing to the application's performance. In addition, the network—the links that allow the different application tiers to communicate—becomes a major factor. In my experience, the network is often the first bottleneck you'll run across, especially if some of your underlying code (such as database drivers or other low-level communications components) isn't very efficient. Although there's nothing a developer can usually do about the network directly, you *can* tune your use of it to be more efficient: Transfer less extraneous data, transfer blocks of data rather than continuously opening and closing connections, and so forth.

Measuring thread wait times also becomes more important. Long wait times on a server tier, for example, can hold up multiple end user client-tier applications, so what might seem like only a minor performance issue in and of itself can contribute to major performance holdups across the entire application. Think database locking, which is analogous, conceptually, to thread wait times: Most developers are familiar with, and have experienced, the performance damage that poor lock management can do. Poor thread management can be even worse.

The trick here is the phrase, "what might seem like only a minor performance issue in and of itself." In other words, it's tough to spot performance problems by just looking at a single tier of the multi-tier application. Is a long thread wait time on the server a problem? That depends—is it holding up client application requests? Tools that can analyze the entire multi-tier application *at runtime* are about the only way to discover this. Factors such as thread wait states can come and go faster than a human can see them, and correlating them to delays on one or more remote machines is simply beyond human capacity to accurately measure. That's why performance analysis toolsets exist in the first place.

Mapping Performance to Project Requirements

How much performance does your application need? The answer needs to be spelled out in the applications' requirements, as already discussed earlier in this guide and at length in *Definitive Guide to Delivering Quality Applications* (Realtime Publishers). Without requirements for performance, you're flying blind: You can do the best job you can at improving performance, but you'll never know whether you've improved it enough for it to be considered high quality, and you'll never be able to make good judgments about the time required to achieve a certain level of performance.

Integrating Performance into Testing

Application testing should always involve performance testing. This of course ties back to the application requirements. The requirements must specify a required level of performance, and application testing must verify that the requirement has been met.

Good commercial software test suites typically include performance measurements as a part of their feature set. Unlike code profiling tools, which may measure technical specifics such as memory consumption and module execution time, testing applications may instead measure more end-user-visible performance metrics such as the time required to complete specific transactions or business processes.

Assessing Performance Impact: Before and After Comparisons

When you hire a good personal trainer at the gym, one of the first things they'll do is take measurements: weight, size, body fat, and so forth. Periodically, they'll re-take those measurements as a means of charting your progress at the gym. It's a sensible approach: If you're not making progress, you'll be able to tell and perhaps alter your tactics or look for problems (like failing to follow the suggested diet!).

Application performance is no different. It's truly important to carefully measure performance both before and after any major changes you make so that you can measure your progress at improving performance. If your changes aren't making the improvements you'd hoped for, you may need to look elsewhere for performance problems, or you might have hit the practical limit of what you can improve given what your application does and how it works. Knowing when you've hit that limit is good information because it helps keep you from spending any more of your valuable time chasing after increasingly smaller improvements.

Good tools can help you do all this by saving performance results and profiles, and comparing them to one another. This helps highlight improvements—as well as inadvertent dis-improvements—and helps you build management reports and make smarter decisions about how to proceed.

Common Sources of Security Problems

Security is one of the things that people tend to think about *after* the application is written. Witness Microsoft's security troubles in the late 1990s and early part of the 21st century, when it seemed that every Microsoft product was awash with security problems. Microsoft addressed the problem by essentially starting to think about security first and throughout the development process—a concept they called their "Trustworthy Computing Initiative." There are a few broad categories of security issues that the Initiative is designed to stop, and they can be useful in helping you develop more secure applications, too.

Note

I completely acknowledge that security can be deadly boring, and that it often seems to add very little value to an application. That's why it's often referred to as a "non-functional" requirement, although you'll find that *having* good security doesn't seem to add value, *not having* security is definitely perceived pretty poorly. In other words, it's worth the boring, additional effort to include solid security in your application because although doing so might be under-appreciated, *not* doing so will be very loudly unappreciated.

