Global Information Assurance Certification Paper

Building a tripwire System for SQL Server

Frank Ress

GIAC GCWN
Practical Assignment
Version 5.0

Option 2: Topics in Windows Security

Contents

References

Abstract

Tripwire is a well known host-based Intrusion Detection System (IDS) that is available for a wide range of operating systems in both commercial and non-commercial versions[1]. It was first released as a non-commercial UNIX version in the fall of 1992. Both the opensource and commercial forms of Tripwire, as well as similar products, are in wide use to today [3, 9].

The traditional file-based approach to tripwire implementations is ineffective for modern relational databases, such as Oracle or SQL Server, however. In order to effectively apply a tripwire-style approach to database objects, it's necessary to use database tools to apply tripwire processes to the contents of the database files.

This paper will discuss the requirements for such a tool, and provide design and implementation details to illustrate how it could be constructed and used. Sufficient detail and sample code will be provided to demonstrate a proof of concept for such a tool, and to build a framework for further work.

Introduction

Tripwire allows system administrators to verify the integrity of the file systems of the computers they manage, by 'fingerprinting' files and providing utilities to compare these digital fingerprints over time. By automating the process of auditing changes to the files (creation, deletion, and modifications of contents or file attributes), tripwire assists the sysadmin in the identification of any unauthorized or inadvertent changes to the monitored files.

One of the drawbacks to tripwire is that it's ineffective when applied to frequently changing files. That isn't a serious limitation in its traditional application – the system files that are most susceptible to attack tend to be relatively static. The sysadmin can configure tripwire to confine its activities to these sensitive files and directories, and ignore the 'noise' from relatively more volatile and less important objects in the filesystem.

[1] The name Tripwire is both a registered trademark of the company that sells the commercial version of this product, Tripwire, Inc., and a generic name for the opensource version of the software that's available for the Linux operating system. Furthermore, the term can also be used to describe the generic process of applying a hashing algorithm to some object, with the goal of producing a digital fingerprint for the object at some point in time, which can be compared to a similarly generated hash at another point in time to detect changes to the object. Unless otherwise noted, use of the term in this paper will refer to the generic process, and will be denoted by use of all lowercase (i.e. "tripwire") *including titles, headers, etc.*

The container files used by most Relational Database Management Systems (RDBMSs), however, typically store both application data (i.e. user data) and RDBMS structural/organizational data in the same files. Information about the database structure, tables, accounts, access rights, etc., is stored in the database (what is commonly referred to as system data or the data dictionary) along with the application data. It would be desirable to be able to monitor the data dictionary for changes, but using the traditional file-based implementation of tripwire is impractical due to the relatively high rates of change in the application data. Even if the system and application data could be isolated in separate container files, the database 'system' file would encompass too many objects for the database administrator (DBA) to be able to effectively follow up on a tripwire alert on such a file.

In a sense, the database is an analog of the operating system, with its system and user files. To be useful, tripwire in a database environment would have to provide the same fine-grained configuration and inspection capabilities within the database as the traditional tool provides in a filesystem, in order to target and monitor just those sensitive objects that are of interest to a DBA.

Problem Definition

Identification of Vulnerabilities

Database vulnerabilities

The "crown jewels" of many an organization is stored in their computer databases. In most companies, these are modern relational databases, the likes of Oracle, SQL Server, and DB2. This is particularly true of structured information, like financial data or customer information.

Structured data is what one would naturally tend to visualize in terms of tables, columns, and rows. (Unstructured data is typically file-centric, like word processing documents.) If the picture that comes to mind is that of a spreadsheet, then you're dealing with structured data. Orders. Credit card numbers. Addresses and phone numbers.

Intruders are thinking about it, too.

According to Dave Thomas, head of the FBI Computer Intrusion Section [6]:

> *The most damaging [thing] is the access to databases, if [an intruder] breaks into a company's computer and steals their databases of information.*
>
> *Anyone with a database is at risk. Anybody that has something that I can buy, sell, trade or barter on the Internet.*

Modern information security environments are built on a foundation of layered defenses – the principle of defense-in-depth. Typically, perimeter defenses such as firewalls and Intrusion Detection and Protection (IDP) systems are deployed on connections between the internal and external network. In many cases, administrators will augment these perimeter defenses with host-based defenses (such as host-based firewall or IDS tools) on any or all systems within the internal network. The advantage of such layered defenses is that if one layer is breached, the defenses of another layer will still be available to thwart the intruder.

When well configured by the administrator, these tools will be set to reject all but specific, necessary traffic. As a last resort, a tool such as tripwire – which allows the administrator to detect when unauthorized changes to the system have been made – can at least alert the administrator to a system that has fallen into the hands of an intruder. As Eric Cole said in the first computer security course taken by this author, "Prevention is ideal, but detection is a must." [4]

To successfully penetrate such well-managed environments, attacks through the protocols left open for legitimate traffic may be the most attractive alternative to the intruder. Even better (for the intruder), these routes may lead directly to the highest-value assets on a system. Why go to the trouble to compromise the operating system to get to the database, when it's possible to compromise the database directly? Such an attack would bypass network-based defenses like firewall routers, as well as a host-based IDS like tripwire, which cannot effectively monitor the database container files.

One example of such an attack is SQL Injection through a website (see, for example, Strawmeyer [20]). Many applications provide access to a database through a web interface. The security infrastructure (firewall, IDP, etc.) has to be configured to allow the http or https traffic necessary for the application to function. The security for the database in such a system would be extremely difficult to build into the traditional perimeter or host-based defenses. In practice, the security of the database depends on the application developers to properly filter and verify the requests that are passed from the web application to the database for processing.

Another type of traffic that is generally allowed to traverse an organizations information security defenses is Email. SQL Server is able to send and receive automated Email messages (so called SQL Mail). This author examined the security implications of this capability in a prior report [14]. While probably not in wide use, SQL Mail has been the subject of at least one "how to" article this author has recently come across [5]. Fortunately, the author of that article was sufficiently aware of the potential for misuse that he briefly mentioned the security implications of incoming message processing by the database. Nevertheless, the SQL Mail feature presents another example of a vulnerability

that could be used to directly target a SQL Server database.

If the database is compromised, absent any type of intrusion detection system, what tools are available to the DBA to detect the break-in? The standard means consist mainly of log files and audit records (if auditing has been enabled by the database administrator).

Most modern relational databases – SQL Server included – provide audit capabilities. In the case of SQL Server, Microsoft provides the SQL Profiler tool for collection and analysis of audit data, and C2 security can be implemented. Even in a non-C2 environment, administrators can configure a database to record any or all changes to data. This type of auditing can be very fine-grained if the administrator desires – for example, recording all logons, database account creation or modification, auditing all changes to the data in a particular table (including capture of the SQL statement executed), recording deletions from a table, etc.

In practice, there are at least three disadvantages to auditing. First, auditing itself adds to the processing overhead on the database. If performance is an issue (probably the norm), auditing will certainly add to the overhead and decrease overall performance of the database.

Second, the DBA has to make some determination, in advance, what operations to audit. A detailed audit configuration, while able to target just high-value resources for monitoring, represents a significant administrative cost to set up and maintain. This approach minimizes the performance cost of auditing, but the trade off is higher administrative cost. The alternative is to audit more widely, but that would needlessly erode performance (and increase the storage required for the logs) for no real benefit. According to the Microsoft SQL Server Books Online (BOL) [13],

> *Auditing can have a significant performance impact. If all audit counters are turned on for all objects, the performance impact could be high. It is necessary to evaluate how many events need to be audited compared to the resulting performance impact.*

Third, the audit records would need to be examined to determine if any unauthorized activities had occurred. Again, depending on the volume of activity being audited, the effort could be significant.

> *Audit trail analysis can be costly, so it is recommended that audit activity be run on a server separate from the production server* [13].

In addition to auditing, most relational RDBMS products, including SQL Server, are capable of being restored to a point in time backup and replaying the subsequent transaction history. In this way, it's possible to recreate the

operations that were applied to the database in sequential order. Assuming a backup exists that predates the intrusion, and that the intervening transaction logs exist, a skilled database professional could, in theory, recover all or most of the legitimate activity and correct the intrusion.

In practice, this would probably be too complex and costly to be practical. It assumes some knowledge of when the intrusion took place, and it would probably be impractical to take the database out of production for the time required for this kind of recovery. It would also take a skilled administrator, who could isolate and remove or undo the unwanted activity. There are commercial products available that would aid in the inspection of these logs [2].

Nevertheless, it should be recognized that transaction logs that can be used for recovery could also be used to provide a record of intruder activities. Not being designed for that latter purpose, however, the practicality of such use is questionable.

The feature that the standard audit/logging tools have in common is that they are activity-oriented. They're designed to record *actions* in a more or less sequential order for later review, rather than compare *objects* for evidence of change.

Inspecting the audit or logging records for evidence of problems requires at least some a-priori knowledge of what to look for (either in configuration of the auditing to limit the information to what's relevant, or to wade through the voluminous output if one elects to preserve complete activity records). The classic tripwire approach simplifies analysis of operating system files by focusing on what has changed. How can this be adapted to the RDBMS?

A key concept for SQL Server (and, indeed, for most, if not all, modern RDBMS systems) is that the database stores information about its own structure and organization within database structures themselves - the 'system tables', or 'data dictionary'. There are records that describe table structure, indexes, stored code modules, account information, etc., and these structures can be queried to reproduce the original code that created the object. Why not reconstruct these SQL commands from the dictionary information, then pass them to a tripwire tool to allow us to compare digital fingerprints over time?

