DEPLOYING PROGRESS IN A SECURE ENVIRONMENT

Paul Koufalis
Senior DBA
White Star Software

CONTENTS

DISCLAIMER

INTRODUCTION

Over the past few years, security has become the new hot topic in the I.T. industry. Whereas in the past many companies simply trusted their employees and expected they would "do the right thing", recent troubles (Enron, WorldCom) have proven this to be a naïve point-of-view. In today's world a company must take proactive measures to ensure the safety and security of their second most important asset: their data.

Once this decision to proactively protect data is taken, the first thing one must understand and accept is that no security system can be 100% reliable because somehow, authorized users must be able to access the data. For every lock, there must be a key; and if there is a key, someone will figure out a way to pick the lock. This is a fact we simply must accept. The situation can get even more complicated: as the end user, you may face resistance from the application partner if you attempt to secure your environment in a way that is outside their currently supported deployment methodology. As an application vendor, you may face apathy and lack of interest from your client base: They simply want the fastest, easiest and cheapest implementation possible!

As a direct result of these limitations, our real objective is to find that acceptable sweet spot somewhere between 0% secure and 99.99% secure. One of the best ways to achieve this objective is to layer your security: In the same way that multiple layers of clothing keep you warmer than one thick garment, multiple layers of security provide a more challenging environment to would-be data thieves. This white paper will examine the various layers surrounding your data and discuss methods to secure each of these layers.

OBJECTIVES AND GOALS

The primary objective of this white paper is to open the reader's eyes to security issues they never even knew existed. Second, and almost of equal importance, this paper hopes to provide the reader with simple and easy-to-implement solutions for securing the various layers surrounding their OpenEdge data. The goal is for you to use the information in this document to build a comprehensive list of possible security issues and with this list, intelligently decide whether the potential breaches are below or above your current risk tolerance. If they are below, you ignore them…for now. If they are above, you can initiate an action plan to eliminate the breaches in question.

AUDIENCE

While the proposed technical solutions are mostly intended for database administrators (DBA) or system administrators, the discussions regarding the potential breaches at each layer should be of interest to both middle and upper management. By understanding the problems, management can better understand the motivation driving the I.T. department's proposed solutions.

UNDERSTANDING THE LAYERS

In order to fully understand the concept of layered security, consider the medieval castle:

If the King could afford a large number of walls and moats surrounding the interior, he and his subjects were all the more safe. But people still needed the ability to easily leave the castle to work in the fields: hence gates were installed in the walls and drawbridges over the moats. Consider the various layers surrounding your data as walls and the access paths as gates. The walls need to be strong enough to resist attacks and the gatekeepers need to be intelligent enough to detect and deny access to enemy spies.

An OpenEdge Database environment is very similar to a castle. Walls are built around the data to protect it from unauthorized access and gates are installed in these walls to allow authorized users to freely interact with the data. In OpenEdge, the basic layers are:

- The data itself
- The database containing the data
- The environment surrounding the database and user
- The filesystem containing the database files
- The server hosting the filesystem
- The network to which the server is attached
- The users who have access to the network

DATA

At the very heart of the castle is the prize itself: the data. Ideally, the first layer of security around the data should be the strongest as it must not only resist attacks from the outside but also from those who are within close proximity.

By default, OpenEdge provides security via certain fields in the _FILE, _FIELD and _INDEX metaschema tables. These fields are collectively known as the _CAN* fields as their names are of the form _CAN-READ, _CAN-WRITE, _CAN-CREATE, etc… These fields have existed since the early versions of Progress[1] and are assigned the default value "*" meaning full access to all. What does this mean? Every Progress database whose security fields have not been modified is 100% open to reads, writes, creates and deletes from any ABL client who successfully connects to the database. Note that I did **not** say authenticate but rather connect. The existence and/or use of usernames and passwords are not pertinent when the default security parameters are left unchanged. Under these default parameters, literally **all** database connection requests are granted and **all** these connected processes have unfettered access to the data.

