13 Tips to Write Secure Applications and Boost IBM i Security

By Robin Tatam

Security. It surrounds every aspect of life in the new millennium, and it’s never going away. Human nature is such that any time we’re told there’s something we shouldn’t do or somewhere we shouldn’t go, someone is always going to do exactly that and go precisely there. This is especially true when there’s something to gain by violating the rules.

The proof? Barely a day passes without new headlines reporting another cyber attack, policy violation, or data breach. Secretly, we breathe a sigh of relief that it happened to someone else, but most of us know that we’ll all eventually feel the impact in some capacity. Maybe it’ll manifest itself through increased bank fees, higher retail prices, or more hoops to jump through when we log in to our corporate firewall. And eventually it could—nay, it will—be us that it happens to.

So who’s responsible for the security of an IBM i server and DB2 database? IBM builds the hardware and develops the operating system, application vendors and internal programmers write the business applications, and users maintain the corporate data. Who shoulders the responsibility for its integrity and its protection? After more than a decade of working as a security specialist, my answer is a resounding “EVERYONE!”

IBM has done their part. Power Systems servers running IBM i™ have garnered an envious reputation for being some of the most securable servers on the planet, with numerous world-class integrity features. Notice I said securable instead of secure. The annual IBM i Security Study reports alarming statistics showing that most of these servers are operating with default—and largely open—security configurations. You would never assume a server running Windows is secure, and you should never assume it with one running IBM i.

System users, such as programmers and administrators, should be managed by a combination of profile settings and good auditing controls. Application users should be managed by the applications. Ultimately, however, users will do whatever applications allow, not necessarily only what they’re designed to do. And it’s programmers who build those applications, so we have to talk about how to build a more secure application.

The goal of this article is to identify important development considerations that will empower programmers to enhance the security of our data rather than undermine it.
1. Docu-what?
Personal experience has taught me that documentation is a dirty word to many in the programming community. It hampers our simple desire to code. But documentation is also the map that guides future travelers. Without it, every enhancement and bug fix begins with a time-consuming discovery task and increases the likelihood that the application could be broken by the change. Before a single line of code is written, planning documents should identify operational requirements such as these:

- Application libraries
- Profile ownership
- Authorization lists
- Authority schemes—public and private
- Runtime attributes

Documentation should also be embedded in the application programs. Comments should be clear, concise, and in plain English (or your language of choice). In other words, do not simply repeat what any programmer can read from looking at the code.

2. Segregate Libraries
When building an application, many programmers lump all of the objects into a single library as it seems simpler on the surface. However, programs and files typically have different security requirements, and having them cohabit can make securing them more complicated. Segregating non-data objects and data objects into different libraries permits library-level security to control access.

3. “Softcode” Library Lists
I was taught that hard-coding a library name into a program was a cardinal sin, but we’ve learned that search path manipulation is an easy way to wreak havoc and even circumvent security. An acceptable balance may be found by storing library names in data objects, such as data areas, so that they cannot be altered by users but can be changed without requiring modifications to the application code.

4. Own Your Objects
Every object has an owner, and if you’ve ever had to delete a user’s profile, then you understand why ownership shouldn’t be retained by the programmer. Objects should never be owned by a group profile, especially one that consolidates the application users; each member would indirectly own the objects, giving them excess privileges. Instead, create a profile that has only one purpose in life: to own the application objects. Just remember, if new objects are created during the execution of the application, the program should also establish the correct ownership and authorities. Try to avoid assigning ownership to IBM-supplied profiles because if anyone—including IBM—makes a change to the profile, it might have an unpredictable impact on the application.

5. Design the Database
Take time to classify the data in order to identify how stringent the access control should be. For a simple scheme, consider public, semi-public, and private. The database should be normalized to prevent information redundancy, and I recommend encrypting fields that are classified as private. Strong encryption functions are inherent within IBM i, and third-party providers can assist if code modification isn’t possible. Journaling is a popular technique that originated with application recovery and now has uses for replication and for auditing. Consider collecting before and after images of data changes in critical files and archiving that journal data according to corporate policy and compliance mandates.