Vulnerabilities introduced by the tripwire solution

A security application must be trusted to have any value. If the tool can be compromised, it will provide a false sense of security. It's a worse situation than having no tool at all. As much as any application, a security tool has to be evaluated to identify its vulnerabilities. Action must be taken to reduce the vulnerabilities of the tool in any way that's reasonable and practical.

This issue will be discussed in detail after the implementation has been more

completely described.

Solution Strategies

The goal is to provide a toolset to the database administrator that will allow him/her to detect unauthorized changes to database objects (tables, views, code units, etc.), in much the same way as tripwire utilities do for system files. There are at least two approaches to a solution.

The first is to query the data dictionary for the object-creation code, write the resulting statement to a file on the host operating system (the 'external' approach), then use a traditional file-based tripwire to fingerprint the files created. The other is to pass the object creation statement retrieved from the data dictionary through a hash function within the database (the 'internal' approach), and use the database to store and analyze the resulting fingerprints.

Both approaches share two advantages over the standard audit/logging tools. First, they would identify only database objects that have changed, eliminating much of the analysis burden of the built in tools. Second, the processing can be scheduled for times when the database workload is light. The audit/logging features are busiest when workload is already high. A tripwire system wouldn't be subject to this limitation.

The external approach

There are at least three advantages to the external approach. First, use of the file system would leverage the existing tripwire technologies that are already available. It would avoid the need to duplicate this well-established functionality and support structure. Second, should a compromise be detected, the necessary source statements would be available for recovery. Third, the files are not readily apparent from the database itself. If the compromise is, in fact, limited to the database and not the host system, using the host system as the repository for the tripwire system would decrease the visibility of the database tripwire IDS to the intruder.

The external approach has its disadvantages, as well. If a traditional tripwire system is not already in use, this approach would require acquisition, installation, and maintenance of another application. In many environments, the operating system is the responsibility of a different individual or group than the database, so a DBA might not be able to make a unilateral decision to deploy such an application. Even if no organizational barriers exist, the DBA may prefer to support a database-centric solution, rather than one that requires operating system skills as well.

The internal approach

The biggest advantage of the internal approach is that it can be (almost) entirely self-contained. (The 'almost' can be eliminated on SQL Server 2005. The limitation for SQL Server 2000 will be discussed presently.) That means that the entire application can be implemented and operated using standard database tools. Instead of storing the object creation scripts in the filesystem, the code can be hashed immediately, and the hash can be stored in a table. There is no absolute requirement that the object creation scripts be stored (although the advantages to doing so, for recovery, still apply). Fingerprint comparison would be a very straightforward operation in an RDBMS.

This approach certainly minimizes the technical components and expertise that would be required – no operating system resources would be needed. Also, the data collected by a tripwire application is structured data, so an RDBMS is an efficient repository for the information. In fact, the commercial Tripwire product uses an RDBMS for its repository, too.

Perhaps the biggest advantage of the internal approach is that database security features can be applied to the tripwire system itself. The database can be used to provide another layer to the defense-in-depth strategy. The tripwire components developed for the internal solution will all be database objects, thus subject to database security mechanisms and available for inspection by all the standard database auditing features, and by the tripwire system itself.

The internal approach was selected to build the prototype for the database tripwire system.

Technical Challenges

Implementing a hash algorithm

SQL Server 2000 doesn't provide a built-in hash function of any type. Coding such a function natively in SQL Server T-SQL is non-trivial [15], and beyond the scope of this project. The focus of this project was not to investigate the implementation details of a hash function or the underlying algorithm, but to use it to solve a larger problem. In this context, the particular hashing algorithm and/or the implementation used can be easily replaced, should that be desirable or necessary.

The implementation details for the hash function are certainly relevant before the database tripwire application should be used in a production environment, but are decidedly peripheral for the purposes of this research. The criteria in this application were ease of use, and that it be based on a widely used algorithm, preferably SHA-1 or MD5.

Fingerprint generation from dictionary objects

Having a hash function, we need something to hash. All database objects are unique. Names differ (at least, within a SQL Server database, two objects of the same type must be uniquely named), structures differ, code modules are different, etc. This uniqueness is characterized by the statements used to create the objects, a subset of SQL known as Data Definition Language (DDL). If any attribute of the object changes, the DDL statement to create the object would also have to change. If we can pass the DDL code for the object through the hash function, we can generate a fingerprint for the object, which will necessarily change if the object is redefined in some way.

One of the classic techniques used by database professionals in a variety of situations is the development of SQL scripts to generate more SQL. Scripts could be written to query the data dictionary tables, formatting the output in such a way that the resulting report recreates the DDL for the object (which could actually be executed to re-create the object in the database).

This approach to generation of the DDL, while viable, is not a simple undertaking. Each type of object has a unique statement structure, generally with a wide range of variations and optional phrases. The generated code would have to be syntactically correct, and would have to allow for all the variations of a fairly rich language. Development of this kind of script would be a serious undertaking for even a single object type, such as a table or a stored procedure. Attempting to do so for the range of objects that would be of interest to a tripwire system would require a major development effort.

Fortunately, Microsoft provides this capability through the SQL Server Distributed Management Objects (DMO) (Merkin, 8]. DMO is used by both Enterprise Manager and Query Analyzer, which are the basic administrative tools provided with SQL Server. Query Analyzer, for example, is a GUI tool that provides both an Object Browser pane (which holds a treeview to navigate and inspect database objects), and a code pane (which can be used to edit and execute SQL, and view the results). By navigating to an object in the Browser and right-clicking, the DBA can instruct Query Analyzer to generate the object creation script (Fig. 1).

DMO is also available to T-SQL scripts via the sp_OA* procedures [11,16], and this is the utility that will be used to retrieve the object DML from the data dictionary and pass it to the hash function for fingerprinting.

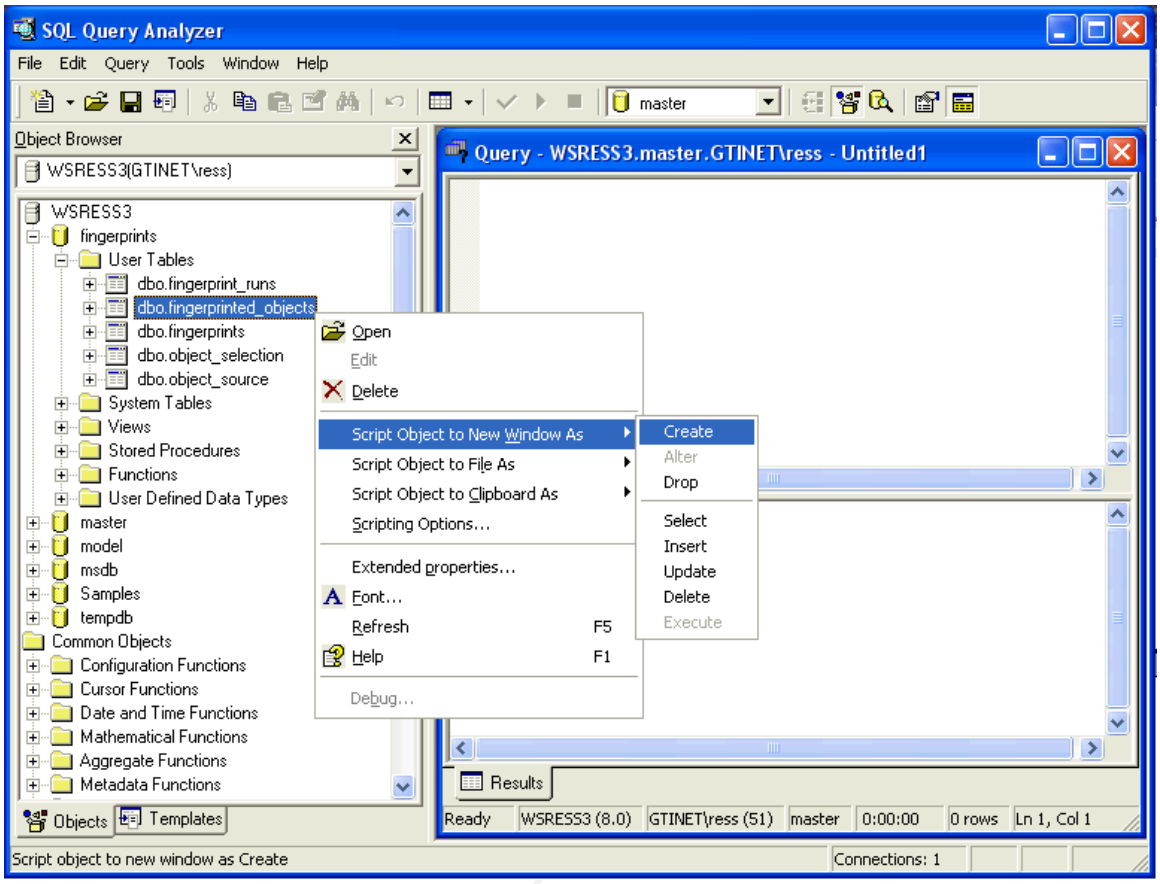

Fig. 1: Query Analyzer object scripting feature.

