QIBM_QDB_OPEN: The Open Database File Exit program

How to use the Open Database File Exit program to secure your database

Gene Cobb

February 26, 2014

This article introduces you to QIBM_QDB_OPEN, the database open exit point that was made available with IBM i Version 5 Release 3. This exit point helps IBM i programmers and administrators to manage database security and enhance IBM i security policies. In this article, readers can learn the ways to use the IBM® DB2® or IBM i OS exit-point configuration options and various considerations and approaches when designing the exit-point program.

Introduction

Terms such as hacking, phishing, Sarbanes Oxley and HIPAA are becoming more common in the lexicon of today's IT professional. Despite their familiarity, they can cause you to shudder slightly if you do not feel completely confident about your current IBM i security implementation. Security is a hot topic these days because one of the most important assets within your business is the information stored in your databases. Therefore, establishing, implementing, and maintaining a sound database security policy must be at or near the top of your IT shop's priority list. Among the list of questions you need to ask yourself regularly are these: "Are you doing everything you can to keep your data safe and secure? Are there any gaps in your security implementation? If so, are there any ways to address those gaps?" The IBM i platform offers many security features, including ways to keep your data secure. This article introduces you to the Open Database File exit point and how you can use it to address potential vulnerabilities in your IBM i database security implementation.

Note: This article was previously published as an IBM white paper in 2007. It is being republished as an IBM developerWorks® article in the IBM i zone to broaden its exposure and increase awareness of a topic and technique that many DB2 for i customers have found to be useful. Even though the information is several years old, the details remain relevant and accurate.

Exit points

If you are an IBM i programmer or administrator, you might be aware of or, perhaps, have even implemented exit points to help manage and secure your IBM i application environment and data.

An exit point is defined as a specific point in a system function or program where control can be passed to one or more specified exit programs. With the use of exit points, you can write programs to perform custom processing each time a particular event occurs on the system.

QIBM_QDB_Open

In IBM i V5R3, support for QIBM_QDB_OPEN, the Open Database File exit point, was added. This exit point gives you the ability to call a program whenever a database table on the system is opened. After the exit point has been properly set up, the exit program is called when any job on the system issues a request to open a physical file, logical file, SQL table, or SQL view. This action occurs regardless of the origin of the open request; this includes programming interfaces [such as record-level access and SQL), as well as non-programming interfaces (such as IBM i Navigator, IBM Query/400, IBM DB2 Web Query for i, IBM Data File Utility (DFU) and even the IBM i commands Open Query File (OPNQRYF) and Display Physical File Member (DSPPFM)]. When the database open and resulting invocation occurs, the exit program runs in the same job that issued the open request and can actually set a return code value to reject the open request.

Understanding database opens

The exit point invokes the exit program when a full open is requested. A full open results in the creation of an open data path (ODP). The ODP provides a direct path from the program to the table so that the program can issue input and output operations to access the data in the table. If you use the exit point, it is important that you understand when a full open occurs. The timing of this event is dependent on several conditions, including the kind of database access you request: RLA or SQL.

Record-level access

Record level access (also referred to as *native I/O*) is the traditional way that high-level language (HLL) programs (such as RPG and COBOL) access the contents of files. Many programs today still use record level access to access the data. The full open in record level access occurs when the program issues the open operation or method, not during the actual attempt to access the data (such as a READ or CHAIN operation).

When an RPG program uses record level access, tables are defined in the program's file specification section (F-spec). The point at which the open actually occurs depends on whether the USROPN keyword is specified in the F-spec. If it is not, an implicit full open occurs at the time the program is first invoked. However, if the USROPN keyword is specified, the program must explicitly open the table using the OPEN operation or the %OPEN built-in function. In either case (implicit or explicit open), the value that the RPG program assigns to the last record (LR) indicator dictates the table-open behavior of subsequent calls of that program. If LR is turned on, the tables are closed before the program ends and a full open occurs in each subsequent call to the program. But, if the program does not turn on LR, all tables are left open for subsequent calls to that program, freeing them from the burden of full opens. In this condition, the ODP is maintained and the program is in what is referred to as re-entrant mode.

Another type of open with record level access is "shared open," a technique used to share a file's ODP among multiple programs in the same job or routing step. One program issues the full open for the file and all other programs in that job can use the same ODP for that file, thus saving system resources such as processor, memory, and disk. A shared open is controlled either by specifying the SHARE(*YES) parameter in the override or by creating the file commands before opening the file. Shared opens are commonly used in applications that invoke the OPNQRYF command.

SQL access

HLL programs and remote interfaces [such as Open Database Connectivity (ODBC), Java Database Connectivity (JDBC), and ADO.NET] can use SQL to access tables. Obviously, an open occurs when the SQL OPEN statement is used on a cursor. Not-so-obvious opens take place when the SELECT INTO, INSERT, DELETE, and UPDATE statements are started. As with all SQL statements, these opens are carried out by the database manager, which internally opens the tables for the application. During the initial execution of the open cursor or other SQL statement, the database manager performs a full open. The table is closed when the cursor is closed or the statement has finished running. If that same statement is run a second time within the same job or connection, the process is repeated and another full open occurs. After this second execution, the database manager does not actually close the table; it is left open for subsequent executions of the statement within the same job or connection. During the third execution of the statement, the database manager does not have to perform the full Open because the table is already open. This is known as a "pseudo open" and, at this point, the statement is in a state commonly referred to as "ODP reuse mode". Pseudo opens are similar in characteristics and behavior compared to shared opens in native I/O operations.

Invoking the exit-point program

Comprehension of the types of opens is important because the exit-point program is only invoked when a full open is requested. The exit-point program is not invoked for pseudo opens, shared opens or when tables are accessed by programs in the re-entrant mode.

In the case of an SQL view, logical file or Query/400 query that is defined as a multi-table join, the exit program is only invoked one time, but its input parameter list contains the view or logical file that is being opened, as well as each of the underlying tables that the view or logical file is comprised. It is up to the exit program to extract and process each of these entries individually.

In addition, many tables (supplied by IBM) and temporary tables are excluded from the exit point's control. Therefore, even if a full open is requested, the exit program is not be invoked if the table being opened is in QTEMP, QSYS, QSYS2 or several other system libraries. For the complete list of these libraries, refer to the QIBM_QDB_OPEN exit point entry in the IBM i Information Center.

Use cases

As is the case with several of the exit points on the IBM i environment, the primary implementation of QIBM_QDB_OPEN is to supplement security, specifically, database security. It can be used to enhance an existing security implementation in the following ways:

- Assisting in the enforcement of a security policy
- Performing some level of auditing in an attempt to establish where security exposures exist

Enforcing a database security policy

In the past, securing your data was fairly straightforward. User-profile settings, such as the initial program and job description, might limit what 5250 applications users were confined to. For example, the limit-capability parameter of the user profile can be set to *YES to prevent users from obtaining a command line. The application can be relied on to marshal access to specific tables for specific users. This application-level security was sufficient to secure the data. Object-level security was not required because there was no other interface to the data. In today's environment, application-level security is not sufficient. You must account for and stem the threat of unauthorized access to remote database objects from users or processes that attempt to access sensitive tables outside of a main application. External applications and interfaces, such as IBM i Navigator and other ODBC- or JDBC-based tools, allow users to gain access to the system that is not under the control or influence of a mainline application and, therefore, cannot be controlled by the security built into that application.

