Firebird File and Metadata Security

Geoff Worboys

7 December 2005 – Document version 0.5

Table of Contents

Introduction

If you stumble on this page and don't know about Firebird, see this link: www.firebirdsql.org

This article discusses the security of Firebird database files and in particular access to the metadata stored in those files. It has been written in response to frequent questions on the Firebird lists concerning these issues. The article avoids technical specifics. To discover how to secure files in your particular operating system refer to relevant documentation for that operating system, to discover how to apply user and object security in Firebird databases refer to documentation available from the Firebird website (noted above) or buy *The Firebird Book* by Helen Borrie.

Background

For an application (user) to access a Firebird database it must connect to the Firebird server process. On receiving a connect request the server process will authenticate the user credentials against the server users defined in the security database. If authentication is successful the server will allow the application access to any database that it requests and then use the roles and privileges defined in that database to provide fine grained access control to objects in that database.

At no time does the user connecting to the database require direct access to the database file itself. All access goes through the server process, which accesses the database file as needed to fulfil requests. It is the server that restricts or allows access to the logged-in user, according to the permissions defined for that user in that database.

> **Note**
>
> The “embedded” variation of Firebird server works differently and would not be appropriate for secure installations. The fact that the embedded server version exists does not really change the security issues discussed this article, it merely highlights the importance of using effective environmental security in secure installations. Embedded server is discussed in more detail later.

Every Firebird server installation has a “SYSDBA” user, this user has unrestricted access to any database available to that server. Database specific privileges are effectively ignored when the database is accessed using SYSDBA. When you first copy a database onto a server you can use SYSDBA to customise the privileges according to the users and requirements of the server. However it also means that, if I have direct access to a database file, I can copy that file from a server where I may not know the SYSDBA password onto a different server where I do know the SYSDBA password and so gain unrestricted access to the database.

As you can see from this description, Firebird security is predicated on the assumption that the Firebird server process will be running in an adequately secure environment. Firebird itself takes no precautions to provide external security. Once a person has physical access to a database file there is no effective way to prevent that user from reading all data (and metadata) within that file.

> **Note**
>
> “Adequate” security is dependent on the level of security required for a particular installation. This will vary considerably from installation to installation.

This means that, for reasonable security, installations should ensure that the database files are adequately secured. In most cases this means that the Firebird server process should run as its own specific operating system user and only that user (and probably the administrator/root user) should have direct operating system access to the database files. The database files should not be located in directories that are accessible from the network, or by any local users other than specifically authorised personnel.

For the security to be effective it is also preferred that the server computer is stored in a secure location to prevent physical tampering that could permit unauthorised users to access the hard disk from beyond the strictures of the operating system.

However, the above explanation does not necessarily help developers who, having written databases for distribution and installation to remote sites, may wish to protect the intellectual property carried in their databases. Such concerns may include specifically the metadata (table structures and relationships, stored procedures and triggers) but may also include specific data carried in some tables. These concerns represent the main purpose of this article.

The Problem

A developer creates a database (and usually an accompanying client application) for installation on servers at remote sites. At such sites it is usual for a person at that site to have full access to the computer on which the Firebird server is running – in order to be able to perform backups and other maintenance tasks. As described in the background information, direct access to the database file provides the ability to gain full and unrestricted access to all the information in the database – both data and metadata.

In such cases the developer may not trust the users at these remote sites to keep the intellectual property represented by the database confidential. The fear may be that the users will reverse-engineer the database for their own purposes, or that these remote sites will fail to maintain the security necessary to prevent unauthorised access to the database.

This leads to the common questions on the Firebird lists along the lines of:

“I want to—”
> “—protect my database design (table structures, stored procedures, triggers etc.) from all users of the database at a remote installation. How can I do this with Firebird?”

“I want to—”
> “—stop any users at a remote installation from accessing these particular tables of data. They contain proprietary information used internally by the application.”

Firebird (at least to v1.5) provides no built-in encryption facilities. The simple answer to both of these questions is that it cannot be done using the current capabilities of Firebird. A user who can get direct access to the file gains access to all details within that file.

In the first instance no workaround is feasible because the server itself must be able to read the metadata. In the second instance it may be possible to implement client side encryption/decryption features, but then you will lose the ability to make effective use of database indexes and search facilities – and key management remains a major problem (more below).

The Solution

There is really only one possible solution to these requests: Host the database and server at your own site and let the clients connect to your server remotely, through dial-up or Internet facilities etc. Terminal server (Windows or Linux/Unix) capabilities could be a useful way to implement such requirements.

In this way you maintain control over the database file and can restrict access to the various features and structures of your database using the usual Firebird internal security features (roles and privileges, etc.).

Difficulties

It is worth pointing out that there are difficulties even in this situation, if your intention is to protect the structure of your database.