Even though DMO is available to handle many of the syntactic complexities of DDL statement creation, these routines need some specific information about which objects to generate. In general, it will be necessary to supply the SQL Server name, the database name, the object name, and – in the case of some objects – the parent table name.

It's not reasonable to burden the user of the system with explicit enumeration of every object that should be fingerprinted. In addition, we want the application itself to identify any new objects that might appear, and alert us of their arrival. After all, what use would a system designed to detect change be, if the user had to notify it in advance about new objects? The user should be able to specify, for example "Fingerprint all tables owned by a particular user in a specific database." The utility should take care of using the data dictionary to identify all objects that meet the criteria, and process them appropriately. If new objects appear that meet the criteria, they should be added.

Hash storage

The application should take advantage of the database for storage of fingerprints

and use the database to compare fingerprints from different runs to report any changes. One of the advantages of hashes is that they minimize the amount of storage required, and their brevity makes comparisons very efficient. There's nothing that would prohibit use of the DDL source code itself for comparison (Spurgeon [18] is an example of a source-comparison approach to a tripwire for a filesystem), but doing so would be much less efficient.

To allow for comparison, each fingerprint hash will have to be associated with the particular database object whose DDL was fingerprinted and the time it was fingerprinted (so there will be a later basis for comparison.)

It's to be expected that each fingerprint run will take some time to complete. Recording the exact time that each object was fingerprinted probably has little value. Recording a single time for each run (such as the start time) is probably sufficient. Further, when specifying the runs from which hashes should be compared, it's likely that the vast majority of the time the runs of interest will be the two most recent. So the notion of a 'run', rather than a 'runtime' will tend to have more value.

The object being fingerprinted, however, will be of particular interest. The system will need to uniquely identify each object fingerprinted, even if two similar objects have the same name (but different owners or databases). The application will have to be able to distinguish between the two, to allow accurate comparisons of fingerprints.

Note that hashes that don't match from one run to another are indications of one or more changes to the fingerprinted object. However, there's no way to tell, from the hash, what has changed. Hashes are designed to be one-way functions. In order to identify what has changed, it will be necessary to retain the source DDL as well, from at least the most recent pair of runs.

Reports

The application should provide one or more reports to identify the objects that have changed. At a minimum, the application will need to be able to identify the object that has changed (object name, database, owner, type of object). The first appearance of fingerprints for new objects and lack of fingerprints for previously existing objects will need to be detected as well.

Reports tend to be the most volatile component of any application. New requirements are identified, so the sort order is changed, or parameters are added to further constrain the volume of output, or additional information is added to a report, etc. Reports to document system configuration may be added.

At this stage, a single report should be developed for the tripwire application, to

report any changes in object fingerprints between two runs. It will have to provide enough information to identify the object(s) that changed. In the initial release, the source DDL will not be included, but that would probably be an early addition, either by incorporating the source in the master report, or as a separate report.

Implementation

Much of the code used in this project was based on the work of Mackey [10], Sampath [15, 16], and Khamal [8] cited in the references. Like so much of the application development in our profession, this project is built upon the work of others. It would not have been possible without colleagues willing to pass on their knowledge through publication. Hopefully, the same will be true of the code in this application as well.

The database tripwire system is logically divided into three components. The first is the build kit for the system (Fig. 3). This code module, if unmodified, will create a new database to act as the repository for most of the application and build the necessary data structures and program units needed for the system.

The second module consists of sample statements to create new selection criteria records and a statement to invoke a tripwire run to fingerprint the objects currently in the database (Fig. 4).

The final module contains the basic report that will compare the fingerprints from the last two tripwire runs to display information about the database objects that have changed in some way (Fig. 5).

Fingerprint Hash Function

The solution implemented was to acquire an MD5 hash function from a public source, coded as a Windows .dll, and call it via a SQL Server extended stored procedure (a SQL Server code module that calls an external routine). The source code for the .dll can be downloaded from Mackey [10]. In this case, the hash function was implemented with a C program distributed under the GNU General Public License. As noted in Sampath [15], other libraries are available for this purpose, including the Microsoft CryptoAPI for Windows 2000 and later versions of the operating system.

This is the one component of the current implementation that requires system-level activities, to store the .dll on the host system for the database. In a SQL Server 2005 environment, the inclusion of the .Net Common Language Runtime environment within the database will allow relatively easy access to the crypto services provided in the Framework Class Library. Such an alternative implementation will eliminate the need for a custom .dll or a call to some other library function.

Note that an extended stored procedure must be created in the Master database (one of the required databases in any SQL Server instance). Once the extended stored procedure to call the MD5 hash .dll is created in the build script (Fig. 3), the script will create a new database for the remaining objects that comprise the database tripwire application. The first object created will be a user function to call the extended stored procedure in the Master database.

Data Model

All tables in the implementation include identity columns. An identity is a generated primary key column for a table. The alternative approach is to attempt to use natural keys, which are a set of attributes (i.e. columns) that uniquely identify each record. There are advantages and disadvantages to both approaches, but in this case identity columns are used.

These identities will generally be ignored in the following discussion, but it should be pointed out that they are used to define most of the foreign key relationships in the application. They will, for example, be used to define the relationship between a fingerprint for an object, the object that was hashed, and the run during which that particular hash was generated.

Configuration

Three tables are used to store configuration information for the system. One (object_type_lookup) is a simple lookup table for the various object types, to provide a translation for the codes used for different object types in the data dictionary [1]. The script will populate this table with the necessary records. The supported object types at this point are system and user tables, views, procedures, and user functions. Other object type records have been included for future enhancements, and further entries beyond what are currently provided will almost certainly be needed.

The second table (selected_objects) is the repository for the DBA to record the object types that are to be fingerprinted. As can be seen in the data model for the system (Fig. 2), there are four attributes for each entry. These are 1) the server name, 2) the database name, 3) the object owner, and 4) the object type (recorded using the codes taken from the data dictionary). The sample maintenance scripts (Fig. 4) provide examples for creation of new entries.

The third table (fingerprinted_objects) is the list of names of the objects that results from applying the entries in the selected_objects table to the data dictionary. Objects with identical names are distinguished by storing the identity of the entry from the selected_objects table.

The fingerprinted_objects table is populated by a procedure run at the beginning of a fingerprint run (proc_populate_objects). It uses the entries created by the DBA in the selected_objects table to determine which dictionary objects currently match the selection criteria, and adds any objects to the fingerprinted_objects table that are not already there.

Note that objects are added to this table, but are typically never deleted. Even if the object itself is dropped from the database (and therefore from the data dictionary), fingerprints and source code records may still exist from fingerprint runs when the object was in existence.

Note also that a parent_id attribute, which is not currently used, has been included in the table to allow for future development. Some objects (e.g. indexes, constraints, triggers) are related to other objects (tables), and this relationship must be modelled to construct the DMO query to retrieve the DDL for these objects. Program stubs in the code for the build script (Fig. 3) have also been included, but commented out, to provide some idea how these objects will eventually be supported.

Fingerprint storage

The remaining three tables depicted in the data model make up the storage for the fingerprint runs themselves. The first table (fingerprint_runs) serves to record a timestamp for each run. As noted in the discussion of the problem, a fingerprint run is one dimension for the comparison of the fingerprint hashes (the object being the other). At the beginning of each fingerprint run, a new record will be added to the runs table, and all fingerprint and source code records created during the run will refer to this run timestamp record.

The second of these tables (fingerprints) is used to store the hash values for each object during the run. Each record will also store the identities for the appropriate records in the fingerprint_runs and fingerprinted_objects tables.

The final table, object_source, is identical to the fingerprints table, except that it stores the object source DDL in each record, instead of the fingerprint hash.

An alternative design that has merit would be to combine the fingerprint and source tables. There is a one-to-one correspondence between the records in the two tables, and the source column could simply be moved to the fingerprints table. The reason for implementing separate tables is to allow different amounts of history to be retained. The fingerprint records will be relatively small, and it may be desirable to retain them for longer periods of time. The source records will be much larger, and it may be desirable to purge them more often.

