SQL Server Anti-Forensics:
Techniques and Countermeasures

Researched and compiled by:
Cesar Cerrudo

Edited by:
Josh Shaul

Abstract:

Database hacking has gone mainstream. Criminals recognize that large caches of valuable data are housed within database systems. Those systems have come under attack with increasing regularity and tenacity. Exploits for database vulnerabilities are now commonly posted on the internet within hours of a database patch being released. The threat is real and growing rapidly.

This paper attempts to get ahead of the curve by discussing the techniques security professionals can use to perform forensics analysis after a database attack. We focus specifically on Microsoft SQL Server 2005, however the information presented is also relevant to other database versions. A discussion of forensics is not complete without covering anti-forensics; the techniques that the most sophisticated attackers use to cover their tracks. We will take a close look at SQL Server anti-forensic techniques, then follow with a discussion of how to protect your audit trail and evidence in the event one of your systems is attacked.

The material is presented at an advanced level. While it is not necessary to be a SQL Server DBA or Security Analyst to understand the concepts, it is expected that the reader has some familiarity with Microsoft SQL Server and Microsoft Windows, and has a basic understanding of information security.

Table of Contents

SQL Server Logging Mechanisms

Let's start by reviewing the mechanisms SQL Server 2005 uses to log database activity and the facilities it uses to store the logs it generates. In researching this paper, we used installations of SQL Server 2005 with Service Pack 2 applied (the latest version available at the time). Most of the material presented applies to previous versions of SQL Server as well with minimal changes.

By default SQL Server logs information to the following locations:

- SQL Server error log
- Windows application log
- Default trace
- Transaction log

This log data is primarily located in data files (sometimes flat files, sometimes in the database) stored on the disk. Forensic investigators can also find valuable data in memory (data which is not written to disk.) This includes the database's data cache and procedure cache, both of which may contain the footprints of a recent attacker.

The SQL Server Error Log and the Windows Application Log:

SQL Server 2005 makes heavy use of both the database error log (SQL Server error log) and the Windows application log. While most log data are duplicated in the two locations, both can offer valuable information to the forensic investigator.

From SQL Server 2005 Books Online [1]:

> *"SQL Server logs certain system events and user-defined events to the SQL Server error log and the Microsoft Windows application log. Both logs automatically timestamp all recorded events. Use the information in the SQL Server error log to troubleshoot problems related to SQL Server.*
> *The Windows application log provides an overall picture of events that occur on the Windows operating system, as well as events in SQL Server and SQL Server Agent...."*

SQL Server typically saves the same information in both logs, providing the Windows application log as a convenience to help administrators correlate database errors with OS errors. Correlating the logs makes it easier to identify certain problems, but there is also a security benefit to using both log locations. Assuming the SQL Server service is not running under LOCAL SYSTEM or under a Windows administrative account (and it should NOT be!), users with DBA rights are not able to manipulate the Windows application log to cover inappropriate behavior in SQL Server.

Although neither of these logs can be disabled, SQL Server users with DBA privileges are able to erase SQL Server error logs.

Reading and Erasing SQL Server Error Logs

SQL Server error logs are saved in the LOG folder in the SQL Server install directory (typically \Program Files\Microsoft SQL Server\MSSQL.1\MSSQL\LOG). By default, SQL Server records up to 7 error log files: `ERRORLOG`, `ERRORLOG.1` ... `ERRORLOG.6`. It is possible to configure SQL Server to increase the number of logs retained, but it is not possible to decrease

it below 7. `ERRORLOG` (without an extension) is always the current error log file. Every time SQL Server is restarted a new error log is created, with the most recent log being renamed with a `.1` extension, the second most recent with `.2`, etc. `ERRORLOG.6` will always be the oldest error log in default configuration, and is deleted when SQL Server makes room for a the creation of a new log file. This process is called cycling the error logs.

Error logs can also be cycled manually by using the `sp_cycle_errorlog` system stored procedure (which calls `DBCC ERRORLOG)`. When `sp_cycle_errorlog` is executed, an entry (or message) detailing the action is logged in both the previous and the new error log files. SQL Server error log files are text files and therefore can be viewed with any text editor or using SQL Server provided tools [2]. The information they contain includes the date and time when the message was logged, the source of the message and the message itself.

For example:

```
Date                     Source        Message
-----------------------  --------      -------------------------------------
...
2007-05-09 13:53:57.34    spid12s      Service Broker manager has started.
2007-05-09 13:53:58.06    spid51s      SQL Server version: 09.00.00.3042
...
```

The following statements can be used to obtain information about SQL Server error logs:

```
Select ServerProperty('ErrorLogFileName')
```

The above statement returns the full path and name of current error log file.

```
EXEC master.dbo.xp_enumerrorlogs
```

The above statement returns a list of all error logs currently stored. The logs are listed by file number; file number 0 representing the current error log. The date for the last log write as well as the size of the log (in bytes) is also returned.

```
EXEC master.dbo.xp_readerrorlog 0, 1, NULL, NULL, NULL, NULL, N'desc'
```

`xp_readerrorlog` returns the contents of the specified error log file. The first parameter is error log file number, the second parameter is a flag that indicates if SQL Server or SQL Agent logs are being requested (1=SQL Server, 2=SQL Server Agent). The usage of the parameters with NULL values is not documented or known. The last parameter N'desc' causes the results to be returned in descending order; setting this value to N'asc' orders results in ascending order.

The SQL Server entries written to the Windows application log provide more detail than the ones saved in SQL Server error log. The most important additional information found in the Windows log is the Windows user name that originated the event. The Windows application log will list the username only when Windows authentication is used (to connect to the SQL Server).

The Windows application log also lists the type of each entry: Information, Error, Warning, etc. This simple addition is helpful as it can be used to quickly focus in on problems using the Windows Event Viewer.