Data access control is handled differently when an SQL connection is made. As of Progress version 9, a native SQL engine has been added and with it SQL-compliant security. In the SQL model, users are mandatory and database objects belong to the user who created them. In other words, only the object creator can read, modify, create or delete data within the object. For any other user to access the data, the owner or DBA must specifically GRANT access to that user.

Comparing the two security models, it is fairly clear that they are complete opposites: the OpenEdge default security model is "GRANT * TO * (PUBLIC)": the default user-id is "PUBLIC" and that user is granted all privileges. In SQL terms, default OpenEdge security is equivalent to executing the command "GRANT ALL ON *object* TO PUBLIC" for every data object in the database. Conversely, the SQL model is "GRANT Ø TO Ø (NOBODY)": the default user is "NOBODY" and that user is granted no privileges.

The DBA's tasks are similarly opposite when it comes to managing OpenEdge and SQL security. On the OpenEdge side, access is already set to "PUBLIC" so the DBA must modify the _CAN* fields to limit users' capacities to modify data. This is done by replacing the PUBLIC token "*" by specific usernames and/or wildcards. For example, "FIN*" means all user-ids that begin with "FIN" are allowed the _CAN* permission in question. Additionally, access can be specifically denied via the "!" token, either alone, signifying that the "BLANK" user-id is denied access, or in a wildcard expression such as "!FIN*", meaning that all user-ids starting with "FIN" are specifically denied the _CAN* access. In SQL, we explicitly execute GRANT statements of the form "GRANT *privilege* ON *object* TO *user-id."* For example, "GRANT SELECT ON PUB.CUSTOMER TO KOUP" permits the user authenticated as "KOUP" to select data from the table CUSTOMER in the PUB schema.

Another very important fact regarding OpenEdge security is that until 10.1A, _CAN* limitations were only applied at compile time. A user compiling a procedure that

[1] The author valiantly attempted to find the exact version in which these features were introduced but 20+ years and many architects and developers later, that information seems to have been lost to the ether…

modified the CUSTOMER table was required to have _CAN-WRITE access to that table. However, once the dot-r object code was generated, any user could run that dot-r and successfully modify the CUSTOMER table in total disregard to the _CAN* fields. To illustrate the potential magnitude of this feature, imagine a crafty developer who does not normally have access to production data. _CAN* security will prevent him from running ad-hoc queries as his queries will not compile. However, this same developer can compile his query against the development database to which he presumably has full access, generate a dot-r, and then successfully run the dot-r against the production DB.

To neutralize this security hole, Progress developed the DBAUTHKEY function: this function writes a unique DB id to the dot-r allowing it only to be run against a specific database. In other words, dot-r's compiled against the development database could not be executed while connected to the production database. Unfortunately, like so many other attempts to secure data, an intelligent hacker found a way to circumvent DBAUTHKEY security. Today, like the alarm on your car, DBAUTHKEY is a sufficient deterrent against unsophisticated attacks but is useless against an intelligent and determined assailant. Nonetheless, there is value in the use of the DBAUTHKEY functionality: it is extremely easy to implement and relatively tricky to circumvent. In combination with other solutions detailed in this paper, it increases the workload and complexity required to compromise your system.

The real solution to the _CAN* issue was made available in OpenEdge 10.1A: the DBA has the ability to force run-time validation of the _CAN* fields via the option "Enable run-time security". With this option enabled, every attempted ABL access to the database is validated at the time of the access, regardless of the source of the query (ad-hoc, dot-r or dynamically-generated).

Now that DBA's have the ability to validate access at run-time, there remains the enormous task of deciding which users have access to which tables. It is relatively easy to say "User x has access to Sales Order Entry". What is more difficult is to extrapolate that functional access to database access. Which tables and fields does Sales Order Entry access? Does it need to read them? Modify them? Create? Delete? What if Sales Order Entry calls other programs that update inventory, shipments, work orders or scheduling? The user must be given access to these tables as well…but only within the strict confines of the Sales Order Entry. In other words, the user can change available inventory levels by confirming the order but cannot go into Inventory Maintenance and manually play with inventory levels!