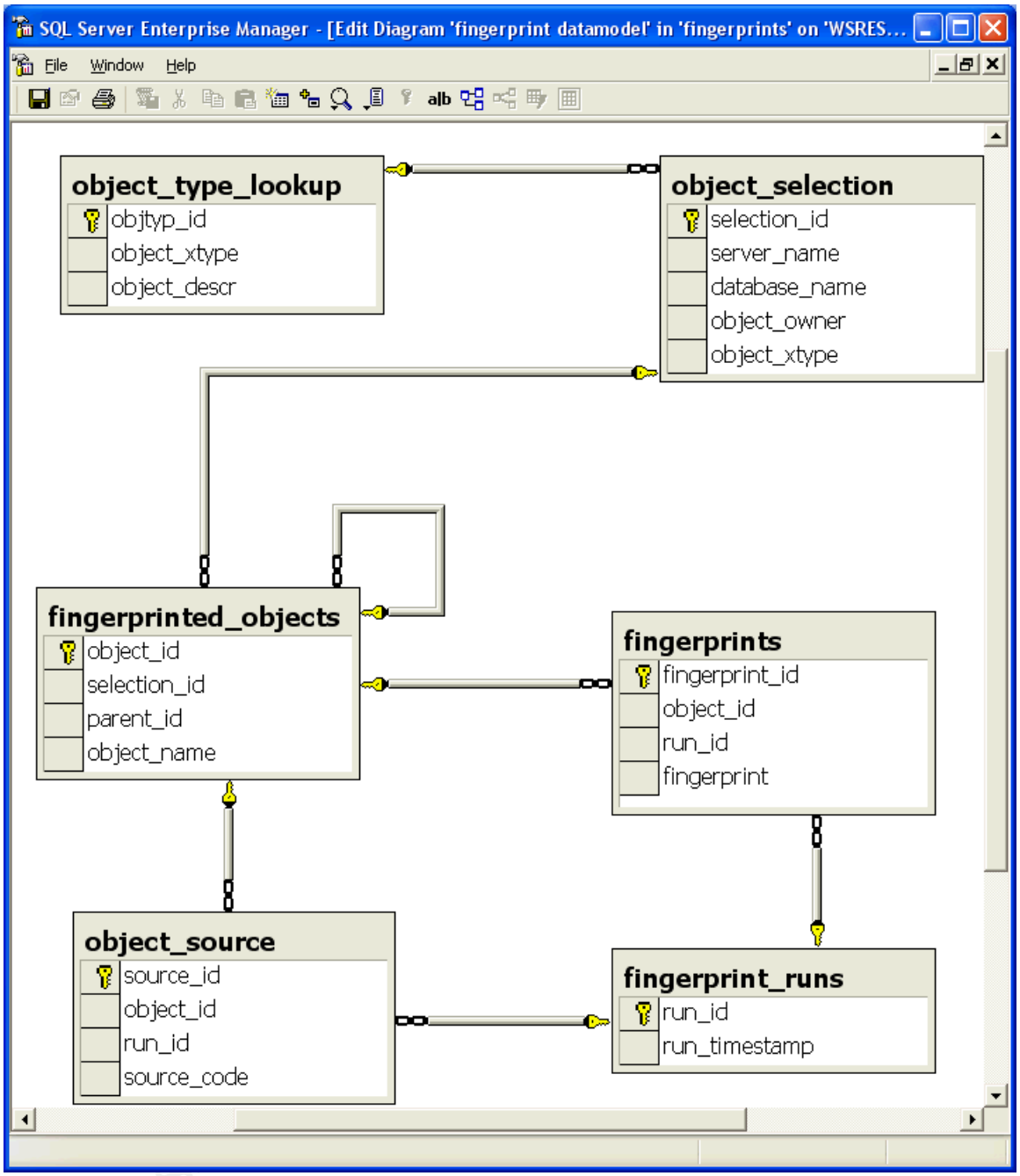

Fig. 2: Data model for database tripwire application.

```
--------------------------------------------------------------------------------------------
--   Object:      mk_sql_tripwire.sql   Script to build infrastructure
--                                      for SQL Server tripwire utility.
--   Script Date: 12/15/2004
--   Created By:  Frank Ress
--------------------------------------------------------------------------------------------

SET QUOTED_IDENTIFIER ON
GO
SET ANSI_NULLS OFF
GO

--
```

```
-- Create a new extended stored procedure that can be used to call the MD5 hash function
-- implemented in a .dll library in the filesystem.  Note that extended functions must
-- be created in the Master database (creation of such a function in another database
-- is not allowed).
--
-- In a SQL Server 2005 database, consideration should be given to using the Microsoft
-- crypto functionality available in the CLR, rather than calling an external function.
--

USE MASTER

EXEC sp_addextendedproc N'xp_md5',
                        N'C:\Program Files\Microsoft SQL Server\MSSQL\Binn\xp_md5.dll'
GO

--
-- Create a new database for the data structures and T-SQL routines used to implement
-- the tripwire system.  The database will be named 'fingerprints'.
--

CREATE DATABASE [fingerprints]
  ON (NAME = N'Hash_Data',
      FILENAME = N'C:\Program Files\Microsoft SQL Server\MSSQL\Data\Hash_Data.MDF',
      SIZE = 10MB,
      FILEGROWTH = 10%
     )
  LOG ON (NAME = N'Hash_Log',
          FILENAME = N'C:\Program Files\Microsoft SQL Server\MSSQL\Data\Hash_Log.LDF',
          SIZE = 10MB,
          FILEGROWTH = 10%
         )
GO

--
-- Create all subsequent objects in the new tripwire database.  This organization
-- is recommended in order to logically and physically isolate, as much as possible,
-- the IDS components from the remainder of the databases in the instance.
--

USE FINGERPRINTS
GO

CREATE FUNCTION fn_md5 (@string VARCHAR(8000))
RETURNS CHAR(32)
--
-- The fn_md5 function is passed a character string (presumably the SQL Source from the data
-- dictionary for a particular object).  It calls the md5 hash .dll through the xp_md5 external
-- stored procedure using the string as an argument, and returns the 32-bit hash for the
-- character string (i.e. the md5 fingerprint hash for the database object).
--
AS
BEGIN
  DECLARE @hash CHAR(32)
  EXEC master.dbo.xp_md5 @string, @hash OUTPUT
  RETURN @hash
END
GO

--
-- The object_type_lookup table is used to provide a list of SQL Server object types
-- (the column name is usually 'xtype') and corresponding descriptions.
--

CREATE TABLE object_type_lookup
             (objtyp_id SMALLINT
                        IDENTITY (1,1)
                        PRIMARY KEY CLUSTERED,
              object_xtype CHAR(2) COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL UNIQUE,
              object_descr VARCHAR(40) NOT NULL
             )
GO

INSERT INTO object_type_lookup (object_xtype, object_descr)
                        VALUES ('C', 'CHECK Constraint')
INSERT INTO object_type_lookup (object_xtype, object_descr)
                        VALUES ('D', 'Default or DEFAULT Constraint')
INSERT INTO object_type_lookup (object_xtype, object_descr)
                        VALUES ('F', 'FOREIGN KEY Constraint')
INSERT INTO object_type_lookup (object_xtype, object_descr)
```

```
                        VALUES ('FN', 'User Defined Function')
INSERT INTO object_type_lookup (object_xtype, object_descr)
                        VALUES ('L', 'Log')
INSERT INTO object_type_lookup (object_xtype, object_descr)
                        VALUES ('P', 'Stored Procedure')
INSERT INTO object_type_lookup (object_xtype, object_descr)
                        VALUES ('PK', 'Primary Key Constraint')
INSERT INTO object_type_lookup (object_xtype, object_descr)
                        VALUES ('RF', 'Replication Filter Stored Procedure')
INSERT INTO object_type_lookup (object_xtype, object_descr)
                        VALUES ('S', 'System Table')
INSERT INTO object_type_lookup (object_xtype, object_descr)
                        VALUES ('TR', 'Trigger')
INSERT INTO object_type_lookup (object_xtype, object_descr)
                        VALUES ('U', 'User Table')
INSERT INTO object_type_lookup (object_xtype, object_descr)
                        VALUES ('UQ', 'UNIQUE Constraint')
INSERT INTO object_type_lookup (object_xtype, object_descr)
                        VALUES ('V', 'View')
INSERT INTO object_type_lookup (object_xtype, object_descr)
                        VALUES ('X', 'Extended Stored Procedure')
GO

--
-- The object_selection table is used by the DBA to specify which database objects should
-- be fingerprinted during a tripwire run.  The current version of this utility does not
-- support explicit inclusion or exclusion of individual objects.  Rather, the DBA is
-- allowed to specify the server/database/owner/object types to be included, and all data
-- dictionary objects that match any of the entries in the table will be included.
--

CREATE TABLE object_selection
             (selection_id SMALLINT
                           IDENTITY (1,1)
                           PRIMARY KEY CLUSTERED,
              server_name SYSNAME NOT NULL,
              database_name SYSNAME NOT NULL,
              object_owner SYSNAME NOT NULL,
              object_xtype CHAR(2) COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL
                           REFERENCES object_type_lookup (object_xtype),
             )
GO

--
-- The fingerprinted_objects table is used to store information about the objects that
-- are being monitored for potential changes.  It is NOT intended to be maintained by
-- the DBA directly, it will be populated automatically by the proc_populate_objects
-- procedure, based on the selection criteria entered by the DBA in the object_selection
-- table.
--
-- Records will be added to this table over time, as new objects are added to the
-- database(s).  Consequently, the proc_populate_objects procedure is designed to
-- first check that an record for an object is not already present.
--

CREATE TABLE fingerprinted_objects
             (object_id SMALLINT
                        IDENTITY (1,1)
                        PRIMARY KEY CLUSTERED,
              selection_id SMALLINT NOT NULL
                        REFERENCES object_selection (selection_id),
              parent_id SMALLINT
                        REFERENCES fingerprinted_objects (object_id),
              object_name SYSNAME NOT NULL
             )
GO

--
-- The fingerprint_runs table is used to create a record for a timestamp for a fingerprint
-- run.  One record is created for each run of the proc_fingerprint procedure.
--
-- These records supply the analysis reports with the timestamps of runs, so it's possible
-- to determine which versions of hashes can be compared.  It also allows object hashes to
-- be associated with a particular copy of the object source code, should it be necessary
-- to analyze the details of a change in an object.
--

CREATE TABLE fingerprint_runs
             (run_id SMALLINT
```

```
                    IDENTITY (1,1)
                    PRIMARY KEY CLUSTERED,
             run_timestamp TIMESTAMP
            )
GO

--
-- The fingerprints table is used to record the hash of each object for each run of the
-- proc_fingerprint procedure.  A fingerprint hash will be generated for each object in the
-- fingerprinted_objects table (if the object still exists).
--

CREATE TABLE fingerprints
            (fingerprint_id SMALLINT
                            IDENTITY (1,1)
                            PRIMARY KEY CLUSTERED,
             object_id SMALLINT NOT NULL
                       REFERENCES fingerprinted_objects (object_id),
             run_id SMALLINT NOT NULL
                    REFERENCES fingerprint_runs (run_id),
             fingerprint CHAR(32) NOT NULL
            )
GO

--
-- The object_source table is used to record the source code (typically some kind of CREATE
-- statement) for each object fingerprinted in each run of the proc_fingerprint procedure.
-- The object source from different runs of the fingerprinting procedure can then be compared
-- if hashes don't match from one run to another.
--
-- Because of the storage required, there's a high probability that an administrator will
-- elect to retain fewer versions of the object source than copies of the fingerprint hashes.
--

CREATE TABLE object_source
            (source_id SMALLINT
                       IDENTITY (1,1)
                       PRIMARY KEY CLUSTERED,
             object_id SMALLINT NOT NULL
                       REFERENCES fingerprinted_objects (object_id),
             run_id SMALLINT NOT NULL
                    REFERENCES fingerprint_runs (run_id),
             source_code VARCHAR(8000) NOT NULL
            )
GO

CREATE PROCEDURE sp_hexadecimal
--
-- Hexadecimal conversion utility taken directly from sample in SQL Server Books Online
--
    @binvalue varbinary(255),
    @hexvalue varchar(255) OUTPUT
AS
DECLARE @charvalue varchar(255)
DECLARE @i int
DECLARE @length int
DECLARE @hexstring char(16)
SELECT @charvalue = '0x'
SELECT @i = 1
SELECT @length = DATALENGTH(@binvalue)
SELECT @hexstring = '0123456789abcdef'
WHILE (@i <= @length)
BEGIN
DECLARE @tempint int
DECLARE @firstint int
DECLARE @secondint int
SELECT @tempint = CONVERT(int, SUBSTRING(@binvalue,@i,1))
SELECT @firstint = FLOOR(@tempint/16)
SELECT @secondint = @tempint - (@firstint*16)
SELECT @charvalue = @charvalue +
SUBSTRING(@hexstring, @firstint+1, 1) +
SUBSTRING(@hexstring, @secondint+1, 1)
SELECT @i = @i + 1
END
SELECT @hexvalue = @charvalue
GO

CREATE PROCEDURE sp_displayoaerrorinfo
--
```

```
-- Error translation utility taken directly from sample in SQL Server Books Online
--
    @object int,
    @hresult int
AS
DECLARE @output varchar(255)
DECLARE @hrhex char(10)
DECLARE @hr int
DECLARE @source varchar(255)
DECLARE @description varchar(255)
PRINT 'OLE Automation Error Information'