Needs of the Access Layer

Various database development libraries interrogate metadata, such as primary key, domain and similar structural information, in order to make development of client applications easier. Consequently, you may discover that you cannot prevent users from accessing metadata without also preventing your application from gathering the information that it requires.

This may mean that you will need to choose between allowing metadata details to escape from your server via a sophisticated data access interface and spending the considerable extra time it takes to develop an application using a less sophisticated access library.

“Leaking” by Inference and Deduction

There is also the issue that most client applications inherently “leak” structural information about the database with which they interact. It is very rare for a database-centric application to have an interface that does not reveal many details about the table structures that it uses.

Some details may be hidden behind views and selectable stored procedures, but defining such features purely to try and hide structural information is an exercise in frustration. It is probably futile, anyway, since some details will escape, whatever you try.

Protecting User Data

Before continuing with other discussions relating to encryption of Firebird data, I do want to highlight that it is possible for users to protect their databases with encryption. This does not help developers who want to hide information from legitimate users, but it may help to meet the requirements of customers wanting to increase the security of their databases.

In some office situations it may not be practicable to locate the Firebird server computer in a truly secure environment. During times when the office is attended the likelihood that anyone will be able to access the computer

to copy the database files (or steal the computer or hard disk to get the files later) may be quite low. However out of normal working hours (nights and weekends) it may be a different matter. Someone could gain access to the office, take the hard disk out of your computer (or take the entire computer) and take it away to access the database.

Encryption

While Firebird itself provides no built-in encryption features there are some excellent products that do. You could install software that creates an encrypted volume on your computer and locate the database file (and any other confidential data) on that volume. When the computer is shut down all data exists in an encrypted file and cannot be accessed without the key. When you start the computer you have to mount the encrypted volume and supply the secret key before the data can be accessed. This additional, and necessarily manual, step in the start up process may be inconvenient but it can provide excellent security for unattended computer systems.

Software with these capabilities includes: TrueCrypt (www.truecrypt.org), Bestcrypt from Jetico (www.jetico.com) and PGPDisk (www.pgpi.org/products/pgpdisk/ – note that this link goes to an old freeware version, that site has links to newer commercial versions of the product). There are others but the last two are ones that I have used myself.

Why doesn't Firebird provide encryption?

Because of the needs described above it is common for users to request that Firebird should, in a future version, add the ability to encrypt metadata, selected user data, or even the entire database. Not being a Firebird core developer, I cannot say categorically that it will not happen. However, the issue is not really whether encryption is practicable or useful, but a matter of whether key management would provide a solution to the problems we are examining.

Encryption can only be as good as the secret key required for decryption. It can be worse but it cannot be better. There are several excellent encryption algorithms available that could be used. When good encryption is used, attacks are likely to be against the key rather than against the encryption itself.

How could encryption work?

So let's look at how things would work if Firebird were to encrypt the metadata in a database...

Before the database could be accessed the secret key would need to be supplied. Giving the decryption key to the user would be pointless, simply bringing us back to the original problem. So, presumably, whenever the customer restarts the server they would call the developer who would then dial in and enter the needed key. Even if this were practicable, it is not necessarily going to solve the problem. For example; the customer could install some monitoring software on their server to detect the key as it is entered.

There are hardware based solutions to provide a key to a decryption process. But again this would need to be in possession of the client, and if we don't trust the client we cant stop them from using it to gain access to the database from another server where the SYSDBA password is known.

Firebird is an open source product. If the encryption facilities were built in, or open source plug-in libraries were used, it would be feasible for users to build their own versions of the server or plug-in that not only performed the necessary encryption and decryption to access the protected database but also output the key, or simply output the decrypted details directly. The developer, not being in control of the server, can neither detect nor prevent such activity.

You might consider building your own version of the Firebird server with the decryption key hidden in the executable. However, decompilers are available. It would not take long to discover the key simply by comparing the decompiled versions of your custom Firebird build with the normal, unencrypted version.

Various database products do exist which purport to provide strong encryption. Perhaps the encryption is strong but, unless the key management is in place to support this feature, the encryption is not going to achieve the desired effect. It may encourage you to believe you are protected, but you need to study the key management to discover if this is really true.

The *painful truth* is that, once you lose control of the hardware on which the encryption and decryption takes place, all bets are off. If the decryption key cannot be kept reliably secure then even good encryption becomes little more than security by obscurity.

Limiting the distribution of data

Some people request encryption of the database data so that they can try and limit the dissemination of data. They are happy for the particular authorised user to see the data, but they wish to limit that user's ability to distribute the data to other people.

Just imagine for a moment that all the key management problems described above have been solved, so that it has become impractical for the user to just copy the database. In such cases the user would simply write a small program that extracted the data they were interested in (from the legitimately installed server) and copied that data to its own file or database.