Stop and think for a minute: we have almost come around full circle! Originally, security was controlled only via the application. This was judged to be insufficient because of the many ways the data could be accessed outside the confines of the application. As a result, run-time security was implemented to control data access regardless of the source. However, the application may require that the user be given more liberal access to the data to support functional requirements. Indirect access to certain objects must be allowed because of the application whereas direct access to those objects must be barred! How do we handle this? Simple, we grant access at this layer of security and more tightly secure outer layers, effectively limiting the channels through which the user can access the data! Before I discuss exactly how, we must explore the next layer: the database.

DATABASE

There are only a few ways to access an OpenEdge database:

- Local shared memory connection
- ABL client-server connection
- SQL client-server connection
- Unified Broker (AppServer Agent, Webspeed Agent) connection

The first two have existed since version 6? 5? I don't even know! SQL and AppServer access, on the other hand, are relatively new.

To secure these access paths, OpenEdge has traditionally relied on one gatekeeper: the username/password[2]. Stored in the _USER metaschema table, these couplets are uniquely responsible for controlling access to data objects via the _CAN* fields discussed in the previous section. The current implementation of _USER has not changed significantly over the past several versions of Progress/OpenEdge. Within the database engine itself, there is no mechanism for ensuring complex passwords nor for enforcing password aging and these functions must be provided by your ABL application. Very simply, the _USER table contains amongst its numerous fields one for the username and another where it stores an encrypted version of the user's password.

To harden data access security, OpenEdge provides three additional security functions:

- The security administrator role
- Admin – Security – Disable Blank User-ID Access
- Admin – DB Options – Disable Blank User-ID

Security Administrator

Very simply, the security administrator is the only valid _USER account that can modify metaschema tables like _FILE, _FIELD, _INDEX and _USER. At the implementation level, when a valid user-id is assigned the role of security administrator via the Database Administration menus, that user-id is entered in the appropriate metaschema tables _CAN* fields. Note however that the OpenEdge "Security Administrator" is just a normal user like any other. The only difference is that his user-id was added to the _CAN* fields of the metaschema tables. Similarly, there is no special meaning attached to an OpenEdge DBA. In the SQL world, the DBA is a specific user role with special permissions to access the database. In

[2] As of 10.1A, new methods for validating users have become available. Discussion of these new methods warrants a whitepaper of its own and will be considered outside the scope of this paper.

OpenEdge, it is simply a title assigned to a person. The database itself has no functionality that recognizes a distinct "DBA" role.

Note that the _CAN* fields themselves are fields that have entries in the _FIELD table and as such have _CAN* fields controlling access to them. Only the security administrators can modify the _CAN* fields.

The same is true for _USER. Only security administrators can modify fields in _USER. One exception: Even though the _PASSWORD field of _USER has _CAN* attributes, these are ignored by the database engine. Only the user himself can modify his password. To get around this, the security administrator must actually delete and recreate the _USER record.

Finally, while this may seem obvious, always assign more than one user to the role of security administrator.

Admin – Security – Disable Blank User-ID

This option was introduced at the same time as the _CAN* fields and simply inserts a "!" character at the beginning of each of these metaschema fields. What this means is that at compile-time, or run-time if run-time security is enabled, any user who is connected to the database but has not been assigned a valid username cannot access any data.

One of the caveats of this option is that it is not dynamic: If a new table or field is created after enabling this option, it will not automatically inherit the "!" in its _CAN* fields. The DBA must manage this himself. Alternately, the "Disable Blank User-ID" option can be re-executed every time new tables or fields are added.

Admin – DB Options – Disable Blank User-ID

Very simply, this new option available as of OpenEdge 10.0A prevents a user from connecting to a database without providing proper username/password credentials. Note the difference between this option and the previous: The first allows database connection but no data access; the second simply refuses the connection. This option permits a very secure database environment. The user cannot connect with a blank user-id and then probe around for unsecured data. The connection attempt will simply be cut and no access whatsoever will be granted.