EXEC sp_hexadecimal @hresult, @hrhex OUT
SELECT @output = '  HRESULT: ' + @hrhex
PRINT @output
EXEC @hr = sp_OAGetErrorInfo @object, @source OUT, @description OUT
IF @hr = 0
BEGIN
SELECT @output = '  Source: ' + @source
PRINT @output
SELECT @output = '  Description: ' + @description
PRINT @output
END
ELSE
BEGIN
    PRINT ' sp_OAGetErrorInfo failed.'
    RETURN
END
GO

CREATE PROCEDURE proc_populate_objects
--
-- The proc_populate_objects procedure processes the records created by the DBA in the
-- object_selection table, which define what target object records are needed in the
-- fingerprinted_objects table.  The name of each object is retrieved from the sysobjects
-- system table.  The sysobjects table is specific to each SQL Server database, so the
-- routine will establish a connection to each distinct server found in the object selection
-- table, in turn.  For each database on the server that has objects the DBA wishes included, a
-- single INSERT statment will be constructed and executed to add them to the object table.
-- Since the objects may already have been included in a previous run, the statement includes
-- a check to prevent duplicate records.
--
AS

DECLARE @servername SYSNAME
DECLARE @cmdstr VARCHAR(255)
DECLARE @object INT
DECLARE @resultcode INT

DECLARE server_cursor CURSOR FOR
SELECT DISTINCT server_name
  FROM object_selection
 ORDER BY server_name

OPEN server_cursor
FETCH NEXT FROM server_cursor INTO @servername

-- While there are still servers with objects to process, loop through each
-- database in turn, adding objects to the fingerprinted_objects table.

WHILE @@FETCH_STATUS = 0
BEGIN

   -- Establish a connection to the server

   SET @cmdstr = 'Connect('+@servername+')'
   EXEC @resultcode = sp_OACreate 'SQLDMO.SQLServer', @object OUT

   -- Uncomment for integrated login
   EXEC @resultcode = sp_OASetProperty @object, 'LoginSecure', TRUE

   -- Uncomment for standard login
   --EXEC @resultcode = sp_OASetProperty @object, 'Login', 'sa'
   --EXEC @resultcode = sp_OASetProperty @object, 'password', 'sapassword'

   EXEC @resultcode = sp_OAMethod @object, @cmdstr
   EXEC @resultcode = sp_OADestroy @object

   DECLARE @databasename SYSNAME
```

```
   DECLARE database_cursor CURSOR FOR
   SELECT DISTINCT database_name
     FROM object_selection
    WHERE server_name = @servername
    ORDER BY database_name

   OPEN database_cursor
   FETCH NEXT FROM database_cursor INTO @databasename

   WHILE @@FETCH_STATUS = 0
   BEGIN
      DECLARE @sqlstring NVARCHAR(1000)
      SELECT @sqlstring =
      N'INSERT INTO fingerprints.dbo.fingerprinted_objects (selection_id,
                                                            object_name)
      SELECT objsel.selection_id,
             syso.name
        FROM ' + @databasename + '.dbo.sysobjects syso,
             object_selection objsel
       WHERE objsel.server_name = ' + '''' + @servername + '''' + '
         AND objsel.database_name = ' + '''' + @databasename + '''' + '
         AND objsel.object_owner = user_name (syso.uid)
         AND objsel.object_xtype = syso.xtype
         AND CONVERT (CHAR(6), objsel.selection_id)
                + syso.name NOT IN (SELECT CONVERT (CHAR(6), selection_id)
                                           + object_name
                                    FROM fingerprints.dbo.fingerprinted_objects)'
      EXEC (@sqlstring)
      FETCH NEXT FROM database_cursor INTO @databasename
   END

   CLOSE database_cursor
   DEALLOCATE database_cursor

   FETCH NEXT FROM server_cursor INTO @servername

END

CLOSE server_cursor
DEALLOCATE server_cursor
GO

CREATE PROCEDURE proc_fingerprint
--
-- The proc_fingerprint procedure is used to create a new set of fingerprint hashes for the
-- objects in the fingerprinted_objects table.  It first adds a new record to the fingerprint_runs
-- table to establish the timestamp for the run.  Then it adds a new record in the
-- object_fingerprints and object_source table for each object in the fingerprinted_objects
-- table.
--
AS

DECLARE @runid INT
DECLARE @cmdstr VARCHAR(1000)
DECLARE @object INT
DECLARE @resultcode INT
DECLARE @objdatabase INT
DECLARE @servername SYSNAME
DECLARE @databasename SYSNAME
DECLARE @objid SMALLINT
DECLARE @objname SYSNAME
DECLARE @objowner SYSNAME
DECLARE @tablename SYSNAME
DECLARE @objxtype CHAR(2)
DECLARE @cmd VARCHAR(300)
DECLARE @objsource VARCHAR(8000)
DECLARE @temp CHAR(20)

DECLARE server_cursor CURSOR FOR
SELECT DISTINCT server_name
  FROM object_selection
 ORDER BY server_name

      DECLARE @message SYSNAME
--      SELECT @message = @servername + @databasename
--      PRINT @message

INSERT INTO fingerprint_runs DEFAULT VALUES
SELECT @runid = MAX (run_id) FROM fingerprint_runs
```

```
OPEN server_cursor
FETCH NEXT FROM server_cursor INTO @servername

-- While there are still servers with objects to process, loop through each
-- database in turn, adding objects to the fingerprinted_objects table.

WHILE @@FETCH_STATUS = 0
BEGIN

   -- Establish a connection to the server

   SET @cmdstr = 'Connect('+@servername+')'
   EXEC @resultcode = sp_OACreate 'SQLDMO.SQLServer', @object OUT

   -- Uncomment for integrated login
   EXEC @resultcode = sp_OASetProperty @object, 'LoginSecure', TRUE

   -- Uncomment for standard login
   --EXEC @resultcode = sp_OASetProperty @object, 'Login', 'sa'
   --EXEC @resultcode = sp_OASetProperty @object, 'password', 'sapassword'

   EXEC @resultcode = sp_OAMethod @object, @cmdstr

   DECLARE database_cursor CURSOR FOR
   SELECT DISTINCT database_name
     FROM object_selection
    WHERE server_name = @servername
    ORDER BY database_name

   OPEN database_cursor
   FETCH NEXT FROM database_cursor INTO @databasename

   -- While there are still databases hosted on the current server with objects to
   -- process, loop through each object in turn, adding fingerprint and source records
   -- to the fingerprints and fingerprint_source tables.

   WHILE @@FETCH_STATUS = 0
   BEGIN

      DECLARE object_cursor CURSOR FOR
      SELECT object_id,
             object_name,
             object_owner,
             object_xtype
        FROM fingerprinted_objects fingobj,
             object_selection objsel
       WHERE fingobj.selection_id = objsel.selection_id
         AND server_name = @servername
         AND database_name = @databasename
       ORDER BY object_id

      OPEN object_cursor
      FETCH NEXT FROM object_cursor INTO @objid,
                                         @objname,
                                         @objowner,
                                         @objxtype

      WHILE @@FETCH_STATUS = 0
      BEGIN

-- Cases that are commented out in the following statement have not been fully implemented.
-- They represent a starting point for work to be done.  Also, the arguments in mixed case
-- (e.g. 'Index') represent object types that are NOT found in the sysobjects table.  Some
-- 'pseudo' xtype will probably be needed to include such objects in the application.
         SET @cmdstr =
           CASE @objxtype
--           WHEN 'Database'
--              THEN 'Databases("'
             WHEN 'FN'
                THEN 'Databases("' + @databasename + '").UserDefinedFunctions("' + @objowner + '.'
             WHEN 'P'
                THEN 'Databases("' + @databasename + '").StoredProcedures("' + @objowner + '.'
             WHEN 'V'
                THEN 'Databases("' + @databasename + '").Views("' + @objowner + '.'
             WHEN 'U'
                THEN 'Databases("' + @databasename + '").Tables("' + @objowner + '.'
             WHEN 'S'
                THEN 'Databases("' + @databasename + '").Tables("' + @objowner + '.'
```

```
--           WHEN 'Index'
--              THEN
'Databases("'+@databasename+'").Tables("'+@objowner+'.'+@TableName+'").Indexes("'
--           WHEN 'TR'
--             THEN
'Databases("'+@databasename+'").Tables("'+@objowner+'.'+@TableName+'").Triggers("'
--           WHEN 'PK'
--              THEN 'Databases("'+@databasename+'").Tables("'+@objowner+'.'+@TableName+'").Keys("'
--           WHEN 'UQ'
--              THEN 'Databases("'+@databasename+'").Tables("'+@objowner+'.'+@TableName+'").Keys("'
--           WHEN 'C'
--              THEN
'Databases("'+@databasename+'").Tables("'+@objowner+'.'+@TableName+'").Checks("'
--           WHEN 'Job'
--              THEN 'Jobserver.Jobs("' + @objowner + '.'
          END

         SET @cmdstr = @cmdstr + @objname + '").Script'
         EXEC @resultcode = sp_OAMethod @object, @cmdstr, @objsource OUTPUT, 4
         IF @resultcode != 0
         -- If the resultcode is non-zero, an error occurred - display diagnostic info.
         BEGIN
            PRINT @cmdstr
            EXEC @temp = sp_displayoaerrorinfo @object, @resultcode
         END
         ELSE
         BEGIN
            INSERT INTO fingerprints (object_id,
                                      run_id,
                                      fingerprint)
                              VALUES (@objid,
                                      @runid,
                                      fingerprints.dbo.fn_md5(@objsource))

            INSERT INTO object_source (object_id,
                                       run_id,
                                       source_code)
                               VALUES (@objid,
                                       @runid,
                                       @objsource)
         END

         FETCH NEXT FROM object_cursor INTO @objid,
                                            @objname,
                                            @objowner,
                                            @objxtype
      END

      CLOSE object_cursor
      DEALLOCATE object_cursor

      FETCH NEXT FROM database_cursor INTO @databasename

   END

   CLOSE database_cursor
   DEALLOCATE database_cursor

   FETCH NEXT FROM server_cursor INTO @servername

END

CLOSE server_cursor
DEALLOCATE server_cursor
EXEC @resultcode = sp_OADestroy @object
GO

SET QUOTED_IDENTIFIER ON
GO
SET ANSI_NULLS ON
GO
```