6. Read the Menu
Application menus were a solid approach to security back when terminal displays were the only way to access data. However, they fall flat on their face now that other interfaces exist that can facilitate direct access to the database. There’s nothing wrong with using menus to improve the user experience and to restrict user movement, but don’t rely on them to police data access. When a user steps outside the constraints of a menu, other controls such as command line restrictions, object-level security, and exit programs become even more critical.
7. Good Code
While it seems obvious, write good code! My development career exposed me to some of the most horrendous applications I could imagine. Unwieldy programs with unmanageable top-down design, missing—or worse, inaccurate—comments, and redundant functions. Programs that are modular, concisely coded, and well-documented are easier to maintain and easier to review for unauthorized code modifications. ILE constructs make this easy. You should be taking advantage of them.

8. Public and Private Authority
IBM i contains a unique concept known as *PUBLIC. This is the default level of access granted to a user who hasn’t been assigned a more specific access level. Many open configurations on servers running IBM i can be traced to the fact that IBM ships the default *PUBLIC authority level as *CHANGE, which is sufficient to read, change, and delete data in a file.

Public authority is determined when the object is created from a parameter on the associated create command. By default, the command defers to a library attribute and that, in turn, defers to the QCRTAUT system value. I recommend setting the appropriate explicit value on the library as it allows different defaults to be used for each application library. The popular “deny by default” data access model calls for the public to be excluded, and the application users—or better yet, user groups—with a demonstrable need are granted private authority.

9. Authorization Lists
Granting authority to hundreds or even thousands of individual objects can be an overwhelming administrative chore. Authorization lists allow many objects to be addressed as a single entity and are a useful mechanism. A significant benefit of using lists is that authorities can be maintained while the associated objects are in use. This can be very beneficial in a 24/7 shop where obtaining object locks can be a monumental task. Just remember that *PUBLIC authority will come from the individual object unless you specify otherwise.

10. Adopted Authority and Profile Swapping
Two techniques are available to allow a program to run with different authority than the user who invoked it. Instead of granting authority to individuals or even groups, authority can be elevated temporarily while the program is running, alleviating the requirement to grant authority to the users themselves. In the case of adoption, a runtime attribute on the program object controls the inheritance. Both special and private authority is culled from the owner of the program, which is beneficial if following the recommendation of using a unique profile for ownership.

Profile swapping allows a user to assume the identity of another and is accomplished via calls to several easy-to-use IBM-supplied APIs. Both approaches have pros and cons, and more detailed information can be found using your search engine of choice. Overall, these are two of the most useful functions in a security-conscious programmer’s toolbox.

11. Cover Your Exits
IBM i contains a registry of exit points, which are almost like subroutines in the OS’s functions. When a function is invoked, the OS can pass control temporarily to the assigned exit program—if one exists—to perform any ancillary task before the OS resumes control and processes the request.

There are dozens of exit points in several different categories, but about 30 of them pertain to network access and should be considered critical. While object-level security remains effective during these requests, the OS can only enforce one setting across all interfaces, so a user who can change data on a green screen can also change data through an FTP or ODBC connection. While these exit programs can be written in-house, regulatory compliance typically frowns upon self-policing, so I strongly recommend evaluating a commercial solution to selectively process and audit user requests. Preventing data leaks is a requirement shared between exit programs and object-level security.
12. Plan for High Availability

System availability is often a requirement of regulatory compliance. Keeping systems running in the face of a technological disaster is a popular business initiative and one that continues to grow in popularity. When designing a new application, consider the expectation that the application may need to be replicated. A side-benefit of journaling for HA purposes is the audit and recovery trail that can be built into the application.

13. Query and Other Tools

When designing a security infrastructure, consider how the application data will be accessed. Programs can use adoption to temporarily access secured files, but other tools may not have that luxury. Placing the invocation command for a native tool (e.g., WRKQRY) inside a CL “wrapper” can allow it to take advantage of the same technique. Consider authorizing some generic profiles to the authorization list to allow users to access data in a limited capacity. For example, grant *USE access to a profile named READONLY and then swap to that profile if a user needs to use ODBC or Query to view the data. If data is encrypted, native reports may have to be authored if plain text viewing or reporting is required.

Summary

In my capacity as an auditor, I discover vulnerabilities on hundreds of servers every year. In the vast majority of cases, these servers are poorly configured and are running applications that rely heavily on legacy security mechanisms or have no security considerations at all. Ironically, many of those same organizations blame IBM for allowing users to extract data, instead of taking responsibility and investing in the design of a secure application.

Programmers are the linchpin when establishing secure applications. Regardless of whether they’re developing internal solutions or commercial products, acknowledge the risk posed by not having their cooperation, train them in security functions, and work to obtain their buy-in regarding the fact that securing an app is one of the most critical aspects of securing data and maintaining compliance.