Events Captured by the SQL Server Error Logs and Windows Application Logs

- Failed login attempts (optionally successful logins as well)
- Backup and Restore information
- Extended stored procedures DLL loading
- Server options being disabled/enabled (`sp_configure`)
- Database options being changed (`sp_dboption`)
- Some DBCC commands
- Error messages
- Etc.

These logs are not all inclusive. In fact, most database queries are not logged, providing few events in a typical SQL Server error log or Windows application log. The two logs rarely contain all the information needed for a forensic investigation. Other sources of evidence are required. The additional sources will be discussed momentarily. First, let's address the types of events you will not find in the error logs:

- Extended stored procedures execution
- SELECT statements
- Some DBCC commands execution
- Data Definition Language (DDL) statements
- Data Manipulation Language (DML) statements

To identify data that was accessed or changes that were made to the data or structure of the database, one has to look elsewhere.

Default Trace

By default SQL Server 2005 runs a trace in order to provide *"troubleshooting assistance to database administrators by ensuring that they have the log data necessary to diagnose problems the first time they occur"* [3]. The default trace is implemented primarily for problem solving purposes, however its biggest value may be the wealth of information it provides for the forensic investigator.

The default trace writes events to a set of files which are stored in SQL Server's LOG sub-folder (the same location where SQL Server error logs are kept). Trace files are named log_X.trc, where the X is a number. A new trace file is created under three conditions:

1. SQL Server start up or restart
2. Default trace enabled option changed from 0 to 1 (disabled to enabled)
3. The current or active trace file grows to more than 20 MB

When a new trace file is created, it is named by incrementing the number in previous trace file name by 1. For example, if previous trace file was log_50.trc, then the new file will be named log_51.trc. A history of no more then 5 trace files is kept by the default trace, so once there are 5 trace files, each time a new file is created, the oldest one is deleted.

The default trace can be enabled/disabled with the following statements:

```
exec sp_configure 'default trace enabled', 0 -- Disable
exec sp_configure 'default trace enabled', 1 -- Enable
```

To determine the current status of the default trace (enabled or disabled), examine the output from the following command (0 = disabled, 1 = enabled):

```
exec sp_configure 'default trace enabled'
```

SQL Server trace files are stored in a proprietary binary file format that has not been documented or disclosed. Microsoft provides two means of interpreting the data in the trace files. First is the SQL Server Profiler tool, which provides a nice GUI to display and filter trace events. Alternatively, you can read the trace files through the database with the function `fn_trace_gettable` as follows:

```
SELECT *
FROM fn_trace_gettable
('C:\Program Files\Microsoft SQL Server\MSSQL.1\MSSQL\LOG\log.trc', default)
```

This statement will display all the information in all the default trace files, which will likely result in a large amount of data.

SQL Server also provides a facility to identify all currently running traces (default and user defined):

```
SELECT * FROM master.sys.traces
```

This query returns a table with 18 columns. Of those 18 a few are quite interesting:

- `id`: This is the trace identifier, represented as a number
- `is_default`: This column has a value of 1 if the trace referenced is the default trace
- `status`: This column indicates if the referenced trace is running (=1) or stopped (=0)
- `start_time`: The time the referenced trace was started
- `last_event_time`: Time the most recent event was written to the referenced trace
- `event_count`: The number of events logged in the referenced trace

Note: The default trace can not be stopped or paused, however it can be disabled (see above).

Events Captured by the Default Trace

The default trace captures some of the same events that are written to the SQL Server error logs and Windows application logs. It also captures several other types of interesting and important events. The contents of the default trace can be invaluable in investigating a potential database breach.

Events that are logged in the default trace include:

- Failed login attempts
- Login creation/deletion/modification
- Use of trace related tables, functions, stored procedures, etc.
- Server options being disabled/enabled (`sp_configure`)
- Object creation and deletion
- BACKUP and RESTORE statements
- DBCC commands
- DENY, GRANT and REVOKE statements
- Etc.

Note: For a complete listing of all the events logged by the default trace, examine the properties in SQL Server Profiler after opening a default trace file.

While the default trace greatly augments the evidence provided by the error logs, there are still some critical operations that are not logged. The default trace does not provide information data that has been read or modified. To find this information, we will again have to seek other sources. Important events not logged by the default trace include:

- Extended stored procedures execution
- SELECT statements
- Data Manipulation Language (DML) statements

Transaction Log

Every enterprise database platform provides a means of reinstituting changes made to the database. Different database vendors have implemented their rollback features differently, but in general they all use a log that lists changes, and includes before and after values. Microsoft stores their rollback data in a file called the Transaction Log.

From SQL Server 2005 Books Online [4]:

> *"Every Microsoft SQL Server 2005 database has a log that records all the transactions and database modifications made by each transaction. The transaction log is a critical component of any database, and understanding and managing this log is a critical part of the database administrator's role. This is particularly true under the full and bulk-logged recovery models, which require backing up the log on a regular basis."*

Before exploring the details of the transaction log, let's explore the recovery models offered by SQL Server. There are three recovery models designed to control transaction log maintenance: simple recovery, full recovery and bulk-logged recovery. The most commonly used are the simple recovery model and the full recovery model.

When a database is set for simple recovery, minimal information is written to the transaction log, and the transaction log is not backed up. Log space is reused frequently so transaction log records are constantly being overwritten. SQL Server system databases (except the *model* database) use the simple recovery model by default.

When a database is set for full recovery, all transactions are logged and those records are maintianed until backed up. In this model, it is possible to restore the database to any point it time. This recovery model is recommended for production databases. When using the full recovery model, it is critical to back up the transaction logs regularly in order to prevent the disk from filling and the database from grinding to a halt. For more information on recovery models, check out the SQL Server Developer Center on MSDN (reference [5]).