Fig. 3: Build script for database tripwire application.

Fingerprint generation

As noted in the discussion of the data model, one procedure adds new objects to the list to be fingerprinted, based on the selection criteria defined by the DBA (proc_populate_objects). A second procedure (proc_fingerprint) actually generates and stores the fingerprint hashes for every object listed in the fingerprinted_objects table – if the object still exists.

At this time, these routines are independent. It allows the DBA to process the selection criteria and check the results in the selected_objects table before beginning a fingerprint run. In a production deployment, it would be preferable to call the proc_populate_objects routine at the start of the proc_fingerprint procedure. As the tool is still in development, this integration has been deferred.

When proc_fingerprint is executed, it first creates a new record in fingerprint runs. It then retrieves records from fingerprinted_objects, along with information joined from the selected_objects table, adding fingerprint and source records to the fingerprints and object_source tables, respectively, for all objects that can be found in the data dictionary.

For both procedures (proc_populate_objects and proc_fingerprint) the records are sorted first by server. The design for the application supports monitoring multiple servers, so a connection is established for a server at a time. All records for one server are then processed before dropping that connection and moving to the next. When proc_populate_objects is integrated with proc_fingerprint, it should be done in such a way that each server connection is opened only once per run.

Objects currently processed

The current version of the application is able to process system and user tables, views, procedures, and user functions.

During testing, one of the procedures (proc_fingerprint) consistently failed to generate a hash. To facilitate debugging, the sp_hexadecimal and sp_displayoaerrorinfo procedures were copied from Books Online [13] and added to the code base. Further research identified a SQL Server error, Bug #356574 [12]. Microsoft reports this bug should be fixed with SQL Server SP3 and SP3a. However, development was performed on an MSDE SP3a database running under Windows XP Professional, SP2. Further research is ongoing to determine the reason for the persistence of this error.

The routines added for debugging proved invaluable, and should probably be retained and used more extensively going forward.

Candidate objects for inclusion

The next objects that will be added to the application are the ones that require a table reference, such as indexes, constraints, triggers, etc. The additional complexity of these relationships will cause a similar increase in the complexity of the structure of the conditional logic in the code. The structure of the parameters that must be passed to the sp_OAMethod routine is already known, so the work required to add these additional objects is relatively straightforward.

There are also objects for which none of the literature or web searches to date has identified any sample code or algorithms. In particular, more extensive research will be needed to determine how to generate the DDL for account creation, role creation, privilege assignment, etc. It may be necessary to backtrack through the SQL Server metadata views to identify the necessary dictionary items to query [7]. In terms of value, these objects are of relatively high priority, since they address database security features, which would be worth monitoring, to say the least. In terms of the work required to include these objects in the application, however, less is known, and their priority is lower.

Sample runs

Sample selection criteria entry and fingerprint execution commands using Query Analyzer are displayed in Fig. 4.

First, we create a couple of scratch tables and then select a few object types that we want to fingerprint (**`STEP A)`**. The scratch tables are included. We query the object_selection table to show the entries we just added (**`STEP B)`**, then we execute the proc_populate_objects to create our list of tripwired objects in the selected_objects table (**`STEP C)`**. The contents of that are also displayed.

Next, the proc_fingerprint is executed to generate fingerprints for all the objects. The contents of the fingerprints table are displayed (**`STEP D)`**. Note that the fingerprint for object 24 is missing (due to the bug discussed previously). It's also worth noting that an entry was created in the fingerprint_runs table, and that the source for each object (except 24) was also recorded in the object_source table.

Next, we change the structure of one of our sample tables, drop another one, and add a new one. Then run the proc_populate_objects and proc_fingerprint routines again (**`STEP E)`**. Once again, we'll display some of the records after each step, but we'll screen out all but the sample tables.

Note, in the fingerprinted_objects table, that all three sample tables are displayed after the second run of proc_populate_objects. Test2 was dropped and test3 was added, but all 3 objects remain. Then, after proc_fingerprint is run, hashes are generated for test1 and test3, but test2 is missing, since the

object no longer exists. Note also that the hash for test1 has changed (a new column was added), and that the hashes for objects 32 and 40 are identical to one another and from run to run (these objects are identical system tables in different databases).

STEP A

```
CREATE TABLE test1 (charcolumn CHAR(1))
CREATE TABLE test2 (intcolumn INT)

INSERT INTO object_selection (server_name, database_name, object_owner, object_xtype)
                     VALUES ('WSRESS3', 'FINGERPRINTS', 'DBO', 'P')

INSERT INTO object_selection (server_name, database_name, object_owner, object_xtype)
                     VALUES ('WSRESS3', 'FINGERPRINTS', 'DBO', 'FN')

INSERT INTO object_selection (server_name, database_name, object_owner, object_xtype)
                     VALUES ('WSRESS3', 'FINGERPRINTS', 'DBO', 'S')

INSERT INTO object_selection (server_name, database_name, object_owner, object_xtype)
                     VALUES ('WSRESS3', 'FINGERPRINTS', 'DBO', 'U')

INSERT INTO object_selection (server_name, database_name, object_owner, object_xtype)
                     VALUES ('WSRESS3', 'FINGERPRINTS', 'DBO', 'V')

INSERT INTO object_selection (server_name, database_name, object_owner, object_xtype)
                     VALUES ('WSRESS3', 'MASTER', 'DBO', 'V')
```

STEP B

```
SELECT * FROM object_selection
```

selection_id	server_name	database_name	object_owner	object_xtype
1	WSRESS3	FINGERPRINTS	DBO	P
2	WSRESS3	FINGERPRINTS	DBO	FN
3	WSRESS3	FINGERPRINTS	DBO	S
4	WSRESS3	FINGERPRINTS	DBO	U
5	WSRESS3	FINGERPRINTS	DBO	V
6	WSRESS3	MASTER	DBO	V