If application-level security describes your environment, it is strongly recommended that you at least investigate the option of implementing object-level security. In the long run, this is one of the best and most efficient methods of securing your data. For more information about object-level security, refer to the IBM i Information Center and the IBM i Security Reference manual.

Supplementing application-level security

If you have determined that implementing object-level security is too large of an undertaking, or the return on investment (ROI) is just not there, consider using the exit point to help you solve some of your gaps in security. For your 5250 applications, you can still rely on application-level security and the settings in the user profile to block unauthorized access coming from the traditional 5250-based application interfaces. In addition, you can use the exit point to block unrestricted access with the aforementioned remote interfaces.

Providing security granularity

Solution providers can use this exit point (and other system exit points) to allow their customers to customize, control, and manage both local and remote database object-access policies. For example, the exit program might use a security table that contains a list of sensitive database tables and authorized users, as well as those users' local and remote access rights (see Table 1).

Table 1: Example of an exit-program security table

Database table	User profile	Local access	Remote access

ORDERS	COBBG	N	Y
ORDERS	KMILL	Y	N
ORDERS	JAREK	Y	Y

When invoked by a database-open request for table ORDERS, the exit program can determine the type of access (local or remote) being requested, look in the security table to verify that the requesting user has that type of access to the table and, accordingly, allow or reject the database-open request.

Using the example data in Table 1, the user-written exit program performs the following tasks:

- Prevents user COBBG from opening the ORDERS table from the main 5250-based application interface; but does allow that user to access ORDERS through an ODBC interface to populate a spreadsheet for a report.
- Allows user KMILL to open ORDERS from the main 5250-based application, but can prevent that user from using a remote interface to access the data.
- Allows user JAREK to open ORDERS from both the local and remote interfaces.

You can even take it a step further and only allow remote access during certain days of the week or hours of the day. Because you write the exit-point program, you can control access to specific users who can access the tables and under specific conditions. As you can see, the exit point provides a degree of security granularity that is not possible when using object-level security.

Auditing

From an auditing perspective, the exit point can help you capture database security-related information, such as:

- Users who attempt to access sensitive tables
- Whether or not an open was the result of an SQL query
- The type of open operation (read, insert, update, and delete)
- The programs and interfaces used when attempting to open tables

Comparing exit point with object auditing

If you are familiar with the security features on IBM i, you might be aware of the audit journal that enables you to instruct the system to capture and log the occurrence of any security-related event. These events are recorded in special system objects called journal receivers. One of the subset features of the audit journal is the ability to log accesses to an object, such as a sensitive table. This is called *object auditing*.

As an example, to set up object auditing for the AGENTS table in the FLGHT400C schema, issue the following commands from an IBM i OS command line:

- CHGSYSVAL SYSVAL(QAUDCTL) VALUE(*OBJAUD)
- CHGOBJAUD OBJ(FLGHT400C/AGENTS) OBJTYPE(*FILE) OBJAUD(*ALL)

Any attempt to access that table is logged to the audit journal. The audit information can later be displayed or extracted by using the IBM i OS Display Journal (DSPJRN) or Display Audit Journal Entries (DSPAUDJRNE) commands.

From an auditing perspective, the capabilities of object auditing and the exit point appear to be similar, but some key differences exist between the two.

- **Administration:** With object auditing, after you have set it up, the system takes over and is responsible for capturing and logging each access of the object. With the exit point, the IBM i programmer is responsible for writing the program to extract and log the details of the access attempt, as well as for creating the log table and other supporting objects.
- **Object-level authority:** The exit-point program is not invoked if object-level authority prevents the requesting user from accessing the table. Because object access is restricted, the table is not opened, and the exit-point program is not called. A somewhat negative ramification of this is that the exit program cannot log occurrences of such unauthorized access attempts. However, object auditing can detect such access requests, even though they are not successful.
- **Ability to take action:** Object auditing can only log the occurrence of the object access. It cannot block the access attempt or immediately trigger work flow to alert someone of the access attempt. As long as object-level authority does not prevent the exit-point program from being invoked, the exit point gives programmers the ability to block access or carry out necessary workflow tasks.

For more information about IBM i audit journal and object auditing, refer to the IBM i Information Center and the IBM i Security reference manual.

Comparing exit point with database triggers

You might be more familiar with the concept of database triggers, which are user-written programs or SQL routines that are associated with a database table. Database triggers are automatically activated (triggered) by the database manager when a row is read or a change occurs in the table, regardless of the interface that initiated the change. The primary purpose of database triggers is to monitor database changes and perform tasks related to those changes. When evaluating whether to use the exit point or triggers for your security requirements, here are some things to consider:

- **Administration:** The exit point is a global (that is, system-wide) setting. You only have to set it up in one place and write the exit program. Depending on the number of tables you want to trigger-enable, database triggers can require more intervention. This is because you must add triggers for each individual table that you want to audit. In addition, to ensure that each type of database access is detected, you must add four triggers to each table: one each for READ, UPDATE, INSERT, and DELETE (although the same program can be called for each one). To perform auditing tasks, both methods require you to write programs to create supporting objects.
- **Object-level authority:** As with the exit point, database triggers are not called if object-level authority prevents the user from accessing the table.

- **Performance:** Because triggers are typically called for each row that is being processed, they can be more expensive in terms of performance. This is especially evident if the query processes many rows. In addition, READ triggers can be expensive and must be used with caution. In some cases with READ triggers, such as when the query contains grouping fields, a temporary copy of the table must be created before processing the statement. As mentioned, the exit-point program is only invoked when a full open is requested. When compared to triggers, this usually results in less program invocations and processing.
- **Information provided:** Triggers have the advantage of having access to the actual data in the event that is audited. In the case of an update, both the before and after image of the row is provided to the trigger program. However, the exit-point program is invoked during a full open and does not provide any information about the rows that are accessed.
- **Ability to take action:** As with the exit point, database triggers can block attempts to access the database or perform other workflow tasks if object-level authority does not prevent the user from accessing the table. To block access, the trigger program simply signals an error to prevent the requested operation from completing successfully.

When deciding which method to use, much of it depends on your auditing requirements. Some auditing regulations (such as HIPAA) stipulate that applications must log every access attempt against sensitive databases and the contents of the rows that are accessed. If this is the case, you really have no choice but to use triggers to fulfill your auditing requirements.

For more information on the database triggers, refer to the IBM Redbooks: *Stored Procedures, Triggers and User Defined Functions on DB2 Universal Database for iSeries.*

The following table provides a summarized comparison of each of the auditing methods.