In order to determine the database recovery model currently in use, run the following query: (Note: substitute the proper database name for the first parameter):

```
SELECT DatabasePropertyEx('databasename' ,'Recovery')
```

With this basic knowledge of recovery models, we can move forward with our discussion of transaction logs. The transaction log is implemented as a separated file or set of files that carry a default extension of .ldf. Typically transaction logs have a similar name to the corresponding database data file but with the word "log" added. For example, if a database's data file is named users.mdf, the log file is typically named something like users_log.ldf.

Transaction logs can be stored anywhere, however, many DBAs place the database transaction log in a different location than they place the database data file.

The following statement can be used to determine the name and location of transaction log and data files for any database (Note: remember to substitute the proper database name):

```
SELECT * FROM databasename.sys.sysfiles
```

The initial size and growth of transaction log files can be set at database creation time and modified thereafter by issuing an `ALTER DATABASE` statement. The transaction log needs to be managed. Left on its own, it can grow and grow until it completely fills the disk it is stored on. In order to free space, the transaction logs must be truncated periodically. This operation marks the space of the transaction log's internal structures as free for reuse; allowing it to be overwritten with new log records.

Under the simple recovery model, log truncation is automatic. It can also be forced by issuing a `CHECKPOINT` statement. In general, `CHECKPOINT` is used to force all the data sitting in SQL Server's buffer cache (which lives in RAM) to be written to disk (in both the transaction log and data files). Under the full recovery model, log records must be backed up for the log being truncated. It is, however possible to avoid backing up by running the `BACKUP LOG` command and specifying the `TRUNCATE_ONLY` option. Doing so, without first doing a proper backup is not recommended. Such a procedure will result in a broken log chain and make it impossible to perform a full database recovery in the case of a failure.

Truncating the transaction log doesn't actually reduce its size on disk, it only marks the internal records as free for reuse. In order to reduce the size of the transaction log it must be "shrunk", where the space allocated to unused internal structures is freed and returned to the operating system.

The following command can be used to shrink the size of a log file:

```
DBCC SHRINKFILE (log_name_or_id, size)
```

If the `size` parameter is not specified, then the file will be reduced to its default size (which was set at database creation time or by an `ALTER DATABASE` statement). Reducing to the default size will generally not release the free space back to the OS. To maximize the reduction, specify a `size` parameter of 1 or 0. The parameter instructs `DBCC SHRINKFILE` to try and reduce the file to 1 MB or less, and releases available space to the operating system.

SQL Server provides a simple means of displaying transaction log records for the current database:

```
SELECT * FROM ::fn_dblog(null, null)
```

Running this function will display all the active transaction log records, but the information displayed is not very user friendly. Explaining the meaning of every column returned by this function is out of the scope of this paper, but an experienced user can familiarize themselves with the output by conducting an analysis of the output.

If an analysis of a full transaction log dump is not practical, there are more intuitive ways to reduce what is returned into a data set.. All the details may not be provided, but it's a productive exercise. Try the following statement:

```
SELECT suser_sname(convert(varbinary,"Transaction SID")), Operation,
AllocUnitName, "Begin Time","End Time" FROM ::fn_dblog( null , null)
```

The output from this query is much easier to understand then the complete, raw output from `::fn_dblog`. This technique presents a list of transactions with basic details for each, including:

- Login that originated the transaction
- Transaction type
- Object name involved in the transaction
- Time transaction began
- Time transaction ended

Microsoft does not disclose the file format for SQL Server transaction logs, however, there are third party tools market that do understand the format and can display all the contents of a transaction log.

Events Captured by the Transaction Log

The transaction log provides information on many of the interesting events that are not logged in the error logs or by the default trace. Most importantly, the transaction log includes a complete listing of all data and structural changes in the database. The following events are captured in the transaction log (from SQL Server BOL [6]):

- Start and end of each transaction.
- Every data modification (insert, update, or delete). This includes changes by system stored procedures or data definition language (DDL) statements to any table, including system tables.
- Every extent and page allocation or de-allocation.
- Creating or dropping a table or index.
- Rollback operations
- The transaction SID (Login that initiated the transaction)

The above mechanisms deliver most of the data necessary to get a complete picture of what has occurred in the database, but there is still one critical piece missing. Select statements. The mechanisms we have looked at so far are not designed for the intense workload associated with tracking reads from the database. Logging select statements is a difficult problem, and while it can be done with a user-defined trace, it is generally accepted that this function is best served by a third-party database activity monitoring product.

The following list of events will not be found in the transaction log:

- SELECT statements
- Extended stored procedure execution
- DBCC statements

SQL Server Data Files

Data files are the physical storage where the database data is saved. One database can have multiple data files; the main data file has a default extension of .mdf, and the secondary data files have a default extension of .ndf. The structure of SQL Server data files is highly optimized and forms much of the core of Microsoft's Intellectual Property for the database engine. The structure of these files is a closely guarded trade secret and is not publicly known. The intention is that only the SQL Server database engine reads and writes data from these files. Users access the data by running SQL queries via the database engine.

Data files store tables, indexes, user data, and metadata. Every DDL or DML statement that causes a change in the state of the database will result in a modification to the data files. Data files can be reduced in the same way as transaction log files. When data is deleted from the database, space is marked as free in the data files and can be released to the operating system using `DBCC SHRINKFILE`.

SQL Server Memory

SQL Server makes heavy use of system memory in order to provide fast access to commonly used data and functionality. The data stored in memory is grouped into a set of caches, the most important of which are the data cache and procedure cache. As you can probably guess, the data cache stores data read to and written from the data files, and the procedure cache stores execution plans for regularly executed SQL statements. While the contents of the SQL Server memory are quite volatile, and there are no guarantees of what you will find in there at any given time, they could be invaluable in investigating a database attack. Everything that happens in the SQL Server appears in memory, at least momentarily. It's important to understand that the data in memory grows stale quickly, especially in busy systems. If you are going to get value out of the available data, you will need to act quickly and examine the contents while an attack is in progress, or immediately after it has concluded.