STEP C

```
EXEC proc_populate_objects

SELECT * FROM fingerprinted_objects
 ORDER BY selection_id
```

object_id	selection_id	parent_id	object_name
21	1	NULL	sp_hexadecimal
22	1	NULL	sp_displayoaerrorinfo
23	1	NULL	proc_populate_objects
24	1	NULL	proc_fingerprint
33	2	NULL	fn_md5
1	3	NULL	sysobjects
2	3	NULL	sysindexes
3	3	NULL	syscolumns
4	3	NULL	systypes
5	3	NULL	syscomments
6	3	NULL	sysfiles1
7	3	NULL	syspermissions
8	3	NULL	sysusers
9	3	NULL	sysproperties

```
10              3               NULL            sysdepends
11              3               NULL            sysreferences
12              3               NULL            sysfulltextcatalogs
13              3               NULL            sysindexkeys
14              3               NULL            sysforeignkeys
15              3               NULL            sysmembers
16              3               NULL            sysprotects
17              3               NULL            sysfulltextnotify
18              3               NULL            sysfiles
19              3               NULL            sysfilegroups
20              4               NULL            fingerprint_runs
34              4               NULL            object_type_lookup
25              4               NULL            test1
26              4               NULL            test2
27              4               NULL            object_selection
28              4               NULL            fingerprinted_objects
29              4               NULL            fingerprints
30              4               NULL            object_source
31              5               NULL            syssegments
32              5               NULL            sysconstraints
35              6               NULL            syslogins
36              6               NULL            sysremotelogins
37              6               NULL            sysoledbusers
38              6               NULL            sysopentapes
39              6               NULL            syssegments
40              6               NULL            sysconstraints
```

STEP D

```
EXEC proc_fingerprint

SELECT * FROM fingerprints

fingerprint_id object_id      run_id  fingerprint
-------------- ---------      ------  -----------
1              1              1       94527fe706d40d5955129315d6a8bf7f
2              2              1       66be860e00cdc3b8ec91caf40af02fb0
3              3              1       2e96d26241cdbbf5046736ad2c33e347
4              4              1       c22200af6ba5270f311eb3c121949e25
5              5              1       b3e22a9626c84400bdbce11fc3d11d6e
6              6              1       866f530646758217cf751a2bf63535c5
7              7              1       f550a04f45db0ba05ac69e872a0e1171
8              8              1       ca64039b8558a48ac9d2188b8bbf30a5
9              9              1       2fd6faed877d44205a7950f01703798d
10             10             1       a844bdeac030cdf89fc2d4362b8c8682
11             11             1       8e36358349bb21f9d30b0bfcf7ce1843
12             12             1       600eec84c67d3babc9503a4f8d025d88
13             13             1       a5947c3053c71c48772cf961bdf0f8a5
14             14             1       d64a2ef74c39e0e12768bfa89bcfa03d
15             15             1       a64b1ddcaa5a31d36a54de043f84bad5
16             16             1       1bec55a014c3e9eb5d03b3f88619145f
17             17             1       5114c7ddd035700c708972eed1344fdd
18             18             1       9a44b83e53ed7104b9c31907387b6957
19             19             1       9199755976e12d364ae28db7ce260c99
20             20             1       162a33a0ad854c7339d2485b97a73652
21             21             1       8400e1354b031791cd09a8e758d82443
22             22             1       2039a90db1d22452b1151be1d62e4e69
23             23             1       1c9b19c5bd4c37a47d22b8ff78412c73
24             25             1       57a8f08291f12e1b2f8515f8e3cd6c6a
25             26             1       432d788d61f9e6e15c55bb6c1bc6208e
26             27             1       c642c8a43371518f3ef1b1a7c7dc7a02
27             28             1       d23961c71851771ecb72f221d3fd8c41
28             29             1       9974ea65ec5037b9025658d8c31b61ea
29             30             1       f4be5f8863627450fb76da9e17abc0a5
30             31             1       af121dbcdb19be8d82906e466697e6ad
31             32             1       0d009950a2d9c825a84853be59fd2945
32             33             1       1276674f78049c01e777d9055ad6a44f
```

```
33              34              1       cc489f65b4b30d274fce336e12c84f23
34              35              1       665373b2ba767a3744740f21a592d86f
35              36              1       a0af6710219d86a929058e1584fc8d28
36              37              1       ea8bd89c6b853fbbeb061f6fda8c92c9
37              38              1       a7aa82c276fa2f76bafafb42af87c857
38              39              1       af121dbcdb19be8d82906e466697e6ad
39              40              1       0d009950a2d9c825a84853be59fd2945

STEP E

ALTER TABLE test1 ADD intcolumn INT
DROP TABLE test2
CREATE TABLE test3 (intcolumn INT)

EXEC proc_populate_objects

SELECT * FROM fingerprinted_objects
 WHERE object_name like 'TEST%'
 ORDER BY selection_id

object_id       selection_id    parent_id       object_name
---------       ------------    ---------       -----------
25              4               NULL            test1
26              4               NULL            test2
41              4               NULL            test3

EXEC proc_fingerprint

SELECT * FROM fingerprints
 WHERE object_id IN (25, 26, 32, 40, 41)
 ORDER BY object_id, run_id

fingerprint_id object_id       run_id  fingerprint
-------------- ---------       ------  -----------
24              25              1       57a8f08291f12e1b2f8515f8e3cd6c6a
63              25              2       cacdea3c5078a1c28aa7d85e85d96166
25              26              1       432d788d61f9e6e15c55bb6c1bc6208e
31              32              1       0d009950a2d9c825a84853be59fd2945
69              32              2       0d009950a2d9c825a84853be59fd2945
39              40              1       0d009950a2d9c825a84853be59fd2945
78              40              2       0d009950a2d9c825a84853be59fd2945
72              41              2       de210fad668769caeb73e30e28f1e095
```

Fig. 4: Sample tripwire execution in Query Analyzer.

Reports

Fig. 5 shows a basic report that will compare the fingerprints from the two most recent runs of the proc_fingerprint routine and the output generated when it was run against the samples created in the previous section of this report. It describes the type of activity that caused the record to be created (either a change in the fingerprint hash from one run to the next, or the addition or deletion of the object), as well as the basic information about the object that will allow the DBA to identify the object affected. At that point, presumably, the DBA will either be able to account for the activity, or she will investigate further to determine the reason for the activity.

Obviously, there are numerous opportunities to improve on this basic report. It

could include the run dates being compared; it could include the source code from the objects in question, etc. It might be useful to produce separate reports for each server, or database. Nevertheless, it serves to illustrate how quickly such a tool could become essential in almost any database environment. Out of the 40-odd objects we were monitoring, we are immediately aware of the ones that we need to check up on.

```
SELECT 'New object    ' Activity,
       server_name,
       database_name,
       object_owner,
       object_name,
       object_descr
  FROM fingerprints currun,
       object_selection objsel,
       fingerprinted_objects finobj,
       object_type_lookup objlook
 WHERE currun.object_id = finobj.object_id
   AND finobj.selection_id = objsel.selection_id
   AND objsel.object_xtype = objlook.object_xtype
   AND currun.run_id = (SELECT MAX(run_id) from fingerprint_runs)
   AND currun.object_id NOT IN (SELECT object_id FROM fingerprints
                                 WHERE run_id = (SELECT MAX(run_id)-1 from
fingerprint_runs)
                               )
UNION
SELECT 'Changed object  ' Activity,
       server_name,
       database_name,
       object_owner,
       object_name,
       object_descr
  FROM fingerprints currun,
       fingerprints lastrun,
       object_selection objsel,
       fingerprinted_objects finobj,
       object_type_lookup objlook
 WHERE currun.object_id = lastrun.object_id
   AND currun.object_id = finobj.object_id
   AND currun.fingerprint != lastrun.fingerprint
   AND finobj.selection_id = objsel.selection_id
   AND objsel.object_xtype = objlook.object_xtype
   AND currun.run_id = (SELECT MAX(run_id) from fingerprint_runs)
   AND lastrun.run_id = (SELECT MAX(run_id)-1 from fingerprint_runs)
UNION
SELECT 'Dropped object  ' Activity,
       server_name,
       database_name,
       object_owner,
       object_name,
       object_descr
  FROM fingerprints lastrun,
       object_selection objsel,
       fingerprinted_objects finobj,
       object_type_lookup objlook
 WHERE lastrun.object_id = finobj.object_id
   AND finobj.selection_id = objsel.selection_id
   AND objsel.object_xtype = objlook.object_xtype
   AND lastrun.run_id = (SELECT MAX(run_id)-1 from fingerprint_runs)
   AND lastrun.object_id NOT IN (SELECT object_id FROM fingerprints
                                  WHERE run_id = (SELECT MAX(run_id) from
fingerprint_runs)
                                )
```

```
 ORDER BY 1, 2, 3, 4, 5

Activity       server_name    database_name  object_owner   object_name    object_descr
--------       -----------    -------------  ------------   -----------    ------------
Changed object WSRESS3        FINGERPRINTS   DBO            test1          User Table
Dropped object WSRESS3        FINGERPRINTS   DBO            test2          User Table
New object     WSRESS3        FINGERPRINTS   DBO            test3          User Table
```

Fig. 5: Sample report and output in Query Analyzer.