Table 2: Comparison of the auditing methods

Method	Administration	Programming required	Disabled by object-level authority	Performance	Row content provided	Ability to take action
Exit point	System-wide setting	Yes	Yes	Only called on full opens	No	Yes
Object-level auditing	System-wide setting	No	No	Logs the event	No	No
Database triggers	Per table setting	Yes	Yes	Called for every row	Yes	Yes

Writing the exit-point program

As mentioned, after the exit point is registered, it becomes a system-wide setting; the exit program is invoked each time a full open occurs on many of the database tables on the system. It is important to understand that the job that issues the database open request waits for the exit program to complete before proceeding with the open request. Depending on how your exit program is written and what it does, this has the potential to have a profound negative impact on application performance. It bears repeating that a careless implementation of the exit point can have a significant impact on application performance.

If you decide to use the exit point, you need to take care of the following considerations during the design phase.

- Make sure that your program can handle the condition of multiple table opens (as in the case of multiple table joins that are defined in a logical file, view or query, and multiple tables or views specified in an SQL statement). For example, if a multitable-join view is opened, the exit program is called just one time, but is passed the view and a list of each of the tables that make up the join view. To handle the open request thoroughly, your program must be able to process each view or table in this list.
- If your exit program itself is opening a table, make sure you check for and handle potential recursive program calls; otherwise, your program finds itself in a recursive infinite loop. In addition, if the exit program opens a table using record level access, you must specify the USROPN keyword in the F-spec section. After you have determined that the table that caused the program to be called is not the table that you are opening in the exit program, you can open it explicitly in your program. Again, failure to do these things results in a recursion loop. The topic of handling recursion-looping issues is discussed in more detail later in this article.
- If the exit program changes the return code to 0, the database open fails. If the program does not change the return code or changes it to **1**, the open request is accepted and the open request continues.
- The exit program must be thread safe.

A simple RPG example

With these considerations in mind, here is a simple example of an exit program that is written in RPG. This program is passed in a data structure that contains header information, such as: current user, number of tables opened, type of open (input, output, update, delete) requested, and an offset value. The offset value contains the starting address of the list structure with information about each of the tables that are opened. The program performs the following tasks:

- It extracts the name of the current user who placed the database open request.
- It uses the offset value in the header structure to calculate the starting address of the table list
- From the table list, it extracts the tables to be opened.
- If the table being opened is FLIGHTS and the current user requesting the open is COBBG, the return code is set to **0** and the program ends. Setting the return code to **0** instructs the database manager to block the open request.
- If none of the tables are FLIGHTS, the program ends without changing the return code. This allows the database open request to proceed.

Example 1: A simple RPG program

```
h dftactgrp(*no) actgrp(*caller)
*
d DBOP0100        ds                  qualified
d  HeaderSize                   10i 0
d  FormatName              8
d  ArrayOffset                  10i 0
d  FileCount                    10i 0
d  ElementLength                10i 0
```

```
d  JobName                 10
d  UserName                10
d  JobNumber               6
d  CurrentUser             10
d  QueryOpen               1
d DBOPFile        ds                  qualified based(DBOPFilePtr)
d  FileName                10
d  Library                 10
d  Member                  10
d                          2
d  FileType                     10i 0
d  UnderlyingPF                 10i 0
d  InputOption             1
d  OutputOption            1
d  UpdateOption            1
d  DeleteOption                  1
d returnCode      s             10i 0
d i               s             10i 0
d userNam         s                   like(dbop0100.UserName)
d fileNam         s                   like(dbopFile.fileName)
d curUser         s                   like(dbop0100.currentUser)

c     *entry        plist
c                   parm      DBOP0100
c                   parm      ReturnCode
/free
 // Extract current user from passed in structure
 curUser = dbop0100.CurrentUser;

 // Process each table being opened by the request
 for i = 1 to DBOP0100.FileCount;
 DBOPFilePtr = %addr(DBOP0100) + DBOP0100.ArrayOffset +
 (i - 1) * DBOP0100.ElementLength;
 FileNam = dbopFile.FileName;

 // Don't allow user COBBG to open table FLIGHTS
   if curUser = 'COBBG'
 and fileNam = 'FLIGHTS';
 returnCode = 0;
     return;
   endif;
 endfor;
 return;
/end-free
```

Recursion considerations

The RPG example (in Example 1) works fine when the one and only intent is to block certain users from a specific table or set of tables. Notice that the exit program itself performs no database operations. The user profiles and tables that must be checked are hardcoded in the program. What if this data needs to be more dynamic? What if the exit program itself needs to read from a table or insert rows into a table (for example, when writing to an audit table)? Changing the program to do this requires opening a database table itself (consequently, the exit point issues a new invocation of the exit program). Because one invocation of the exit program already exists on the program stack, a recursive program call occurs; this is a problem because RPG does not support recursive program calls.

You might think that this issue can be handled by compiling the RPG exit program with the ACTGRP(*NEW) parameter. This instructs the system to activate the program in a new activation group each time it is called. Even though this works in some cases, it is not advisable for performance reasons (creating a new activation group is an expensive system activity and occurs for each database open). In addition, when you specify ACTGRP(*NEW), the exit point fails if it is called as the result of an open-table operation from an external SQL user-defined function (UDF).

To avoid RPG recursion issues, you can try the following options:

- **Hard code the values:** As demonstrated in the previous example, you can hard code the values if the data never or rarely changes and you do not require audit logging. In other words, the exit program completely avoids accessing all database tables. Any modifications require updating the source code and recompiling and promoting objects through your change management system.
- **Use nondatabase objects:** Rather than using database tables to store dynamic data (such as user profiles to block and table names to secure), use an alternative system object to store the data. Objects such as data areas and user spaces are valuable for this purpose. Audit logging can be accomplished by some other mechanism, such as sending messages to a message queue or entries to a data queue.
- **Use a different programming language:** If your exit-point program requires database I/O, it is recommended that you use a language that supports recursion [such as C or control language (CL)]. The entire program can be written in C or you can write a CL program that issues a bound procedure call (CALLPRC) to an RPG procedure to perform most of the work. Although you cannot recursively call an RPG program, you can recursively call an RPG procedure. (For examples of a C program and a CL program that handle the recursion, refer to Appendix.)