From a programming point-of-view, this makes the traditional application username/password maintenance screen more complicated. If the connection to the database is made via client-server, the programmer can temporarily store the username and password in memory and use the values to initiate a database connection. In other words, no databases were connected upon startup. In a properly secured shared-memory connection environment this is impossible as the user will not be able to connect to the database's shared memory segments after the initial _progres executable startup.

One solution to this issue is to provide a middle ground between the two "Disable Blank User-id" functions. A generic login account can be created **but that account must have absolutely no table access!** It is up to the DBA to ensure that this limitation is maintained across the lifespan of the database. Once the connection is

made, the login program can use the SETUSERID() function to change the user's identity to his real identity.

Of course, as with many solutions, there are minor problems: In order to validate the user's password via the SETUSERID() function, the generic login account must have read access to the _USER table. As a result, an unauthorized user could potentially download the usernames and encrypted passwords, run a hacking routine against them at his leisure, and then use the hacked passwords to gain illicit access to the database.

Additionally, even if this option is enabled, the client and server must exchange metaschema information, including table and field information, before validating the username/password combination. A network packet sniffer could be used to record these pre-authentication exchanges and extract valuable information from the downloaded metaschema data.

Usernames and Passwords

As we mentioned earlier, a valid username and password is the only way to gain access to a secured database. As such, we must ensure that this gate is as secure as possible.

One of the biggest violations of this rule is the use of generic user accounts. There should not be one "SecAdmin" account with the password shared by two users but rather users "KOUP" and "JACM" should be assigned the role of security administrator. This way, with the help of tools such as OpenEdge Auditing, there can be no ambiguity as to who did what.

Second, this access path must strengthened by the enforcement of complex passwords and password aging. Since OpenEdge does not yet provide this functionality, it is up to the programmer to include it in his application. When a user logs in, the login program should validate the password age and if necessary force the user to change it. Additionally, a nightly batch job can be run to change the password of users who have not used the system in x days.

Solutions

Combining the data access issues from the previous section to the username and password challenges in this section, the following solution is proposed for securing your existing OpenEdge application:

1. **DO NOT** enable DB Options – Disable Blank User-ID
2. **DO** enable DB Options – Runtime Security (as of 10.1A)
3. **DO** enable Security - Disable Blank User-ID
4. **DO** assign at least two users to the role of security administrator
5. **DO NOT** allow generic accounts
6. **DO** enforce complex passwords and password aging via your application

This solution simply states that if a user provides a valid username and password, he should have complete access to all data in the database. At first, this may sound counter-intuitive and certainly counter to the discussions in the past two sections! The reply is simple: if we cannot effectively assign access rights at the table and field level then the simplest solution is to give complete access to all **valid** users and secure the manner in which they can access the data. I.e. we secure the next outer layer.

ENVIRONMENT

In the introduction, I mentioned that for every lock there is a key, and if there is a key, someone will figure out a way to pick the lock. With that said, you nonetheless should **not** be making the potential hacker's life easier by leaving break-in tools lying around. Remember, your goal is to protect the data.

Development Environment

A data thief does not need to break into your secure production environment if you copy all the data every Sunday to refresh your entirely unsecured development environment. The would-be hacker is not necessarily looking to corrupt your data by randomly modifying or deleting rows in tables. More likely, that would-be hacker is an underpaid employee who was offered a significant sum of money to acquire your customer list or price list or worse, the employee might access financial information in order to provide insider information to a trader. That employee may not have access to that data in production but he may very well have full access to a test, train or development environment.

At the very least, test environment refreshes from production should be scrambled: change customer names; change the prices and costs of items; mangle financial information. If possible, create a standard data set for testing and use only that for your non-production environments.

Development Licenses

Progress provides three levels of database access based on the license keys in the progress.cfg configuration file found in the installation directory:

1. Runtime: Only pre-compiled dot-r programs can be executed
2. Query: Ad-hoc queries that do not modify the database can be compiled on the fly but only pre-compiled dot-r programs can modify data.
3. Full Dev: The user can execute ad-hoc queries that can read, modify, create or delete data.