Securing the tripwire Application

The database tripwire application, in its present form, is **NOT** a production application. During development and testing there are enough 'challenges' to be met, and application security is not yet a priority. The following discussion is not a description of what has been done, rather it's a description of what needs to be done.

First and foremost, this is a database application, much like any other, and the principle of least privilege applies. The application should be strenuously tested to determine the minimum rights necessary for it to function. A database account should be dedicated to the tripwire application, to allow the privileges of the application to be tuned without interference from the needs of other applications that would otherwise share the account. Initially, the account should have minimal rights, and additional rights and privileges should be added only as it's proven during testing that they're absolutely necessary. Included in this analysis should be consideration what rights could be granted and later revoked (such as elevated privileges for installation of the application, like the right to create tables and procedures, which are not needed later, during operations).

It's also imperative to make sure that the components of the tripwire application (tables, code units, etc.) are secure. As much distance – logical and physical - between the tripwire components and the monitored components as can be created is probably best. Use separate database accounts – tables should be owned by a tripwire account, rather than anyone else. Create separate database for the application (which is why the script builds a new database). Use an entirely separate server for the tripwire database. In any case, insure that only authorized access to the tripwire components is tolerated. Don't rely on security through obscurity – it doesn't hurt, and in some cases it's even helped [19], but it's not enough. Someone will find your application, and corrupt it if they can.

Imported code, such as the xp_md5.dll file, should be treated with suspicion. The source code should be inspected to determine exactly how it operates, then it should be compiled locally to be sure the executable matches the source. What better way to Trojan a system than to have the administrator do it for you?

SQL Server supports Integrated Security (i.e. “Windows authentication”), where authentication of database connections is a function of Active Directory. Assuming the process that’s attempting a database connection has been authenticated by Active Directory, and the corresponding SQL Server account has been defined to use Windows authentication, the connection is established. For a utility like a database tripwire system that will be used within a single Active Directory domain, this is a very reasonable configuration, and probably far preferable to including SQL Server username/password pairs in connect strings. (But it’s also another argument in favor of dedicated accounts for the tripwire application. Don’t use a highly privileged account, such as one that’s a member of BUILTIN/Administrators, with Integrated Security. You’re probably better off using SQL Server authentication for a dedicated SQL Server account and more restricted privileges.)

If the database tripwire application is deployed in a distributed manner, such that some or all of the monitored objects are on remote servers, it’s advisable to encrypt the network communications between the servers. Either use IPSec or Transport Layer Security and certificates to encrypt the traffic between the servers.

If a server and database can be dedicated to act as the database tripwire master server, centralize as much of the system as possible on that server and do as much as possible to harden and secure it. Get rid of unnecessary services. Restrict the accounts allowed to login. Install a firewall and a host-based IDS. Turn on database auditing. (Hey, nothing else should be running on this database, so all activity is important.)

Further Research and Development

The potential for further research and development of this application is extensive.

In terms of functionality, much could be added. The obvious first priority would be the addition of more objects to the set that can be fingerprinted. The utility of the application would be greatly enhanced by a graphical interface and a more extensive reporting system. The error handling routines that were added during development should be improved and used more universally throughout the application. It would be useful to have the ability to restore the previous version of an object when an unauthorized change is detected.

It may be useful, in some cases, to adapt the application to fingerprint the *data* in certain tables. Many systems contain tables with largely static data (e.g. the object_type_lookup table in this tripwire implementation). Traditional audit tools would also work well for this situation, but so would a tripwire approach. The notion is worth considering.

In terms of security, probably the most useful enhancement would be to fully implement and test the capabilities of the system for remote monitoring of database objects. The optimum configuration for the application, from a security perspective, is probably a dedicated master tripwire database server on a hardened system that is used to fingerprint objects on client database servers. The communications between servers should be encrypted using server certificates, Transport Layer Security or IPSec, and Integrated Security. Be sure to refer to Blackburn [2], however, for a contrarian's opinion on the use of Integrated Security.

Any tripwire application relies on the integrity of the hashes to make meaningful comparisons. This system stores those values in database tables. As noted, secure the tables using all the features the database provides. As a final measure, consider encrypting the fingerprint hashes. Hashes are not encryption algorithms [17], but hashes can be encrypted, just like any other data.

One of the issues for a distributed implementation will be the need for components of the system to be installed on the client database servers (e.g. the xp_md5.dll). Ideally, a zero-install footprint on client database servers would be required, but that may not be possible, for functional or performance reasons. It would be worth considering the possibility of remotely installing some or all of these components at the beginning of each fingerprint run and removing them after the run has finished.

In any case, some effort should be taken to determine the minimal level of privilege required to perform the tripwire functions, both on the target databases and in the master tripwire database server.

Finally, it would be interesting to adapt these tools to other databases, like Oracle or DB2. There is great similarity in the structure and operation of most RDBMS products, and any of them would benefit from this capability.

Conclusions

This research has shown that development of a database tripwire application to efficiently identify changes to database objects is a very tractable problem. This function is not well served by current tools, so a database tripwire has great potential to support the work of database administrators. Databases are high-value targets, and the effort expended to safeguard these

Not only can a database tripwire be used to safeguard the integrity of databases from intruders and unauthorized use, it can also assist in monitoring and managing the day to day changes that any database undergoes as objects are added, deleted, and altered by administrators and database developers.

The current version of the tool offers tangible proof that the effort required to produce a useful product is within the means of most organizations. Furthermore, the prototype has been constructed in such a way that it can be incrementally improved, so that it can continue to be enhanced and extended to increase its value to the database administrator.

References

[1] Bardhan, Jai. "Use Sysobjects in SQL Server to Find Useful Database Information." Devx, a JupiterWeb site, a division of Jupitermedia Corp. 12 Jun. 2000. <http://www.devx.com/tips/Tip/14107>.

[2] Blackburn, Peter, and William R. Vaughn. Hitchhiker's Guide to SQL Server 2000 Reporting Services. Boston: Addison Wesley, 2004: 233-234.

[3] Busse, Fridtjof. "AIDE vs. Tripwire". Personal web site. 22 Aug. 2003. <http://www.fbunet.de/aide.shtml>.

[4] Cole, Eric. SANS Institute. Track 1 – Security Essentials. Lecture comment. SANS West Conference. 7 Mar. 2003.

[5] Gunderloy, Mike. "Give your data the power to speak with SQL Mail". SQL Server Solutions. Dec. 2004: 7-10.

[6] Hochmuth, Phil. "Profiling cybercrime: Network threats and defense strategies. Serious business." Network World. 29 Nov. 2004. <http://www.nwfusion.com/supp/2004/cybercrime/112904qanda.html>.

[7] Kelley, Brian. "SQL Server's Metadata Views." Database Journal. 15 Mar. 2003. <http://www.databasejournal.com/features/mssql/article.php/1460131>.

[8] Khanal, Shailesh. "Generate Scripts for SQL Server Objects." Database Journal. 16 May 2003. <http://www.databasejournal.com/features/mssql/article.php/2205291>.

[9] Kim, Gene H., and Eugene H. Spafford. "The Design and Implmentation of Tripwire: A File System Integrity Checker." COAST Laboratory, Department of Computer Sciences, Purdue University, West Lafayette, Indiana. 23 Feb. 1995. <http://www.cs.ucsd.edu/classes/sp99/cse227/Tripwire.pdf>.

[10] Mackey, Vic. "MD5 Hash SQL Server Extended Stored Procedure." Codeproject web site. 27 Jan. 2004. <http://www.codeproject.com/database/xp_md5.asp>.

[11] Merkin, Damian. "Introduction to SQL-DMO." SQLTeam.com web site. 18 Aug. 2002. <http://www.sqlteam.com/item.asp?ItemID=9093>.

[12] Microsoft Corporation. "FIX: Input Parameters to the OLE Automation (sp_OA) Stored Procedures Truncate at 4000 Characters". Microsoft web site. 9 Oct. 2003. <http://support.microsoft.com/default.aspx?scid=kb;en-us;325492>.

[13] Microsoft Corporation. SQL Server Books Online. CD-ROM. Microsoft Press. 2000.

[14] Ress, Frank. "SQL Server Email – vulnerability issues and prevention strategies." GIAC

GSEC Practical. 6 Oct. 2003.
<http://www.sans.org/rr/whitepapers/application/1219.php>.

[15] Sampath, Srinivas. “Yukon and the CLR.” SQLJunkies web site. 20 Apr. 2004.
<http://www.sqljunkies.com/Tutorial/46640BA1-46C1-4DF8-94AF-6ADF84DDCF81.scuk>.

[16] Sampath, Srinivas. “Using COM Objects in SQL Server.” Personal web site. 4 Oct. 2003.
<http://www32.brinkster.com/srisamp/sqlArticles/article_31.htm>.

[17] SANS Institute. Track 5 – Securing Windows. Volume 5.3. SANS Press, Apr. 26, 2004: 38-39.

[18] Spurgeon, John, and Ed Schaefer. “Entrap: A File Integrity Checker”. Sys Admin. Dec. 2004: 44-57.

[19] Stoll, Clifford. The Cuckoo’s Egg: Tracking a Spy Through the Maze of Computer Espionage. New York: Doubleday, 1989: 3-10.

[20] Strawmeyer, Mark. “Secure Your ASP.NET Application from a SQL Injection Attack”. Aug. 2003. <http://www.developer.com/net/asp/article.php/2243461>.