Data cache information can be retrieved using the `DBCC PAGE` command. Access the procedure cache by running:

```
SELECT * FROM sys.syscacheobjects
```

This command displays the stored execution plans from the procedure cache. The column named `sql` displays the actual SQL statement for which the execution plan has been cached.

If you want to explore the contents of the data cache, you will require the virtual addresses that have been allocated to SQL Server. The following query will display the location (address), size, and attributes for the range of pages assigned to the database:

```
SELECT * FROM sys.dm_os_virtual_address_dump
```

SQL Server memory can be read directly by running the `DBCC BYTES` command. `DBCC BYTES` is simple; it takes a starting memory address and a length and simply reads data from memory and returns it to the user. Using this command, it's possible to see everything that is occurring within the database. Even reading clear text passwords of recently created or modified SQL Server logins is possible if you know where in memory to look. However, the privilege to execute `DBCC BYTES` is restricted to SQL Server administrators.

SQL Server Anti-Forensics

As the Forensics Wiki [7] succinctly puts it, "Anti-forensic techniques try to frustrate forensic investigators…" The whole idea behind anti-forensics is to erase or damage the evidence of an attack or inappropriate activity. It's the next level of escalation in the battle between hackers and computer crime investigators. The remainder of this paper will describe some of the anti-forensic techniques that attackers can use to cover their tracks and some of the tools available to thwart their efforts.

In order for these techniques to be successful, an attacker must be able to run commands with database administrator (sysadmin) privileges. This requirement is common across the anti-forensics field. For example: erasing the evidence of most operating system level attacks requires root or administrator privileges. Details on how to obtain these powerful privileges will not be discussed in this paper, but it is clear that obtaining powerful privileges is not difficult if a database has not been properly secured. Possible avenues to obtain sysadmin rights include:

- Exploiting a vulnerability (although SQL 2005 only has a few of these)
- Finding a valid login and password (by brute force, guessing, stealing, or with the help of a trojan)
- Taking advantage of weak or missing access controls
- Being an evil DBA
- Exploiting a vulnerability in an application or in the OS

Hacking SQL Server 2005 is not easy. Covering your tracks is even more difficult. In a default installation of SQL Server 2005:

- Failed login attempts are logged.
- Logs are written to the SQL Server error log and the Windows application log. This cannot be disabled.
- The Default trace is running.
- The Recovery model is set to simple on system databases (except on *model*) and to simple or full on user databases.
- `xp_cmdshell` is disabled.
- SQL Server runs under a low privileged operating system user account.

For the purposes of our discussion, we define Hacking SQL Server as:

- Stealing data
- Inappropriately changing data
- Taking over the Windows server (SQL Administrator != Windows Administrator)
- Leaving little or no evidence of the breach

Erasing the Evidence: An Example

In this fictional scenario, we will assume that our attacker doesn't need to worry about failed login attempts being logged. She has obtained a valid login with elevated privileges and can log in without problems. We will also assume a default installation of SQL Server 2005.

Once the attacker has logged in, she starts running commands. At this point, her problems begin: some commands will be logged in three places (SQL Server error log, Windows application log and default trace), other commands will be logged in two places (SQL Server error log and Windows application log) and a few other commands will be logged in one place (default trace). Also DML and DDL commands will be logged in transaction logs, and of course the data in data files will change as a result of them.

Being sysadmin won't help our attacker to delete Windows application logs, but she can delete SQL Server error logs. The database provides a simple stored procedure to do this:

```
exec sp_cycle_errorlog
```

`sp_cycle_errorlog` is used to create a new error log file, which in turn deletes the oldest one in the system. Calling `sp_cycle_errorlog` repeatedly will clear the previous error log files. Doing so will also leave behind some clues. Some error log files will be missing and a log cycling message will be written in each new log file, in the Windows application log, and in default trace.

The default trace is also easy to attack for a sysadmin. Our attacker can delete the trace file by executing:

```
exec sp_configure 'default trace enabled', 0
```

This procedure will disable the default trace. The attacker can then delete the associated file. Disabling the default trace however will be logged in both the SQL Server error log and the Windows application log.

Since we're running with the default configuration of SQL Server, our attacker can run any SELECT statements she pleases without triggering logging. These SELECT statements are written to memory in the procedure cache, but that cache can easily be cleaned (along with other caches) by running the command

```
DBCC FREESYSTEMCACHE('ALL')
```

When running this or any DBCC command, its execution will be logged in the default trace. This logging can be eliminated by previously disabling the default trace.

Attack Scenario: Own the Server, Own the Data

In this scenario, we will take a step back and examine a possible attack against a SQL Server database. We will examine how a knowledgeable attacker can steal and modify data in the database, and even gain complete control over the database server while leaving minimal evidence behind.

Our attacker has already decided that SELECT queries will be the primary means to steal data. SELECT is cleary not the most efficient way to steal data in bulk. Time must be spent looking for valuable or relevant data, and ueries must be written and run. It is a time consuming attack.

Taking a full database backup would be much quicker. With one backup, the attacker can obtain everything in the system without having to spend time searching for interesting data. However, the backup attack poses significant challenges when it comes to anti-forensics. Databases can be huge. In order to create a backup, a great deal of free disk space is required on the server. If the space to store the backup is available and the backup succeeds, the attacker must find a way to copy the backup file to their own system. Backing up will leave other evidence: the log chain will be broken, a file will be created in the server, and the BACKUP command will be logged everywhere, including places not covered above. It's possible, but highly unlikely, to execute a backup attack without leaving hard evidence of the attack. As such, our attacker has chosen SELECT as the methodology for her attack.