Lack of Clearly-Stated Security Design Goals

One reason that security is often perceived as boring and of no additional value is that project requirements seldom state what their security requirements are. That's a no-no: Project requirements *must* state the conditions under which the software will be used, how it must be secured, what auditing must be provided, and so forth. Without any security requirements clearly stated up front, nobody should be surprised later if the application turns out to have no security whatsoever.

Inappropriate, Ineffective Security Models

Developers who write their own security models are usually asking for a world of trouble. I like to say that security makes an application exponentially more complicated: If you have an already-complex application, having to worry about an internal security model squared or cubes that complexity. Instead, whenever possible, try to rely on external security models from proven sources.

For example, Microsoft SQL Server provides an excellent security system, mapping logins to AD, providing granular permissions on database objects, and so forth. It seems like nobody uses it, though: I'm constantly running across applications that simply login to SQL Server using a single, generic login account or an application role, then implement their own security to control what users can access. There's a perception that managing security in SQL Server itself will be difficult, when in fact it's several orders of magnitude easier than re-inventing the wheel and building a good security system into the application.

The problem is that many applications that take the "I'll do it myself" approach don't actually do it themselves. They tend to rely on UI as a security mechanism: If users don't have a UI to delete customer records, for example, they won't be able to delete customer records at all. That's the type of thinking that leads to trouble, though, because all a user has to do is simply bypass the application's UI and talk to SQL Server directly, which isn't that difficult—especially if SQL Server has been left wide open. Developers too frequently rely on applications to provide security and avoid the hassle of truly protecting the data. Something as simple as Microsoft Excel can execute SQL queries, letting a mischievous user do all kinds of damage.

Thus, unless you're an expert at designing security systems, try to adopt simple security models that leverage the professionally-designed security systems you already have at your disposal. And if you do need to design your own security system, make sure it's *deep* (meaning it provides security at every level of the application, not just the UI), and *broad* (meaning it covers every possible circumstance under which the application *may* be used).

Failure to Consider and Protect Against "Unexpected Usage"

One reason developers—and I include myself in this statement—are so bad at developing security systems is that we don't have vivid enough imaginations. This is the same problem that winds up causing most security-related bugs. For example, when Microsoft issues a patch because a malformed JPG picture, when viewed in Internet Explorer, can suddenly cause malicious code to be executed—well, that's a lack of imagination on the part of the original developers. Call it a failure to implement boundary checks or whatever you like, but it's essentially nobody thinking that the software would ever deal with this unexpected, deliberately-malformed file.

Any time you make an assumption, also assume that someone else will neglect your assumption and that another someone will deliberately try to thwart it. A customer name can be 100 characters in the database? Okay, so you create a 100-character text field and think you're okay, right? Wrong: Assume someone will find a way to access your code directly, without the GUI, and stuff a 200-character name down the application's throat to see what happens. Assume they will bypass your entire application and try to insert the data directly into the database. Yes, it's a little paranoid. That's the world of security: Considering protection against unexpected usage.

> **Note**
>
> Nearly *every major security issue* that's been well-publicized in the past decade has been a lack of consideration for unexpected usage. The "Conficker" worm of 2009 exploited a software flaw. The "I love you" virus of the 1990s exploited a software flaw. The IIS-targeting "Code Red" virus exploited—you guessed it—a software flaw. And these flaws were universally a matter of developers not considering a certain type of unexpected condition. This is one reason why I hate developing my own application-specific authentication or security layers: It's just a bunch more code where my imagination won't be as vivid as it needs to be.

Lack of Security in Coding Standards

We've discussed coding standards in previous chapters. Although coding standards are often implemented to improve maintainability and debugging, they can also be used, to a degree, to help improve security. For example, the Computer Emergency Response Team (CERT) at the Software Engineering Institute of Carnegie Mellon University has a list of 10 security-relating coding standards:

- Validate input
- Pay attention to compiler warnings
- Design for security policies
- Simplify your code
- Deny privileges by default
- Use the "principle of least privilege"
- Sanitize data before sending it to other systems
- Practice defense in depth
- Use effective QA techniques
- Adopt a secure coding standard

Resource

Read the full list, including descriptions and examples, at https://www.securecoding.cert.org/confluence/display/seccode/Top+10+Secure+Coding+Practices;jsessionid=5648C04E538EBE721DC23493B9927631.