I guess it is possible that Firebird might provide some form of application authentication system in the future that may make it possible to limit this form of data extraction, however most of the same problems exist. If you do not control the server you cannot prevent the user from installing a version of the server that does not require the authentication.

Removing SYSDBA access

At various times people have suggested that removing SYSDBA access to a database could be the solution. The idea behind it is that, when the database is moved to a new server where the SYSDBA password is known, it will not help the person because SYSDBA does not have access anyway. Some have reported limited success in this respect by creating an SQL role name of SYSDBA and making sure it does not have access to the database objects.

However it does not really solve the problem. The database file can be viewed with a hex viewer or similar utility and the list of available user names discovered. (Discovering the owners of the database objects would be particularly useful.) Once known, these names can be added to the new server and used directly.

An even simpler workaround might be to use the embedded version of Firebird server (see below) or to compile your own version of the Firebird server that ignores security constraints.

Custom names for SYSDBA

There has been some suggestion about allowing the SYSDBA user name to be changed. This may offer some limited protection against brute-force network attacks against the SYSDBA password, since such attacks would need to guess both the user name and its password, but it does not help protect the system from a person with direct access to the database file.

Deleting stored procedure and trigger source code

When you write and define a stored procedure or trigger for a Firebird database, the server stores an almost complete copy of the procedure source code along with a “compiled” copy referred to as BLR (Binary Language Representation). It is the BLR that is executed by the server, the source code is not used.

Some developers attempt to protect at least some of their database metadata by deleting the source code from the database before distributing the database (a simple direct update against the relevant metadata table fields). I recommend that you don't do this for two reasons...

1. BLR is a fairly simplistic encoding of the source code. It would not be difficult to decode the BLR back to a human readable form. Such a decoding would be without comments and formatting, but the SQL that goes into a stored procedure or trigger is rarely so complicated that this would cause much of a problem. Hence the protection offered by the removal of source code is not very significant.

2. The source code can be useful for other purposes. It allows fixes to be applied directly to the database without needing to bring in the full source from elsewhere (and then remembering to remove it again when the fix is applied). The source code is also used by various utilities, such as my own DBak application – an alternative backup program to “gbak”. I have not bothered to write my own BLR decoder at this stage, so DBak relies on the availability of the source code in order to be able to build a DDL script to reconstruct a database.

Embedded Firebird Server

There is a special version of the Firebird server referred to as “embedded”. This is a special client library that includes the server itself. When an application links to this library it loads the server and allows direct access to any database that is accessible on the local computer. This version of the server does not use a security database. The user name specified during the “logon” (no password authentication occurs) is used to manage user access to database objects (via SQL permissions) but if that user name is SYSDBA (or the owner of the database) then unrestricted access is possible.

The features of embedded are useful for developers wanting to create easy to distribute single-user applications that do not need security.

From that brief description it appears that having an embedded server client installed on a server hosting other databases could present itself as a major security risk. In reality the risk is no greater than if the embedded client did not exist.

When an application loads the embedded server, the server operates in the application (and therefore the user's) security context. This means that the embedded server will only be able to access database files that the user could access directly through the operating system. Giving an untrusted user access to install programs on a secure server is bad news in any case, but provided you have specified appropriate file permissions on secure databases, the embedded server itself is no threat.

The threat comes from all the other things that the user could install.

The fact that the embedded server exists only serves to highlight what is possible given direct access to a database file, especially in an open source environment. If it did not already exist then it would certainly be possible for someone to compile an equivalent capability.

Other Forms of Obscurity

Various other forms of security by obscurity have been proposed. Such things as special events that fire on login and log off to call user functions to prevent or deny access. Such features may offer some limited use for closed source systems, where the obscurity of the implementation helps to hide exactly how information is being protected. But for an open source system the work around for such hacks is to simply build your own version of the server that bypasses the event or code which is preventing access. It is difficult to offer obscurity in an open source system.

Consider also what happens when you distribute your compiled executables. Compiled programs are great examples of obscurity. No encryption is used (usually), all steps of the code are there to be analysed by anyone with the time and knowledge and, indeed, there are decompilers available to assist with this process. Once a person discovers what libraries your code was compiled with, isolating the results to only your own “secret” code makes the whole process much faster. Have you written to Borland, Microsoft or whoever requesting that they somehow encrypt their compiled binaries?

Acceptable Low Security

My comments so far have been directed at the idea of strong security and I guess the concept of security by obscurity has been written with some contempt. However at times weak security is all that you want. Sometimes the data is just not that valuable. You want to stop the casual browser and make it at least inconvenient for the more advanced thief.

I have used such schemes myself in various places. Often there is no point in throwing Twofish, AES or whatever at such schemes because those are all about strong encryption. They are heavy with processing overhead and complication relating to keeping the security strong. A simple XOR against some known string (the key) may be sufficient. If the key can be discovered by the thief then it does not matter whether you have used weak or strong encryption, the game is over anyway.