Disabling Error Logging

The first step for our attacker is to disable all of the SQL Server logging mechanisms. However, as we've discussed, this leaves behind evidence. Instead of disabling logging in the normal or intended way, the same effect can be achieved by modifying (or patching) SQL Server process memory. To do this, the attacker to be able to put code inside the SQL Server process, and get it to run. This may sounds unlikely, difficult, or impossible, but in actuality is simple for a skilled database user.

The simplest means of running code is to run a binary file directly via `xp_cmdshell`. The problem is, `xp_cmdshell` is disabled by default and the enablement will be logged everywhere. If `xp_cmdshell` is enabled (non-default configuration) using it is the best and easiest way to run code that can patch SQL Server memory in order to disable logging. This can be done without leaving any evidence. By default `xp_cmdshell` execution is not logged.

Another option is available. One that does not require any modification to a default SQL Server installation, since it relies on a function that can't be disabled nor removed. Specifically, SQL Server provides Extended Stored Procedures (XPs). These are similar to normal stored procedures, except the code is implemented in DLLs instead of SQL. SQL Server dynamically loads the corresponding DLL when a XP is executed for the first time.

XPs can be added to SQL Server using `sp_addextendeproc` system stored procedure:

```
EXEC master..sp_addextendedproc 'xp_test', 'xp_test.dll'
```

`sp_addextendeproc` calls `DBCC ADDEXTENDEDPROC( @functname, @dllname)` passing the XP name and DLL name as parameters. The name of the XP created will be logged in the default trace, as will the execution of the DBCC command, but no entries are generated in the SQL Server error log or the Windows application log. After `sp_addextendeproc` is executed, a row with information about the XP is added to the `sys.sysobjvalues` and `sys.sysschobjs` system tables in the `master` database. Because table data is being modified (by adding a row), this action will be logged in the transaction log file and data will be changed in the data file.

In a forensic investigation it's important to be able to quickly list the any and all user created through extended stored procedures, and then compare them with what is known to belong on the server. It is recommended that you run the following on your critical SQL Server 2005 databases and save the results:

```
SELECT * FROM master.sys.extended_procedures
```

If you suspect an attack, run the statement again and compare it with the stored results. If there are new XPs listed, be sure to scrutinize their functionality and place on the system.

After an XP has been added to the system (we'll call it `xp_test`), it can be executed with the simple command:

```
EXEC master..xp_test
```

SQL Server starts processing this command by loading the XP DLL into process memory. The name of the DLL and the XP are logged in both the SQL Server error log and Windows application log. Log entries are created only the first time an XP is run, Subsequent calls to an XP are not tracked. As such, it is especially important to carefully review the logs. Missing one event can result in missing the only real evidence of an attack.

SQL Server's mechanism for logging the loading of XPs and DLLs has a design weakness that

makes it possible to execute an XP without causing any logging. The key to this attack is the order of operations implemented by the database. Immediately after a DLL has been loaded into memory, SQL Server executes the DLL function `DllMain()`. This happens before any entries about the XP or DLL are written to either the SQL Server error log or Windows application log. Code placed in the function `DllMain()` can be used to disable logging on both logs without leaving any trace.

Writing code to patch SQL Server process memory to disable logging is not difficult. It requires a skilled developer with time, patience, and a debugger. This is a sophisticated attack, but it is possible for someone with the right skills. Organizations running a database that is loaded with sensitive and valuable data must acknowledge that a determined attacker can be successful in executing this exploit.

This is the type of exploit that can be built on one system and executed on another, meaning it's something that a hacker can create in comfort and. It is also possible for someone to write and then post exploit code on the Internet to perform this memory patching. If this scenario, it is easy for anyone with Sysadmin access to successfully run this attack.

No exploit code is shown in this paper. However, we have completed the research to demonstrate that the attack is possible. It requires blocking two function calls, one to stop logging in the Windows application log, and one to stop logging in the SQL Server error log. The mechanics of how to block these function calls is outside of the scope of this paper, however, it is a simple attack that takes no more than modifying a few dozen bytes in memory.

SQL Server 2005 uses the following APIs to write to the error logs:

- Windows application log: `ReportEventW` from `Advapi32.dll`
- SQL Server error log: `NTWriteFile` from `Ntdll.dll`

With a DLL in hand designed to disable SQL Server logging on error log and Windows application log, the attack can begin. Our hacker loads and executes the extended stored procedure that runs the attack code. Logging to both the SQL Server error log and the Windows application log are now disabled. However, this action did not go unnoticed. Three entries detailing the creation of the extended stored procedure have been recorded by the default trace.

Disabling the Default Trace

At this point, you may be wondering why our attacker didn't disable the default trace first, before disabling the error logging mechanisms. Disabling the default trace leaves evidence in both the SQL Server error log and the Windows application log. Exactly what our attacker is trying to avoid. Instead, she chooses to disable the default trace now, knowing that she will have to clean up the three entries about the XP and that she will need to re-enable the trace before leaving (otherwise the fact that the default trace is disabled is evidence of an attack).

Disabling the default trace is simple, requiring only one SQL command:

```
exec sp_configure 'default trace enabled', 0
```

This change takes effect immediately; no database stop/start is required. However, the trace file containing the three entries showing the loading of the attack XP remains. Before leaving the database, our attacker must erase or overwrite this evidence.

The simplest way to clean up the trace file is to fill it with garbage data up to its 20 MB limit. It's simple to add functionality to the XP that locates and overwrites the contents of the default trace file. Once the attack is complete and the default trace reenabled, it will likely be no more then a few days before this old trace file is automatically deleted by SQL Server. If someone examines the file while it still exists, they will find no useful information, just a corrupted trace with no clear explanation of how the corruption occurred.