The point is that there are certain basic, everyday practices that can make applications considerably less vulnerable, which require developers to have less vivid and paranoid imaginations, and can be incorporated into coding standards that everyone on the team follows consistently, checks through peer review, and so forth.

Detecting Security Problems

Although performance problems are typically noticed pretty quickly by application users, security problems may go unnoticed until they're exploited—which is why detecting them during development and testing is so important. Finding security problems is usually tricky because security problems aren't what I call "active bugs." In other words, an application can run perfectly smoothly, without error, and still be laden with security problems.

Finding security problems really consists of two parts: Relying on best practices to avoid them in the first place (which we've already touched on) and relying on industry knowledge to look for known potential problems. The latter is basically a matter of memorizing all the different security things that commonly crop up in an application, then diving in and looking for them—a task eminently suited to automated tools, which can remember a far larger set of potential conditions, and find them faster. Figure 5.4 shows how one commercial tool works with security scanning. The product comes with a rich set of security-related rules, along with detailed explanations about what each one involves. It can scan for code that meets these rules, thus spotting potential security problems. Commercial providers of security scanners include Micro Focus, Fortify, and Veracode, and Microsoft offers a tool named PREFast (which scans C and C++ code) as part of its Windows Driver Development Kit (DDK).

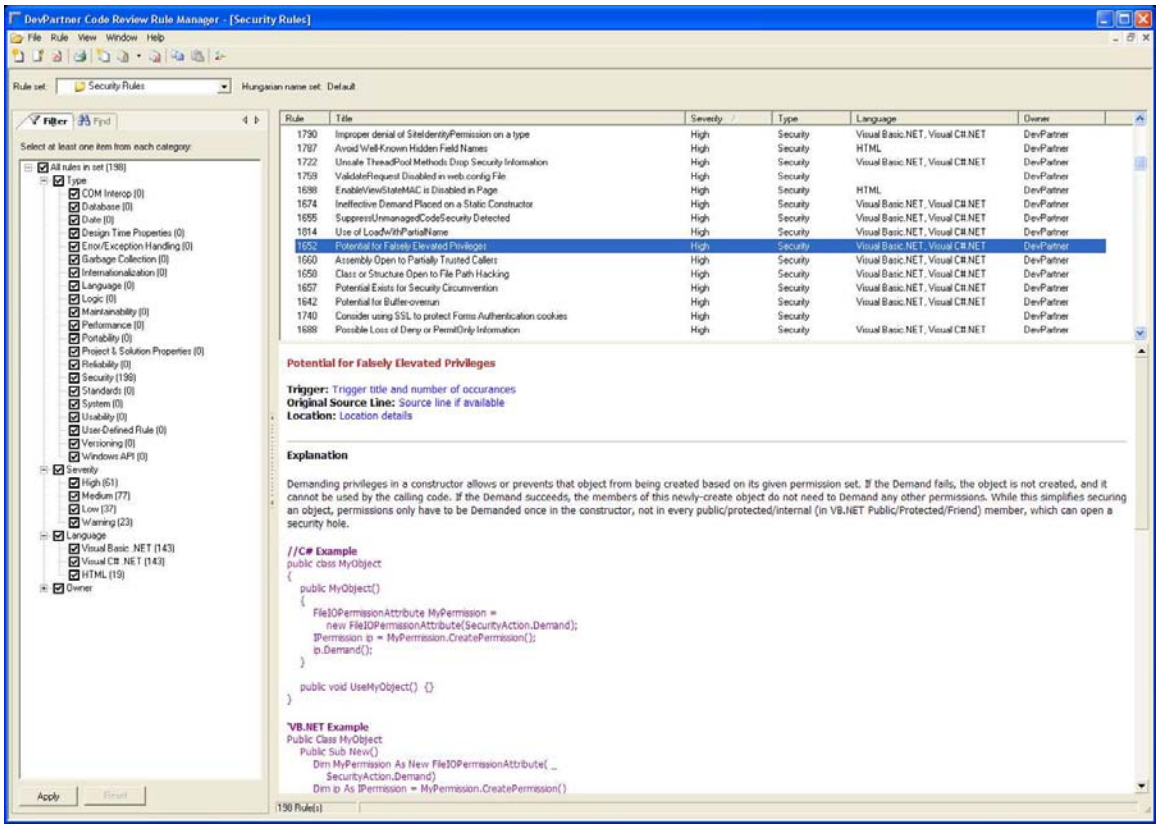


Figure 5.4: An automated security analysis example.

Integrating Security into Testing

Perhaps it should go without saying, but let's say it anyway: Security stuff needs to be tested. This, again, can be tricky. You can take the approach of performing "white hat" testing, where you actively try to break the application's security. The problem with this approach is that you can spend tons of time with little to show for it, and may not improve the application's security at all. The best you can usually do in this regard is to test for the specific security goals called out in the project's requirements to ensure those goals have been met.

A second approach is to use automated security scanners as a part of the testing process. Many tools can integrate security scanning into a continuous build process so that security scanning is always being performed. This doesn't by any means ensure an absolutely secure application, but it does help provide a *more* secure application by calling your attention to common coding practices that leave themselves open to exploitation.

Assessing Security Impact: Before and After Comparisons

Just as I discussed for performance, it's important to measure your security progress, both to confirm that you are indeed making progress and to help weigh that progress against the associated time and costs. There are two general ways to do so, and you can use them both because they're actually complementary.