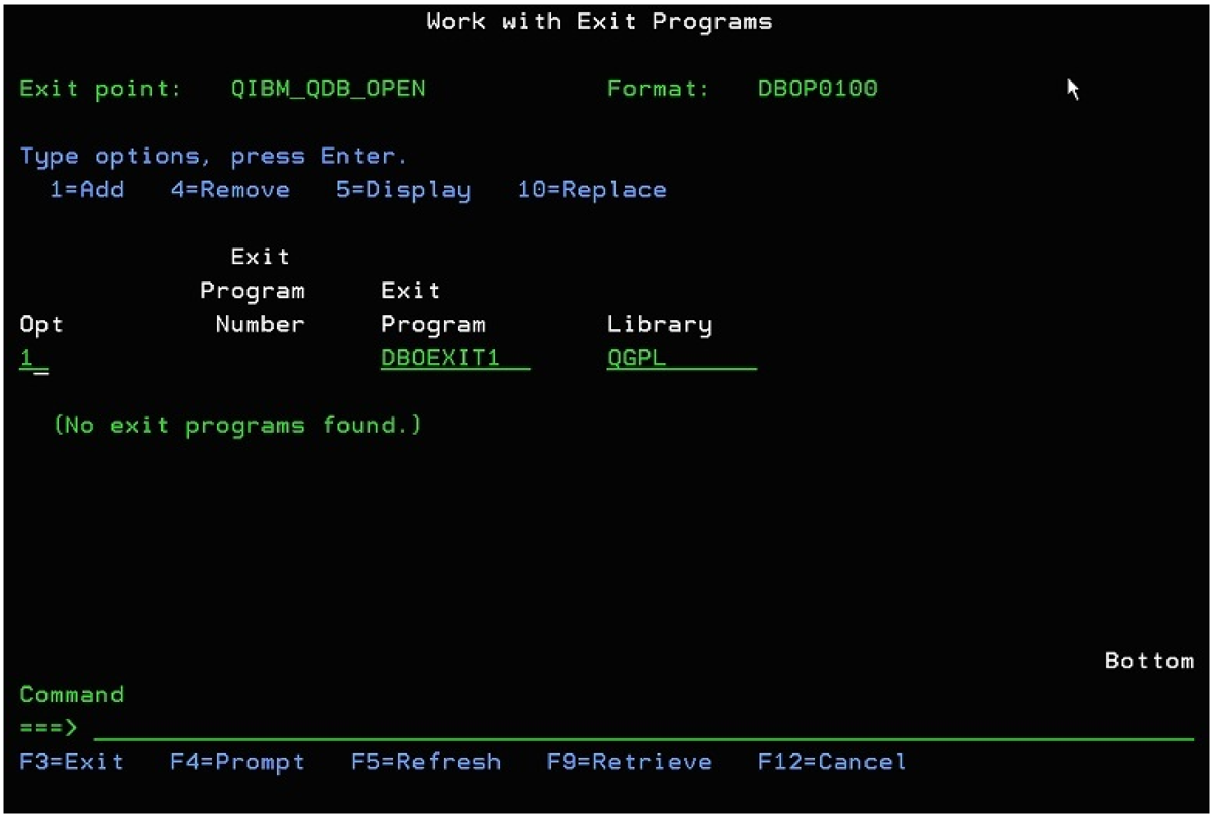
Adding the exit program to the exit point

After the exit program is created, you can instruct the system to call it when a database is opened. To do this, add the exit program to the QIBM_QDB_OPEN exit point by performing the following tasks.

1. Sign on to an IBM i 5250 session.
2. On the command line, type the Work with Registration Information (WRKREGINF) command and press Enter.
3. The Work with Registration Information screen is displayed with the list of system-exit points. Use this list to display information about an exit point and the exit programs associated with an exit point.
4. Type 8 next to the QIBM_QDB_OPEN exit point and press Enter.

On the Work with Exit Programs screen, specify option **1**, exit the EXITPGM program and the QGPL library (as shown in Figure 1), and press Enter.

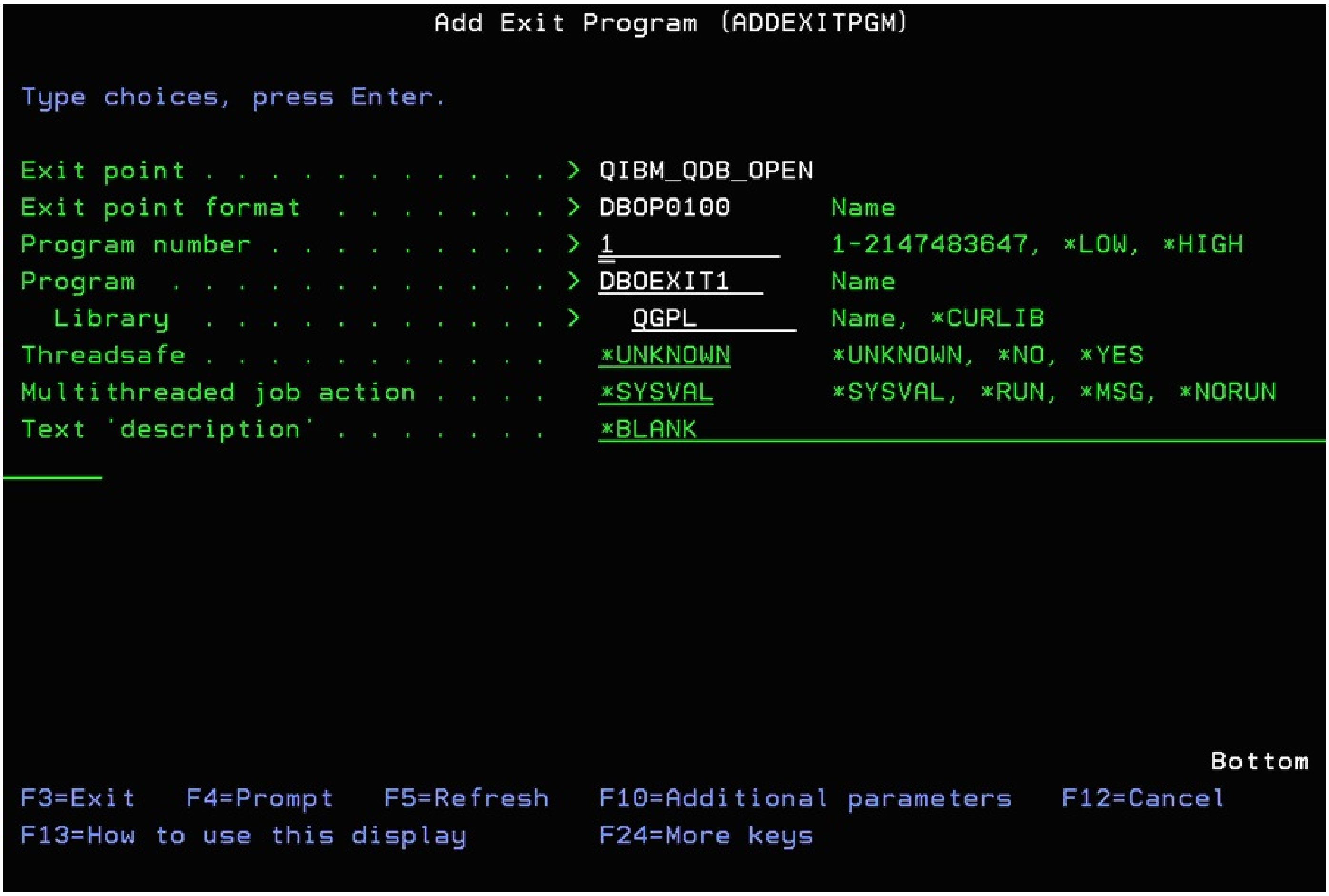
Figure 1: Work with Exit Programs screen with the Add option



```
                           Work with Exit Programs

Exit point:   QIBM_QDB_OPEN           Format:   DBOP0100

Type options, press Enter.
  1=Add   4=Remove   5=Display   10=Replace

              Exit
            Program    Exit
Opt          Number    Program        Library
1_                     DBOEXIT1__     QGPL______

  (No exit programs found.)




                                                                    Bottom
Command
===> ______________________________________________________________________
F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel
```

The Add Exit Program screen is displayed as shown in Figure 2.