Finally, it is also possible to disable the default trace by patching SQL Server memory with a method similar to that used for disabling the SQL Server error log and Windows application log. However this is a complex attack that takes time to research and construct. Corrupting the file is a much quicker path to essentially the same result.

Clearing the Master Transaction Log and Data File

Our attacker's primary goal is to steal data from the database. A secondary, but equally important goal is to leave little or no evidence of the theft. So far, our attacker has gained access to the database and has disabled three of the primary logging mechanisms offered by SQL Server. However, she still has hurdles to overcome in order to complete the task.

Next is the master database transaction log. Since all changes to the database are recorded in the log, records were created when the XP was added. Unfortunately for the forensic investigator, clearing these entries is trivial for our attacker. Remember our discussion of recovery models? The master database is configured with the simple recovery model by default. This means that space in the transaction log is recycled and reused on a regular basis. It is likely that the evidence will erase itself without the intervention of our attacker.

Regardless, our attacker is taking no chances. Her plan is to remove the attack XP before leaving the system (again, removing the evidence). Once that change is complete, she will ask the database to shrink the *master* database data file and transaction log, releasing the space created back to the operating system. This is a simple task that our attacker accomplishes by running the following commands in sequence:

```
DBCC SHRINKFILE (1,1)
DBCC SHRINKFILE (1,0)
DBCC SHRINKFILE (1,1)
```

These commands shrink the *master* data file. They are run repeatedly because experience shows that as the files are shrunk, internal data file structures reorganizes, making more free space available to release to the OS. Three times is sufficient. The same procedure (only with a different argument) shrink the *master* transaction log file:

```
DBCC SHRINKFILE (2,1)
DBCC SHRINKFILE (2,0)
DBCC SHRINKFILE (2,1)
```

It is possible that the name of the attack DLL and XP related information will remain somewhere in the *master* database data file (*master.*mdf) in space marked as free after the XP is removed from the system. To eliminate the risk of her attack being discovered by a careful examination of the contents of that data file, our attacker wipes the free space in that file before shrinking it and freeing the contents. This is accomplished by writing SQL code that runs in a loop and essentially does nothing. As a side effect of this process, it will overwrite the contents of memory in the *master* data file:

```
WHILE @i<1000
BEGIN
```

```
      BEGIN TRAN
      ... (code setting @randomvalue in each iteration)
      DBCC  ADDEXTENDEDPROC(@randomvalue, @randomvalue)
      ROLLBACK TRAN
      SET @i=@i+1
END
```

If you look carefully, you will see that this code adds a new XP and then removes it (by rolling back the transaction). It does this in a loop, so the same operation is repeated over and over again 1,000 times. This has an impact on the contents of the *master* data file because of the way the database performs its internal bookkeeping. As soon as the XP is added, records about the XP are added to the *master* database (in the free space). This is done before the transaction is complete (or committed). When the transaction is rolled back, the records are deleted, and the space they occupied is again marked as free. By repeatedly adding and deleting XPs, the free space in the data file is overwritten with garbage data.
There is a little problem by running the above statements while they will successfully overwrite the name of the XP and DLL, some deleted records will remain in the datafile no matter the file is shrunk, a smart forensic investigator could find them by doing some manual analysis of the datafile but the only useful evidence that he would obtain would be the date that the records were created, if the statements are not run then forensics investigators could also find the DLL name together with the file system path, if it's a UNC patch then it would be a useful lead to find more evidence.

Because the *master* database transaction log may also contain remnants of the XP and DLL names in the free space, our attacker has also wiped it clean using a similar procedure:

```
WHILE @i<1000
BEGIN
      CHECKPOINT
      SET @i=@i+1
END
```

This time, the loop runs a `CHECKPOINT` statement, forcing the database to push any changes sitting in memory to the data file. Since there are no changes to write, the only impact `CHECKPOINT` has is to place an entry into the Transaction log. By repeating it over and over again, our attacker can be sure that all free records that remain after shrinking are overwritten with meaningless data. Wiping the transaction log is performed after shrinking the files since shrinking seems to generate entries on transaction log.

Cleaning up SQL Server Memory

The final source of forensic evidence that our attacker must sanitize is the SQL Server procedure and data caches. The database provides simple functions to do this:

- `DBCC FREESYSTEMCACHE('ALL')`: Erases the procedure cache and other caches
- `DBCC DROPCLEANBUFFERS`: Erases the data cache

Normally, running these commands would be problematic since they leave behind evidence in the default trace. However our attacker has already disabled the default trace. As a result, these DBCC commands are not logged and will go unnoticed.

The first command could rise some alerts because the procedure cache is fully erased and future executions will force SQL Server to recompile and rebuild execution plans, this could cause the server to slow down and be noticed, then if DBAs look at the problem they could find out that something happened. Just for being extra careful and to not raise any alerts there is a

nice trick to avoid erasing all the procedure cache and just erase the statements that were ran by the attacker. The trick consists in running all the statements from master database using full qualified name:

- SELECT * FROM targetdatabase..sometable
- UPDATE targetdatabase..sometable WHERE … etc.

In this way the statements get cached from master database and not from the database were the object accessed (table) resides in. Stored procedures and views must be avoided since they can get cached from the target database because they are compiled if they weren't already in the cache. Since all the statements will be cached from master database then the attacker can run the next command that will only remove those statements:

- DBCC FLUSHPROCINDB(1) -- 1 is the id of master database

Avoiding Additional Logging

Our attacker has set herself up to perform many actions in the database without being logged. However, logging is not fully disabled; the transaction log continues to function. Our attacker needs to tread carefully. She needs to avoid running statements that will be logged in user database transaction logs. Cleaning these logs is not difficult, but doing so will leave a trail of evidence because user databases are frequently backed up and their logs files are huge. If our attacker "cleans" a transaction log by issuing a `BACKUP LOG .. WITH TRUNCATE_ONLY` statement, the log chain will be broken. If a database recovery is needed, the missing logs will be noticed and will be a good indication that something is awry. In addition, using `DBCC SHRINKFILE ...` will reduce the size of the transaction log to something smaller than usual, which can easily be noticed and is an indication that something is wrong.