One way is to give your application a "security score," awarding a point (for example) for each security goal that has been met. This does, of course, require that you actually have security goals, which is something I'll discuss next. As your application's security matures and you complete more of your security-related goals, the application earns more points and you have a simple metric for gauging your progress. By measuring the time it takes to complete each goal, you'll get an idea of resource expenditures, too.

The second way is kind of the opposite: Log all security problems as defects, then start working them off. The fewer defects you have, the better your security situation is. Defects should typically point back to a specific project requirement. Most defect-tracking systems allow you to also track developers' time, so you can start to get a feel for how much effort has been spent, and how much more effort will likely be needed to work off all the security problems.

Automated scanning tools like the ones I've already discussed also provide a means of assessing your security before-and-after. By periodically re-running tools, you'll be able to chart your progress based on their reports.

Mapping Security to Project Requirements

If your project requirements don't specify any security goals, your application will, quite simply, not be very secure. Developers, designers, and testers should not be expected to create a more secure application on their own initiative: Security, as with everything else in the application, needs to be driven by project requirements. Having the right tools in place to track project requirements, relate specific sections of code back to those requirements, and build test cases that test for those requirements, is the best way to ensure that requirements are being met.

Coming Up Next

In the next chapter, I'll look at code testing with a specific eye toward testing for errors, coding inefficiencies, and performance. We'll look at how to construct test cases and at various types of low-level testing, along with test monitoring and reporting ideas. We'll examine some specific types of testing tools, including code coverage analysis, runtime error detection, memory tracking, performance analysis, and so forth. All of this will be firmly from the developer's point of view—think *unit testing* rather than the more comprehensive and sweeping tests that might be performed by a quality assurance tester. In other words, these are tests you, a developer, can do on your own to help improve the quality of your application.

After that, our final chapter will wrap up everything with a look at the benefits of automated debugging, analysis, and testing. With the rest of this guide behind you to define the things you need to do, automation can help you do them more efficiently, accurately, and consistently.

Download Additional eBooks from Realtime Nexus!

Realtime Nexus—The Digital Library provides world-class expert resources that IT professionals depend on to learn about the newest technologies. If you found this eBook to be informative, we encourage you to download more of our industry-leading technology eBooks and video guides at Realtime Nexus. Please visit http://nexus.realtimepublishers.com.