There is one simple rule with regards to these licenses:

"Users with access to full development licenses must not be able to access production data with those licenses"

Often, when Progress software is installed, all the keys are entered during the same installation process. As a result, the entire user base ends up running with full development capabilities. In a secure environment, the full development license should be installed separately and read access to that configuration file should be limited to authorized users.

DBAUTHKEY

As mentioned earlier, DBAUTHKEY restricts which dot-r programs can be executed against which physical database. By using DBAUTHKEY, dot-r programs compiled

against an unsecured test database cannot be executed against a production database.

Unfortunately, DBAUTHKEY is not foolproof and a determined hacker can circumvent DBAUTHKEY security. However, this is not a reason not to use it. Remember, your goal is to make illicit access to your data more difficult.

PROPATH

An unsecured PROPATH is one of the simplest yet most overlooked security holes I have ever seen. Very simply, the PROPATH lays the foundation upon which RUN commands within the ABL find the programs they are trying to run. If the PROPATH is not secure, or if it contains a ".", it allows hackers to insert their own code in the place of the original application code simply by using the same name and directory structure as the original.

Locking down the PROPATH also discourages another common practice: developer testing in production. A developer must implement a fix but cannot reproduce the problem in production. As a result, he modifies his personal PROPATH in production to run his test version of the program as opposed to the original. The typical result: rather than fixing the original problem, the tested program caused more data corruption.

Additionally, lock down the directories and files found in the PROPATH. These directories contain all the dot-r programs and should be read-only for normal runtime users. If you lock down the PROPATH but allow a user to insert a rogue program in a valid PROPATH directory worse damage can be inflicted on the database because all users will run the new rogue program found somewhere in the global PROPATH directories.

AppServer

The AppServer is just another OpenEdge ABL client that can run programs on the behalf of a remote user. Literally, the user connects to the AppServer and says "Please run this program". The would-be hacker could drop a program somewhere on the server running the AppServer and request that the AppServer run it. Now if the AppServer happens to have a higher security access than the user (ex.: if the AppServer is running as root) then that AppServer will happily execute the rogue program on behalf of the user!

To prevent this, we of course should limit the hacker's ability to drop programs where the AppServer can see them. However, we can also limit what the AppServer is permitted to run through the SESSION:EXPORT function. This function permits the administrator to limit what programs the AppServer is allowed to execute.

$DLC

The root (or administrator) account is required to install OpenEdge products and by default access to most of its contents is open to all. There is no reason for this except to simplify the life of the data thief.

First, as mentioned earlier, install full development licenses separately and lock down read access to the configuration file containing the full development license.

Second, in all $DLC directories, lock down read access to procedure libraries to which the normal runtime user does not need access. For example, the procedure libraries containing the Progress Editor and Data Dictionary are not required by normal users. Lock down access to these files.

Also lock down execution rights on Progress tools in $DLC/bin like _mprosrv (start a database), _mprshut (stop a database), _proutil (database general utility) and _rfutil (roll-forward general utility).

Finally, lock down all the files in $DLC/properties. These contain startup information used by the AdminServer, Appservers, OpenEdge Management and other OpenEdge processes. Only members of the DBA group should be able to modify these files.

Starting Server Processes

Since Progress was installed using the root account, it is often considered the default to actually run Progress server processes using root. This should not be the case. The root account should be limited to running operating system processes only. For the database server, AdminServer and AppServers, separate service accounts should be created. Note that these should be no-login accounts. We do not want to create generic accounts to which multiple users have the password. Instead, tools like “su” or “sudo” can be used to start processes using the credentials of the service account.

This is especially true for the AdminServer as by default it will spawn it’s children (AppServer, WebSpeed Transaction Server, NameServer, etc) using the same account. If OpenEdge Management is installed, it runs in the same JVM as the AdminServer, effectively giving the OpenEdge Management Administrator full root access to the server. The same is true with WebSpeed. Anyone accessing the WebSpeed Workshop page can potentially have access to a UNIX command line with root authority.

Finally, do not forget other batch processes that may be running on your system. For example, you may be running scheduling software for your nightly batches.