Our attacker is focused on stealing data. However, if her list of goals included adding or modifying data, or changing the structure of the database, there would be no means to avoid leaving clues in the transaction logs. Were she giving hacking tips to a friend, our attacker would recommend running commands that make changes (DDL and DML commands) as another SQL Server login (not the one used to break in and disable logging).

Running commands under another security context (login) is also a simple task. Using the commands `SETUSER` [8] or `EXECUTE AS` [9] it is possible to impersonate any SQL Server login. Using impersonation can send an investigator on the wrong tracks. Any entries recorded in the transaction logs will refer to the user being impersonated, not the real user logged into the database. Typically the best user to impersonate is the DBA for the database. They are most likely to ignore entries in the transaction logs that are tagged with their login ID. They are also the ones with full control of the database and therefore have the means to make changes at will. Incriminating the DBA may even lead to the DBA covering up \the attack in order to avoid being questioned, accused or even fired over the incident. Of course, any SQL Server user could be impersonated. As such, all DBAs should be properly monitored.

Completing the Attack

Back to our attack story. Our attacker has penetrated the database and disabled logs. Remember, we started with the assumption that our attacker is a sysadmin in the database. She has no rights in the operating system, and since this is a default configuration of SQL Server, `xp_cmdshell` is disabled and cannot be used.

Let's turn our focus back to where we started, with an extended stored procedure and a DLL. The same XP that loads the DLL that patches memory to disable logging can implement other

functions as well. It is just a DLL and can do whatever the SQL Server service account has the rights to do. Although SQL Server typically runs under a low-privileged OS user, it does run as a service and therefore has one very important privilege: Impersonation.

Elevating OS Privileges with Impersonation Tokens

On Windows every service can impersonate (act in behalf of other user accounts). If a Windows administrator logs on to SQL Server, it becomes possible for SQL Server to run code in the context of that administrator account. SQL Server doesn't provide any built-in functions to impersonate users. However, as we have already seen, there are many things SQL Server can be forced to do with the right (or wrong, depending how you look at it) user is sitting at the controls. If a vulnerability has been exploited in SQL Server giving an attacker control, and a Windows administrator logs in (or was already logged in at the time of the attack), a highly skilled attacker can force the database to run code as administrator, and thereby take ownership over host Windows OS. This is awell known issue which was described in detail by David Litchfield (a well known researcher) in the past [10]. Because of this issue, it is recommended that you avoid making connections to SQL Server using a Windows administrator account.

Our attacker has done her homework and is aware of these issues. She has all the research in hand including Litchfield's paper about snatching security tokens, Cesar's presentation about token kidnapping [11], and another paper providing sketchy details on a Windows design weakness that allows a service account to essentially become LOCAL SYSTEM [12]. Using token kidnapping techniques, it's possible for any SQL Server sysadmin to elevate privileges to LOCAL SYSTEM on any Windows 2003 system. We could say that in Windows 2003 a SQL Server administrator==Windows administrator. She now begins the search for impersonation tokens using her XP and the code in her DLL. Odds are very good that she will find a token she is looking for.

Once our attacker obtains the privileged impersonation token, she quickly gains complete control over the Windows OS. At this point she can take or modify files. She can load code such as keyboard sniffers, rootkits, or even viruses. She can even use attack tools to crack OS user password, likely opening the door to easily attack other servers in the network. The attack is complete and successful. And there is almost no trail of evidence to indicate what has occurred.

Countermeasures

The attack described above is feasible, and while it certainly requires skill and knowledge to execute, those skills are not beyond most database developers and database administrators. Sophisticated data theft is a real threat. However, proactive organizations can implement effective countermeasures that can be used to stop attacks, collect forensic evidence, and foil the plans of even the most advanced database thieves.

Database Vulnerability Assessment

The first step in preventing attacks is to lock down all of the systems on your network. Begin with the critical systems that contain sensitive data, but don't stop there. Since an attacker can take over a poorly protected low priority system and use that access as a launching pad to gain access to a high priority system, it's key to protect everything connected to your network.

If you don't already have a program in place for locking down your databases, the fastest and most effective way to get started is by using a database vulnerability assessment tool. The tool will provide you with a set of best practice guidelines on how to securely configure your databases. It can also be used to efficiently measure adherence to those guidelines.

Database vulnerability assessment will ensure that you have appropriate access controls in place, security features enabled, security patches applied, and strong password use enforced. By ensuring that your databases are properly locked down, you can block an outside attacker's entry into your system or an insider's ability to escalate their privileges in a system that they have legitimate access to.

Database Activity Monitoring

Once your systems have been properly locked down, the next step is to monitor the activity of anyone using or attempting to use the database. Database activity monitoring allows organizations to look at all database queries in real-time in order to detect attempts at database attacks, and in order to track user activity. Database activity monitoring is especially important if a database cannot be properly locked down (because the application using the database requires insecure settings or does not support the latest security patches). It's also critical if the attacker is the DBA herself.

A strong database activity monitoring solution would have detected our attack example at several steps along the way, and most importantly, would have been able to preserve the evidence of the attack in a protected repository, outside our hacker's reach.

Let's examine the specific operations that would have been detected by a database activity monitoring system, and how the evidence of those events would have been preserved, despite our hacker's ability to use anti-forensic techniques to defeat the native logging capabilities of the SQL Server database.