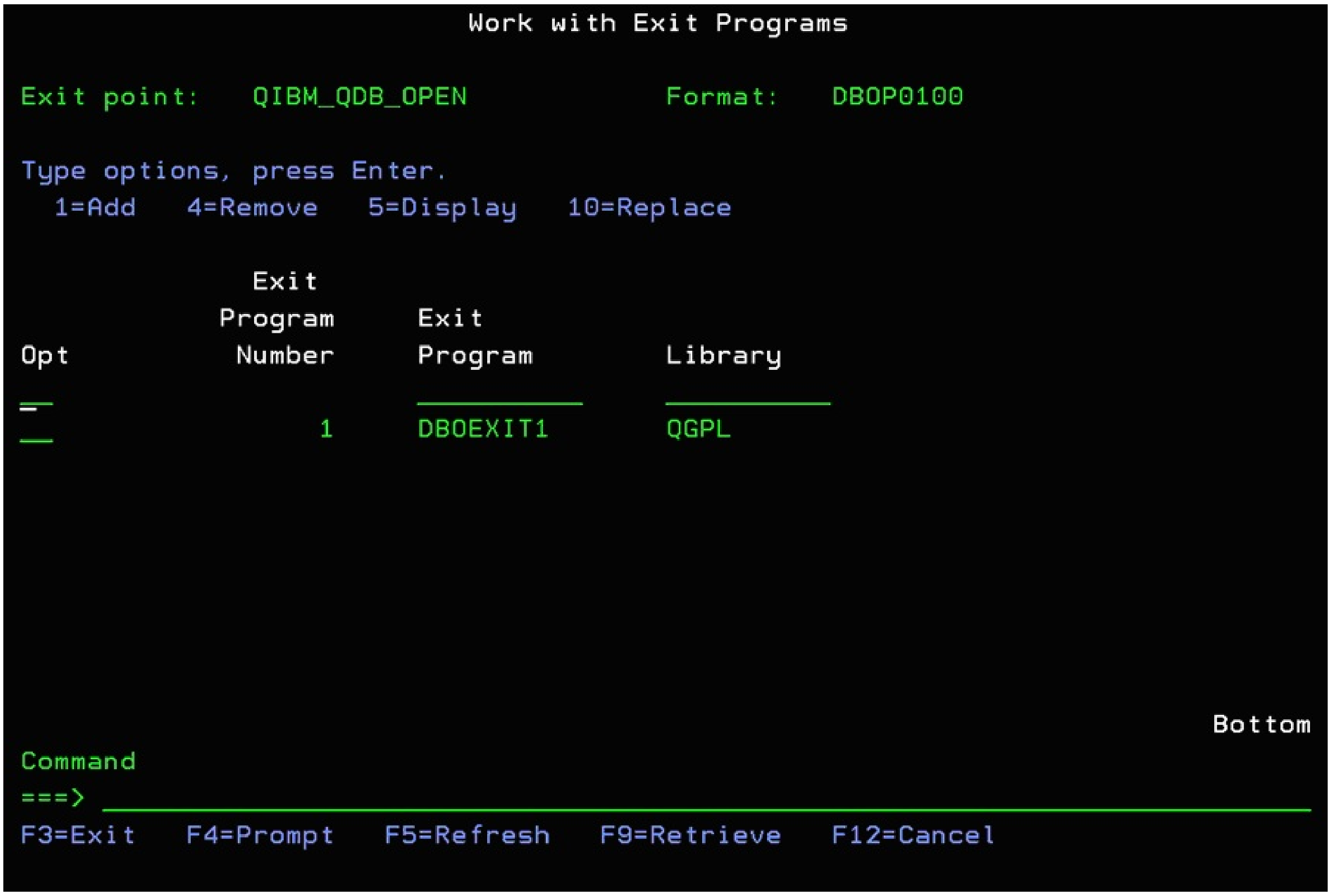
Figure 2: Add Exit Program screen



```
                         Add Exit Program (ADDEXITPGM)

Type choices, press Enter.

Exit point . . . . . . . . . . . > QIBM_QDB_OPEN
Exit point format  . . . . . . . > DBOP0100      Name
Program number . . . . . . . . . > 1__________   1-2147483647, *LOW, *HIGH
Program  . . . . . . . . . . . . > DBOEXIT1__    Name
  Library  . . . . . . . . . . . >   QGPL______  Name, *CURLIB
Threadsafe . . . . . . . . . . .   *UNKNOWN      *UNKNOWN, *NO, *YES
Multithreaded job action . . . .   *SYSVAL       *SYSVAL, *RUN, *MSG, *NORUN
Text 'description' . . . . . . .   *BLANK





                                                                         Bottom
F3=Exit   F4=Prompt   F5=Refresh   F10=Additional parameters   F12=Cancel
F13=How to use this display        F24=More keys
```

Press Enter to add the exit program. The exit program is then added to the exit point as shown in Figure 3.

Figure 3: Exit program added



```
                         Work with Exit Programs

Exit point:   QIBM_QDB_OPEN            Format:   DBOP0100

Type options, press Enter.
  1=Add    4=Remove    5=Display    10=Replace

               Exit
             Program     Exit
Opt           Number     Program        Library

_                        __________     __________
_                  1     DBOEXIT1       QGPL

                                                                  Bottom
Command
===> ____________________________________________________________________
F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel
```

As an alternative, a quick way to attach the exit program to the exit point is to issue the following IBM i OS Add Exit Program (ADDEXITPGM) command:

```
ADDEXITPGM EXITPNT(QIBM_QDB_OPEN) FORMAT(DBOP0100) PGMNBR(*LOW) PGM(QGPL/DBOEXIT1)
```

Note: The PGM parameter must be qualified with either an explicit library name or *CURLIB (the job's current library).

Verifying the exit-point program

After you have attached the exit program to the QIBM_QDB_OPEN exit point, you can verify its effectiveness in the following ways.

- If your exit program performs any logging, check the log table.
- Check your job log. If the exit program fails for any reason (such as the program is not found, you are not authorized to the program, or there is a function to check in the program) the messages are left in the job log, but processing continues.
- Debug the job. Perhaps the best way to verify that the program is behaving correctly is by starting a debug session, setting a breakpoint, opening a table (to invoke the exit-point program) and stepping through the code in the debugger.