> **Note**
>
> Most simple XOR based algorithms can be broken with little effort. Consult a good encryption reference for more information and other options.

Choosing Obscurity

The thing about security by obscurity is that it must be obscure! If Firebird were to implement some sort of encryption into its disk reads and writes then it would not be obscure because it is an open source project. It would take almost no time at all to recompile the source to discover the key being provided and everything is lost.

So if you really needed this feature you would obtain the Firebird source, insert your own obscuring code into the disk read and write methods and compile your own variation of the Firebird server. (Such code could be discovered by decompiling the executable but it does take a fairly serious thief to try this.)

Before you do this, try to work out whether it would actually solve your problem, if the user also takes a copy of the specially compiled executables along with the database; or if it remains possible for a user to extract the secrets directly from your running server.

The Philosophical Argument

There is also the philosophical question of why you would choose an open source database server product to build a closed source database. Many people have contributed to the project in the firm belief that open source is the best way to provide software.

But, more particularly, when it comes to the storage of users' data I am a firm believer that the users should insist on the ability to access their own data – which will often include the need to understand the structures and the processes you have built (the metadata). If you go out of business or become otherwise unavailable it may be of critical importance that the users can at least extract their own data (in appropriate formats) in order to be able to move to alternative systems.

Can you trust the users to respect your intellectual property while you are still in business and available? Provide the necessary services and facilities and hopefully they will. If not, there is a good chance that there is little you can do to stop them.

Conclusions

The problem has been that too many people do not understand security and how difficult it is to do well. Regrettably there have been many software products that have encouraged such misunderstandings by implementing obscurity rather than true security. Witness the number of companies around that provide “data recovery” services, by which they mean *bypassing or breaking the supposed security of obscured data*.

Encryption is not a panacea for security. If you are not in control of the environment (the hardware, the operating system and all software running on that system) then you have no control over the security – regardless of what encryption schemes you may have in place. This is the situation when you distribute your database to remote server installations.

If you really need to protect the data or metadata in your database then you will need to retain control of the database file and the environment in which it is accessed. No other solution will offer you the same level of security.

Acknowledgements

I would like to thank the various people that have reviewed and commented on this article. I would also like to thank the many people that contribute to the Firebird support list, which is the source of much of the information that appears in this article.

Appendix A: Document History

The exact file history – starting at version 0.5 – is recorded in the `manual` module in our CVS tree; see http://sourceforge.net/cvs/?group_id=9028

Revision History

N/A	14 Feb 2005	GW	First edition.
N/A	11 Apr 2005	GW	The section on “Acceptable low security” was reviewed to try and highlight simple XOR algorithms as weak to ensure that readers investigate further if interested in this approach.
N/A	26 Apr 2005	GW	Additional section on Embedded server (and references to it). Moved footnote into an italic note, footnotes don't work well with HTML. Added a TOC.
N/A	4 Dec 2005	GW	Added reference to TrueCrypt. Added *Use of this Document* section. Added an Acknowledgements section.
0.5	7 Dec 2005	PV	Moved *Document History* and *Use of This Document* into appendices. Added version number for use within the Firebird project. Added document to Firebird CVS repository.

Appendix B:
Use of This Document

I have tried to make this document accurate at the time of writing but I cannot guarantee that there are no mistakes. Security is a complex subject, where security is important to your product or installation you should seek professional advice.

I place no particular restrictions on the use of this document. You are free to reproduce, modify or translate this document. However, altered versions of the document should be annotated with the changes that have been made and the name of author of the change (so that my name is not associated with text that I did not write). —G.W.

Security in Firebird 2 (All Platforms). Be aware of the following changes that introduce incompatibilities with how your existing applications interface with Firebird's security: Direct connections to the security database are no longer allowed. Apart from the enhancement this offers to server security, it also isolates the mechanisms of authentication from the implementation.Â Also read the file security_database.txt in the upgrade directory beneath the root directory of your installation. Trusted Authentication on Windows.Â This affects not just user data but also the metadata stored in the system tables. There is a metadata script to enable you to upgrade the stored sources of triggers, stored procedures, views, constraints, etc. The metadata file of a Firebird vault is by default stored under the following location on the server computer: C:\Program Files\M-Files\Server Vaults\\MetaData. Do the following steps to check the size of the metadata file: Open M-Files Admin. In the left-side tree view, expand the desired connection to M-Files Server. In the left-side tree view, right-click the document vault of your choice and select Properties from the context menu. Result:The Document Vault Properties dialog is opened. Open the Advanced tab.Â Result:The metadata file location along with the file data location is shown at the bottom of the dialog. Open the metadata file folder you have located in the previous step and select the metadata file. Check the file size in the status bar.