A comprehensive database activity monitoring solution will monitor Microsoft SQL Server activity using a host agent approach. A lightweight software component runs on the database server that observes all queries run against the database, and then reports information on what has been run back to a secure, dedicated data store in real-time. No monitoring data is stored on the database server that is being monitored, ensuring that forensic data cannot be deleted or altered, even by an attacker with sysadmin level access. This real-time mechanism

of shipping log information off the database server is the key component that ensures forensic evidence will be preserved after the attack. The attacker has no means to locate or access the protected data store.

The first activity of the attacker that would have been captured is the login to the database. The date/time, username, host and application used are all captured. This first step provides a forensic investigator with important information:

- When the attack began
- Which database user initiated the attack (defeating the use of Impersonation)
- Which host initiated the attack (allowing for examination of logs on the host to collect further evidence)

Next, the database activity monitoring tool would have captured and alerted on the loading of the attack XP. Everything from the time the XP was loaded to the name of the DLL containing the attack code will be captured and logged. This allows investigators to examine both the target system and the attacker host for evidence of the DLL used, which may provide detailed information into the exact steps taken to patch SQL Server memory or perform other OS level operations.

In the case where the attacker not only loaded an XP, but later came back and ran it, that execution would also be logged.

Activity monitoring tools capture and log DBCC command execution. Therefore the shrinking and backup of the data files and transaction logs are detected and logged. The DAM solution also captures stored procedure execution, with particular focus on key system stored procedures such as `sp_configure`. The disabling of the default trace would be captured in the secure repository, and an alert would be raised within a 3rd party SIEM device that has been deployed to monitor and correlate security events within the enterprise.

Finally, the meat of the attack would be captured and logged. SELECT statements, data modification using DML statements (INSERT, UPDATE, and DELETE), and schema modification using DDL statements (CREATE, ALTER, and DROP) would all be logged. Special attention would be given to queries against sensitive data, with real-time alerts forwarded to the SIEM when a large set of sensitive data is returned by a query, or when an unusual application (such as an ad-hoc query tool) is used to access any sensitive data.

The database activity monitoring system not only preserves the evidence of the attack, it provides real-time intelligence indicating an attack may be occurring, and allows security teams to immediately react and potentially catch the perpetrator in the ac . Sophisticated attacks require sophisticated countermeasures. Database activity monitoring delivers state of the art protection for critical enterprise databases. Deploying database activity monitoring can mean the difference between an attempted attack and a major data breach that can cause irreparable damage to a business or brand.

Conclusion

Microsoft does a great job of building security into the SQL Server 2005 database system, but their implementation is not perfect. A slew of capabilities for collecting forensic data are available and enabled by default. However, each can be defeated by a skilled and patient attacker. This kind of problem is not specific to SQL Server or to Microsoft. The same types of issues exist in other major database platforms, including Oracle, DB2 and Sybase.

We should not throw up our hands and give up on databases, computers, or the internet. We do need to recognize that it is war out there, and anybody could be the next target. By building an effective layered defense around critical IT assets such as database systems, it is possible to detect and prevent even the most sophisticated of attacks. It's just a matter of being proactive, staying vigilant, and picking the right tools to do the job.

About the Authors

Cesar Cerrudo is the lead researcher for Application Security Inc's Team SHATTER and is the founder and CEO of Argeniss, a security consultancy firm based in Argentina. He is a security researcher and consultant specializing in application security. Regarded as a leading application security researcher, Cesar is credited with discovering and helping to eliminate dozens of vulnerabilities in leading applications including Microsoft SQL Server, Oracle database server, IBM DB2, Microsoft BizTalk Server, Microsoft Commerce Server, Microsoft Windows, Yahoo! Messenger, etc.

Cesar has authored several white papers on database, application security, attacks and exploitation techniques and he has been invited to present at a variety of companies and conferences including Microsoft, Black Hat, Bellua, CanSecWest, EuSecWest, WebSec. HITB, Microsoft BlueHat, etc. Cesar collaborates with and is regulary quoted in print and online publications including eWeek, ComputerWorld, and other leading journals.

Josh Shaul is the Vice President, Product Strategy with Application Security, Inc. He is the foremost security policy and standards guru at the firm, with added expertise in trusted computing and application-level security issues. He is responsible for setting the technical direction for AppSecInc products and for helping customers with the development of strategic database asset protection, utilizing DbProtect™, the company's industry-leading database security suite.

Josh is the author of the critically acclaimed *Practical Oracle Security: Your Unauthorized Guide to Relational Database* and is a frequent speaker at major industry conferences and events, including: Microsoft TechEd, McAfee FOCUS, CSI Exchange, GFirst, IOUG COLLABORATE, and others.

References:

[1] SQL Server logs
http://msdn2.microsoft.com/en-us/library/ms191202.aspx

[2] How to: View the SQL Server Error Log
http://msdn2.microsoft.com/en-us/library/ms187109.aspx

[3] Default trace enabled option
http://msdn2.microsoft.com/en-us/library/ms175513.aspx

[4] Understanding and managing Transaction Logs
http://msdn2.microsoft.com/en-us/library/ms345583.aspx

[5] Overview of recovery models
http://msdn2.microsoft.com/en-us/library/ms189275.aspx

[6] Transaction Log logical architecture
http://msdn2.microsoft.com/en-us/library/ms180892.aspx

[7] Anti-Forensics techniques
http://www.forensicswiki.org/wiki/Anti-forensic_techniques

[8]SETUSER
http://msdn.microsoft.com/en-us/library/ms186297.aspx

[9]EXECUTE AS
http://msdn.microsoft.com/en-us/library/ms181362.aspx

[10] Snagging Security Tokens to Elevate Privileges
http://www.databasesecurity.com/dbsec/db-sec-tokens.pdf

[11] Token Kidnapping
http://www.argeniss.com/research/TokenKidnapping.pdf

[12] The Weakness of Windows Impersonation Model
http://www.gentlesecurity.com/04302006.html