Securing the exit-point program

After the exit-point program is written, set up, and verified, you obviously want to secure it. Not doing so creates the possibility of exposure to exit-point circumvention. A malicious user might modify or delete the exit program, or perhaps modify the contents of the security table (and other database objects) used by the exit program. Such actions can potentially allow the user to gain access to sensitive tables otherwise rejected by the exit-point program.

Even if you do not implement object-level security for your other application objects, strongly consider using it to secure the exit-point program and all of the objects it accesses. Here is a general object-level security recommendation for the exit program and its associated objects.

- Compile the exit program with the USRPRF(*OWNER) parameter. When the program is compiled in this way, then (while running), it adopts the authority of the user profile that owns the program.
- Change the owner of the exit program to a profile with only sufficient authority to the objects accessed by the exit program. This is the profile whose authority is adopted while the program is running and allows the running program to access objects for which the specified profile has authority.
- Grant the public *USE authority to the exit-point program. This gives users the authority to call the program (but not to modify or delete it).
- Grant the user profile that owns the exit program sufficient authority to database objects and other nondatabase objects (for example, data queues and message queues) that are accessed by the exit-point program (such as the security table and any audit tables). This ensures that the program has the rights necessary to read, write, update, and delete data in those objects.
- Revoke all *PUBLIC authority to database objects and other nondatabase objects that are accessed by the exit-point program. This prevents users from accessing those objects by any interface other than the exit-point program.
- Qualify library names on dynamic calls. If your exit program makes any dynamic calls to other programs, make sure that the library name is qualified on the call statement. This prevents a program by the same name from being called if it is inserted in a library that is higher in the library list than the intended program.

Other considerations

Here are other considerations that you need to know about this exit point.

- If the exit program issues a return code of 0, the database open request fails and the following appears in the job log:

  ```
  CPF428E The open failed because exit program DBOEXIT in library
  DBOEXIT1 associated with exit point QIBM_QDB_OPEN returned a reason
  code that ended the request.
  ```

  In addition, if SQL access is used, SQL error message SQL0953 also appears in the job log:

```
SQL State: 57014
Vendor Code: -952
Message: [SQL0952] Processing of the SQL statement ended. Reason code 11.
Cause . . . . . :   The SQL operation was ended before normal completion.
The reason code is 11.
```

- If the query engine creates temporary tables during its processing, the exit program is not called for those temporary tables. This is true for all interfaces that use the query engine (SQL, OPNQRYF, and Query/400).
- If you keep up with recent enhancements to the SQL query engine, you might be familiar with Materialized Query Tables (MQT). If so, you might recall that the query optimizer has the freedom to rewrite an SQL query and can opt to use an available MQT instead of the requested table. If this is the case, the open request is issued for the MQT (instead of issuing the request for any of the tables on which the MQT is based).
- When registering or deregistering the exit program, be wary of timing. The exit program might not be called for any jobs that are started before the exit program is added to the exit point. Conversely, if you remove the exit program from the exit point, jobs already started might continue to call the exit program.
- The exit program must be defined in the system auxiliary storage pool (ASP).

Summary

Exit points have been used for years by experienced IBM i programmers to help manage their application, systems, and security policies. The exit points provide an effective way of calling a custom program whenever something specifically occurs on the system. The exit point QIBM_QDB_OPEN was introduced in IBM i OS V5R3 to assist IBM i programmers and administrators in their quest to tackle various database security issues. This article should have helped you understand what this exit point does and how it can be used to manage database security and enhance your comprehensive IBM i security policies.

Appendix

The following C and CL program code examples show how to handle the recursion.

Example that handles recursion: C program

It is possible to call C programs recursively. The following example program compares the name of the table being opened to the name of the audit log table DBLOG (to avoid infinite recursion looping). If they are not the same, the program logs the details of the open request to the audit table. The program only logs information for the first table that is passed in and does not perform any type of processing to determine if the user is authorized to open the table.

To create your exit program as described, perform the following steps:

1. Create the log table that is to be used for auditing. The log table contains details about each of the database opens that the exit program processes.
2. At the IBM i Navigator Run SQL script window, issue the following statements to create the table.

```
CREATE TABLE QGPL.DBOLOG (QRYOPN CHAR(1), USER CHAR(10), LIBRARY CHAR(10), FILE CHAR(10),
  JOBNAME CHAR(10), CURNAME CHAR(10), CURDATE CHAR(8), CURTIME CHAR(8));
```

3. Create the C module by issuing the following statements.

```
CRTSQLCI OBJ(QTEMP/DBOEXIT) SRCFILE(QGPL/QCSRC) SRCMBR(DBOEXIT1) COMMIT(*NONE) DBGVIEW(*SOURCE)
```

4. Create the C program by issuing the following statements.

```
CRTPGM PGM(QGPL/DBOEXIT1) MODULE(QTEMP/DBOEXIT1) ACTGRP(*CALLER)
```

Example 2: A sample C program

```
#include <stdio.h>
#include <string.h>
exec sql include sqlca;
main(int argc, char **argv)
{
int i;
int *returnCode = (int *) argv[2];
struct header
{
  int len;
  char eyecatcher[8];
  int fileOff;
  int numFiles;
  int fileSpecLen;
  char jName[10];
  char userName[10];
  char jobNumber[6];
  char currentName[10];
  char isQueryOpen;
  char reserved[3];
} *inHdr;

struct fileDef
{
 char name[10];
 char lib[10];
 char mbr[10];
 char reserved[2];
 int fileType;
 int lowerFileType;
 char inpOpt;
 char outOpt;
 char updOpt;
 char dltOpt;
 char reserved2[4];
} *fileInfo;

char user[11];
char qryopn;
char inputOpt;
char outputOpt;
char updateOpt;
char deleteOpt;
char filename[10];
char library[10];

char jobName[10];
char curName[10];

inHdr = (struct header *) argv[1];
fileInfo = (struct fileDef *) (argv[1] + inHdr->fileOff);

/* If the open is for the SQL table, return to avoid recursion */
/* Also, don't collect information on QSYS and QTCP users */
if (!strncmp(fileInfo->name,"DBOLOG",6) ||
    !strncmp(inHdr->userName,"QSYS",4) ||
    !strncmp(inHdr->userName,"QTCP",4))
  return;

/* Get the user and the type of access */
strncpy(user,inHdr->userName,10);
qryopn = inHdr->isQueryOpen;

/* Get the file and library */
strncpy(filename,fileInfo->name,10);
strncpy(library,fileInfo->lib,10);
inputOpt = fileInfo->inpOpt;
outputOpt = fileInfo->outOpt;
updateOpt = fileInfo->updOpt;
deleteOpt = fileInfo->dltOpt;

/* Get the job and the current name */
strncpy(jobName,inHdr->jName,10);
strncpy(curName,inHdr->currentName,10);

/* Insert the info into the SQL table */
exec sql INSERT INTO qgpl/dbolog VALUES(:qryopn,:user,:library,:filename,
         :jobName, :curName, :inputOpt, :outputOpt, :updateOpt, :deleteOpt,
         CURRENT DATE, CURRENT TIME);

}
```

Example that handles recursion: CL program with RPG procedure

If C is not your language of choice, you can still use RPG to perform most of the processing. Because a CL program can be called recursively, an exit program can be written in CL that makes a bound call to an RPG procedure. This works because it is possible to call RPG procedures recursively. For the following examples, a data area, a table, two modules, and one program were created. This exit program performs two primary functions. It prevents any non-application interface from opening the application table. And, it logs the details of the open request to a table for auditing purposes.