Shell Access

Though it should go without saying, I feel compelled to state that no user should have shell UNIX access. The user’s startup script should immediately send them to the application. Additionally, the “exec” UNIX command should be used to start the application. Without it, a fast user can CTRL-C during the script execution and end up at the command line.

Also consider using one of the available restricted UNIX shells. In most cases the use of a restricted shell should not affect your application and if by some chance the user is able to shell out to UNIX they will be severely limited.

Solutions

Once again, your responsibility is to make the hacker's job more difficult by putting as many barriers in his way as possible. Additionally, you don't want to help him by leaving tools lying around:

1. **DO NOT** refresh development data from production data
2. **DO NOT** allow users with full development licenses to access production data
3. **DO** consider deploying DBAUTHKEY
4. **DO** remove the "." from the PROPATH
5. **DO** remove lock down access to files and utilities in the Progress installation directory.
6. **DO NOT** use root to start your database server and/or batch processes
7. **DO NOT** give users shell access

FILESYSTEM

In a client-server environment, users typically do not have access to the server containing the database, much less the filesystem. However, there are still a significant percentage of current OpenEdge deployments that are entirely host-based, with a character interface (ChUI) and a shared memory connection to the database. In these types of deployments, the filesystems containing the database, the application code and the various support files must be protected.

Specifically, users should **not** have write access to any of the files nor directories containing the OpenEdge installation, the application code nor to the database files themselves. For example, only root should have write access to the $DLC directory, with the possible exception of $DLC/properties. Similarly, only the deployment group should be allowed to write into the application code directories.

The SetUID Bit

The often misunderstood setuid bit is a UNIX-only attribute which allows the user who is executing a program to run that program with the security level of the **owner** of the program. In the case of OpenEdge, the _progres client executable belongs to root therefore all ABL ChUI users initiate their sessions as root. However, once the initialization process is complete, the _progres executable automatically downgrades its security level to that of the user.

Why is it designed that way? Simply, if correctly deployed, users should not have the right to modify physical database files and shared memory segments. This is to prevent an unauthorized user from accidentally or intentionally deleting or physically corrupting the database. However, in order to allow the _progres client executable to connect to the database nonetheless, it starts its session as the super-user. The super-user has full access to all files and is able to open the database files and shared memory segments. Once this is done, it downgrades itself but retains the open handles to the db files and shared memory.

Securing the Database

At the very least, all database files and directories should belong to a service account (ex.: prodba) and only that account should be allowed to read or write into those directories and files. As explained above, users who connect to the database at startup will not have any problems making a shared-memory connection. However, users who attempt to connect to the database after the session startup will face a permission-denied error and be forced to connect via client-server.

Note that both the files **and** directories must be write-protected. Though counter-intuitive, file modification is controlled by permissions on that file but file creation and deletion are controlled by permissions on the parent directory. In other words, it is entirely legal to delete a file that you cannot modify if you have write access to the directory without having write access to the file itself.

Many people will only secure the write permissions of the database files without blocking read access. This allows unauthorized users the ability to copy the database to a new location where all files now belong to them. From there, it is a simple task

to edit parts of the database using a hexadecimal editor and disable all or part of the internal OpenEdge security.

We already mentioned securing the files and directories containing the application code, the Progress installation directory and now the database extents directories. You should also secure any directory containing parameter files (PF) or scripts. Changes to these files could cause an authorized user to run commands he never intended to run.

Solutions

1. **DO** remove all read, write and execute permissions on all the database extent files and directories

2. **DO** remove write access on all the application code directories

3. **DO** remove write access on all complimentary files and directories (ex.: PF files)

4. **DO NOT** distribute source code to the production servers. Generate r-code on your development server and distribute only the dot-r's to the production server.

SERVER

As we mentioned earlier, in a client-server environment, the user should have no access whatsoever to the production database server except through the pre-defined ABL, SQL and AppServer ports. Local users should only be able to access the server through a terminal session. Everything else should be closed unless otherwise needed for a specific and well-defined purpose. Unfortunately, this is not usually the case. Type the command "netstat –a | grep LISTEN" on your UNIX or Windows server. All those listeners are awaiting connections from the outside.