To create your exit program as described, perform the following steps.

1. Create a data area to hold the data-open exit-point *switch*. If the switch is on (1), processing continues. If it is off (0), there is no further action (thus, allowing the database open to occur). This added feature, though not necessary, allows you to turn the exit program on and off easily.
   ```
   CRTDTAARA DTAARA(QGPL/DBOEXITSW) TYPE(*CHAR) LEN(1) TEXT('DB Open exit point switch (1=On,0=Off)')
   ```
2. Create the table that is used to store the database libraries that need to be secure. The exit program checks all database tables in these libraries for remote access. When the exit program is called, it extracts the library name of the table being opened and looks for a row in this table with a match to that library name. If a matching row is not found, the table being opened does not need to be secured from external access and the program ends immediately. Otherwise, the table does need to be secured and the exit program continues processing. From an IBM i Navigator Run SQL script window, issue the following statement to create the table.
   ```
   CREATE TABLE QGPL.SECDBLIB (DB_LIB CHAR(10) CCSID 37 DEFAULT NULL);
   ```
3. Create the table used to store the list of users who have remote access authorization to the database libraries in the SECDBLBFL table. The exit program searches this table for a

row that matches both the library of the table being opened and the user who is issuing the open request. If a matching row is found and the remote access flag is set to 1, the user is authorized to access the database table with a remote interface. From an IBM i Navigator Run SQL script window, issue the following statement to create the table.

```
CREATE TABLE QGPL.AUTUSRFIL(USER_NAME CHAR(10), DB_LIB CHAR(10), LCL_ACS_FL CHAR(1),
  RMT_ACS_FL CHAR(1));
```

4. Create the log table used for auditing. The log table contains details about each of the database opens that the exit program processes. From an IBM i Navigator Run SQL script window, issue the following statements to create the table.

```
CREATE TABLE QGPL.DBOLOGFIL (QRYOPN CHAR(1), ACCESSED INTEGER, USER CHAR(10),
  LIBRARY CHAR(10), FILE  CHAR(10), JOBNAME CHAR(10), CURNAME CHAR(10),
  CURDATE CHAR(8), CURTIME CHAR(8), PGMSTK CHAR(1000));
```

5. Create the CL DBOEXIT1 module. This module first retrieves the data area to check the value of the database-open exit-point switch. If the switch is on, a bound procedure call is issued to the RPG CHKOPEN procedure. This module is also the module that contains the program's entry procedure. Because it is a CL program, it can be called recursively. To create this module, issue the following command from an IBM i OS command line.

```
CRTCLMOD MODULE(QTEMP/DBOEXIT1) SRCFILE(DBOEXITPT/QCLSRC) DBGVIEW(*ALL)
```

Example 3: DBOEXIT1 source code (a sample CL program)

```
PGM        PARM(&DBOP0100 &RETURNCODE)

DCL        VAR(&DBOP0100) TYPE(*CHAR) LEN(2000)
DCL        VAR(&RETURNCODE) TYPE(*INT)
DCL        VAR(&LOGFLAG) TYPE(*INT)
DCL        VAR(&MSG) TYPE(*CHAR) LEN(500)
DCL        VAR(&DBOEXITSW) TYPE(*CHAR) LEN(1)

/* If exit point switch is on, issue bound call to procedure    */
RTVDTAARA  DTAARA(QGPL/DBOEXITSW) RTNVAR(&DBOEXITSW)
IF         COND(&DBOEXITSW = '1') THEN(DO)
  CALLPRC    PRC(CHKOPEN) PARM((&DBOP0100) (&RETURNCODE))
ENDDO

ENDPGM
```

6. Create the RPG DBOEXIT2 module, which contains the CHKOPN procedures that perform the following tasks.
   - It must extract the tables that are to be opened.
   - If the table being opened is one of the three tables that the exit program opens, the return code is set to **1** and the exit program ends immediately to avoid a recursion loop.
   - If the table is in an application library that is not defined in the SECDBLIBFL table, it is not a database table that needs to be secured. The return code is set to 1 and the exit program ends.
   - The module issues an API to retrieve the current job's program stack. Each entry in the program stack is examined. If the program of the stack entry (that is currently being examined) is located in the same library as that of the table to be opened, the open request originated from the application. If none of the program stack entries are in the database library, the open request originated from a remote access request (an interface that is external to the application).
   - If the table-open request is issued by an application interface, the return code is set to **1**.
   - If all of the following conditions are true, the return code is set to **1**:

    - The table-open request is a remote access request.
    - The library of the table being opened and the user requesting the open are defined in the AUTUSRFIL table.
    - The remote access flag for that row is set to **1**.
- The details of the open request (including the full program stack) are logged to the DBOLOGFIL table for auditing purposes.
- The return code is returned to the caller. If the return code is **1**, the open request is allowed to continue. If the return code is **0**, the open request is rejected.

Note: All I/O processing in this example uses embedded SQL in RPG free-format. This is a new feature in IBM i V5R4. In prior releases, programmers had to end free-format mode to embed SQL statements.

7. To create this module, issue the following command at the IBM i OS command line.

```
CRTSQLRPGI OBJ(QTEMP/DBOEXIT2) SRCFILE(QGPL/QRPGLESRC) COMMIT(*NONE) OBJTYPE(*MODULE)
```