Additionally, by default most servers are accessed via non-encrypted utilities such as telnet and ftp. It is a relatively simple task to sniff for plain-text packets on a network and extract a user's username and password.

Finally, many servers are running file-sharing utilities like Samba. While Samba might seem simple to install and get up and running, it is by default fairly non-secure. A simple example of this is where administrators install Samba so that users can generate reports to text files and then open those text files via Windows Explorer. They therefore give all users access to their home directories on the UNIX box. However, this also gives the user the ability to modify his .profile startup script and give himself shell access.

If not configured properly, Samba access can also give a hacker the ability to drop rogue dot-r programs into PROPATH directories to which he normally would not have access. Or he may be able to physically access the database files.

Solutions

1. **DO** stop all network services that are not required.
2. **DO** replace all standard utilities by their encrypted versions
3. **DO** use file sharing tools like Samba with caution and test extensively

NETWORK

The outermost circle of defense is the network. By correctly configuring network access, the system and network administrators can completely control access to the server containing the database files and manager. The easiest way to do this is through the implementation of Virtual Local Networks, or VLANs.

VLANs allow the network administrator to segregate network traffic destined for the server from all other network traffic. Using enterprise firewall software like Checkpoint, he can also specify which computers on which other VLANs can access network resources on the database server VLAN. For example, the network can be configures such that the database servers are in VLAN 10.10.10.x; all finance users are in VLAN 10.10.20.x; and all warehouse workers are on VLAN 10.10.30.x. Via the firewall software, the administrator can specify that workstations in the 10.10.20.x VLAN can access the 10.10.10.x VLAN and deny access to workstations in the 10.10.30.x VLAN. Note that this does not prevent unauthorized use of a workstation in Finance's 10.10.20.x VLAN. If the Vice-President of Finance goes home in the evening and leaves his workstation unlocked, the janitor could potentially have access to all the company's financial data!

Open network ports can also give a potential intruder access to try and hack his way into the database. Normal connection attempts via the _progres executable are handled and secured by normal OpenEdge security features. Conversely, network access directly to a database server's or broker's TCP/IP port cannot be controlled so easily. This hacked connection will likely not follow the standard handshake protocol and thus will not necessarily be disconnected. It cannot be concluded with certainty that useful information could be extracted from the database in this way but it still remains a possibility. All that is required is a hacker with the right tools and motivation to try.

Additionally, as we mentioned earlier, workstations or servers with full development Progress licenses should not have access to the production database servers. The easiest way to accomplish this is again through VLAN segregation. If a person serves the dual role of development and support, that person could easily access production data through a separate computer or via some Citrix-like remote desktop software. Considering that a more-than-adequate workstation costs less than a thousand dollars, there is really no argument against installing two workstations to support the user's two roles.

Solutions

1. **DO** segregate and control access to database servers via VLANs

2. **DO NOT** allow workstations or servers with full development access to production database servers.

USERS

It's sad to say, but your very own employees are most likely the biggest potential security hole in your organization. No matter how much you're paying them, somewhere there is someone who could use an extra few thousand dollars. And if all they have to do earn that money is extract a little data from your databases, what's going to stop them? But even if all your employees are completely honest and trustworthy, there is still the very real chance that human error will somehow corrupt your data or make it available to the wrong people. The perfect example: a developer who has multiple windows open on his desktop, some in production and some in development. It is not difficult to believe that this developer might accidentally run something in production when he meant to run it in development.

The first solution to this problem is to segregate the various support and deployment roles in your organization. The person who fixes a problem should be different from the person who tests the fix and still different from the person who deploys the fix in production. This way, there are multiple people involved. If the intent is malicious, more than one person must be persuaded to risk his career for a sum of money. On the other hand, if the intent is honest, there is more chance that an error will be caught if multiple people are involved in its deployment.

And of course there is the example mentioned above in the network section: users who do not lock their workstations. It only takes a minute or two to sit at a desk and print or email sensitive data to which a thief would not normally have access. What makes this issue most frustrating is that there is no policy easier to implement yet more difficult to enforce!