Example 4: DBOEXIT2 source code (an RPG example)

```
h nomain

//------------------------------------------
//  Prototypes
//------------------------------------------

d chkOpen         pr
d   dbopParm                              likeds(dbop0100)
d   returnCode                  10i 0

D rtvPgmStk       PR                  extpgm('QWVRCSTK')
D                             2000
D                               10I 0
D                                8    CONST
D                               56
D                                8    CONST
D                               15

//------------------------------------------
//  Data structures
//------------------------------------------
d DBOP0100        ds                  qualified
d  headerSize                   10i 0
d  formatName                    8
d  arrayOffset                  10i 0
d  fileCount                    10i 0
d  elementLength                10i 0
d  jobName                      10
d  userName                     10
d  jobNumber                     6
d  currentUser                  10
d  queryOpen                     1

//******************************************
p chkOpen         b                   export
//******************************************

d chkOpen         pi
d   dbopParm                              likeds(dbop0100)
d   returnCode                  10i 0

//------------------------------------------
//  Data structures
```

```
//------------------------------------------
d DBOPFile        ds                  qualified based(DBOPFilePtr)
d  fileName                     10
d  fileLibrary                  10
d  member                       10
d                                2
d  fileType                     10i 0
d  underlyingPF                 10i 0
d  inputOption                   1
d  outputOption                  1
d  updateOption                  1
d  deleteOption                  1

D rcvVar          DS          2000
D  bytAvl                       10I 0
D  bytRtn                       10I 0
D  entries                      10I 0
D  offset                       10I 0
D  stkEntryCnt                  10I 0
D  stkEntThrd                   10I 0

D jobIdInf        DS
D  jIDJobNam                    10    inz('*')
D  jIDJUsram                    10    inz(*blanks)
D  jIDJobNbr                     6    inz(*blanks)
D  jIDIntJobID                  16    inz(*blanks)
D  jIDRsrvd                      2    inz(*loval)
D  jIDThrdInd                   10I 0 inz(1)
D  jIDThrdID                     8    inz(*loval)

D stkEntry        DS           256
D  stkEntryLen                  10I 0
D  procOffset                   10I 0 Overlay(stkEntry:13)
D  procLen                      10I 0 Overlay(stkEntry:17)
D  pgmNam                       10    Overlay(stkEntry:25)
D  pgmLib                       10    Overlay(stkEntry:35)
d  fullEntry              1    256
//------------------------------------------
//  Standalone variables
//------------------------------------------
d qryOpn          s                         like(dbopParm.queryOpen)
d userNam         s                         like(dbopParm.UserName)
d jobNam          s                         like(dbopParm.jobName)
d curUser         s                         like(dbopParm.currentUser)
d fileNam         s                         like(dbopFile.fileName)
d fileLib         s                         like(dbopFile.fileLibrary)
d i               s             10i 0
d j               s             10i 0
d fullStack       s           1000a
D rcvVarLen       s             10i 0 Inz(%size(rcvVar))
D
D apiErr          s             15a
D procNam         s             10a
D rmtAcsFlg       s              1a
D dbLib           s             10a

/free
 // Don't perform checks on QSYS and QTCP users
 curUser=  dbopParm.currentUser;
 if dbopParm.currentUser = 'QSYS'
    or dbopParm.currentUser = 'QTCP';
   return;
 endif;

 returnCode = 1;
 QryOpn =  dbopParm.queryOpen;
 userNam =  dbopParm.userName;
```

```
jobNam =  dbopParm.jobName;
for i = 1 to dbopParm.fileCount;
  DBOPFilePtr = %addr(dbopParm) + dbopParm.arrayOffset +
                (i - 1) * dbopParm.ElementLength;
  fileNam =  dbopFile.fileName;
  fileLib =  dbopFile.fileLibrary;

// Avoid an infinite recursion condition by not continuing if the
// file being opened is one of the files that this program will be
// opening.
  if fileNam = 'DBOLOGFIL'   or
     fileNam = 'SECDBLIBFL'  or
     fileNam = 'AUTUSRFIL';
    returnCode = 1;
    return;
  endif;

   // See if the file being opened is in a library that has
   // been defined in the table SECDBLIBFL. Entries in this
   // table are ones that need to be secured from unauthorized
   // remote access attempts
  exec sql SELECT db_lib INTO :dbLib
           FROM dboexit/secDBLibFl
           WHERE db_lib = :fileLib;

  // If not not found, we do not need to secure this table.
  // Execute next iteration to check the next table in the list
  if %subst(sqlstate:1:2) <> '00';
    iter;
  endif;

  // At this point, we know the file being opened is one we need
  // to secure. To determine if the file is being accessed by a
  // remote interface, we need to examine the entries in the
  // program stack.
  // In this case we are going to assume that application programs
  // are located in the same library as the database files. So
  // if any of the progam stack entries has a library that matches
  // the library of the file being opened, we can assume that the
  // file is being opened via a local, application interface.
  // Otherwise the inteface is external to the appication, so we
  // need to continue processing.
  returnCode = 0;
  fullStack = *blanks;

  // Call API to retrieve the program stack
  rtvPgmStk(rcvVar:rcvVarLen:'CSTK0100':JobIdInf:
            'JIDF0100':apiErr);
  // While checking each pgm stack entry,
  // save it in varable fullStack so that the
  // full stack can be included in the log file.
  for j = 1 to stkEntryCnt;
    stkEntry = %subst(rcvVar:Offset + 1);
    procNam = %subst(stkEntry:
                 procOffset + 1:
                 procLen);
    if procNam <> *blanks;
      fullStack = %trim(fullStack) + ', '
          + %trim(pgmLib)  + '/'
          + %trim(pgmNam)  + ':'
          + procNam;
    else;
      fullStack = %trim(fullStack) + ', '
          + %trim(pgmLib)  + '/'
          + %trim(pgmNam);
    endif;
```

```
     // If the program library is same as file lib, this
     // indicates that the interface opening the file is
     // part of the secured application. Set return code
     // to 1 to allow the open request to continue.
     if pgmLib = fileLib;
       returnCode = 1;
     endif;

     offset = offset + stkEntryLen;
   endfor;

   fullStack = 'Stack count is ' + %char(stkEntryCnt) + ': ' +
               %subst(fullStack : 3 : %len(fullstack)-2);

   // At this point, if return code is 0, we know that the user
   // is attempting to access the file via an interface external
   // to the application. Check the table of authorized users to
   // determine if this user has remote access authorization
   if returnCode = 0;
     exec sql SELECT remote_access_flag INTO :rmtAcsFlg
              FROM autUsrFil
              WHERE user_Name = :userNam AND
                  application_library = :fileLib AND
                  remote_access_flag = '1';

     // If state is ok, match was found. Set return code to
     // 1 to indicate that file open request can continue
     if %subst(sqlstate:1:2) = '00';
       returnCode = 1;
     endif;
   endif;

   // Keep an audit log entry of this db open request
   exec sql INSERT INTO dboLogFil
            VALUES(:qryOpn, :returnCode, :userNam,:fileLib,
                   :fileNam, :jobNam,:curUser, CURRENT DATE,
 CURRENT TIME, :fullStack);

 endfor;
 return;
/end-free
p chkOpen         e
```

8. After the modules have been created, create the program by issuing the following command.

```
CRTPGM PGM(DBOEXITPT/DBOEXIT1) MODULE(DBOEXIT1 DBOEXIT2) ENTMOD(DBOEXIT1) ACTGRP(*CALLER)
```

Resources

- IBM i and System i Information Center
- DB2 Information Center
- IBM Redbooks

Acknowledgements

Thanks to the following people who reviewed and contributed to this article.

- Kent Milligan
- Jarek Miszczyk
- Michael Cain
- Patrick Botz

- Mark Anderson