

Executive Summary

Overall, there were a number of important findings during this audit – but none of them represents a very serious problem. Nearly all of them can be easily remedied. Given the scope of this audit, the best way to summarize findings is to break them down against two roles, or disciplines, frequently associated with database applications – namely from a development perspective, and from a sysadmin/dba perspective.

Development

From a development and application interaction perspective, there were no glaring problems detected during this audit. There is an issue, however, with the way UPDATES report on their activities, which is covered in more depth below. Otherwise, everything was pretty sound, which is a big plus – as it's easy for a skilled admin to fix the vast majority of the problems discovered through a bit of interaction with the server. (As opposed to the hassle that would be involved had these problems stemmed from faulty or bad code.)

Sysadmin / DBA

Security: There are some pretty major issues with security, which are detailed below, but they're effectively just a question of a few mouse-clicks (and a few code changes) away from being fixed. Furthermore, REALLY solid security for your applications isn't that far away given some of the coding choices (like securely encrypting customer information (like passwords) while at rest).

Backups/Recoverability: Key data is being backed up and copied off box. I've got a couple of recommendations on improving how this takes place to help make it more reliable included below.

Server and Database Settings: Placement of database files, and file growth for databases should both be addressed. Neither of these will harm day to day operations, but the first issue could impact 'up time', and the second issue will definitely affect performance and scalability. Both issues are easy to fix (though addressing the first will result in momentary down time).

System Maintenance: One of SQL Server's great selling points is that it can do pretty well without a DBA to maintain and fine tune it. However, as an application built on SQL Server becomes more and more successful (i.e. logs more data), it begins to need a DBA to help it perform optimally. Currently, there really isn't a lot of data in place on DEDICATED, but there is still some room for performance improvement.

Importantly, none of the issues listed above represent anything that can't be fixed with a modicum of effort. Meaning that, overall, things are in good shape.

Findings

Good

- Code is generally solid. Transactions are used where logical, and error handling is solid where needed.
- SQL Injection (a major security concern) is not possible with current code.
- Excellent use of raising custom errors. RETURN parameters (discussed below) are another good way of indicating failure, but tend to suffer from 'magic number' issues in terms of factors (i.e. an integer return code has to be translated into some sort of an error).
- There was nothing that made me want to 'run screaming from the building.' I've seen lots of spooky things in production on a number of systems – nothing like that here (so, while this may sound trite – it's not).

Critical

There was only one critical issue, revolving around the misuse of administrator permissions. I've broken this one issue into two constituent 'violations' for purposes of clarity.

- **Too many users with sysadmin/administrator privileges.** There are roughly 5 users that have admin (full access) permissions on DEDICATED. Generally, there shouldn't be more than 2 (sa and BUILTIN\Administrators – though many DBAs will remove the later).

The Problem: Too many users with sysadmin permissions can make auditing difficult, and provide a much wider attack surface to hackers.

The Solution: This won't be too hard to fix. For starters, we should just discuss the nature and need of some of the existing sysadmin accounts, and then work towards dropping all accounts but sa. We can then create other accounts for 'daily' use on the server – such as looking at data, etc. Having non-admin accounts for daily access is a best practice and lowers the potential for inadvertently doing damage to core systems/data. (I'd say that within about 1 hour we could get this problem changed from being a 'worst' practice to a 'best practice' with very little pain to end users (you).)

- **Web Application Accessing Database as sysadmin/administrator.**

The Problem: Security compromises are usually not 'all or nothing' issues. Generally hackers manage to take advantage of one small aspect of an application, and then use that to further look for additional areas to exploit. Allowing the web application to access the database as a system administrator means that if the web app is compromised, hackers will have gained unfettered access to the database, data, and the computer account that the database service runs on (which is currently part of the Trusted Computing Base (i.e. an admin on DEDICATED)). In other words, if hackers were able to somehow access the database via the web application, they could take control of the entire box and all data stored in your databases.

Mitigating Factors: Microsoft's ASP.NET has proved to be a very stable application platform. Chances of hackers exploiting your web application are currently very small.

The Solution: A simple solution would be to have the web application connect as a 'normal' database user (as the fixed SQL Server database role known as db_owner (i.e. an owner of the database)). This would allow the web application to execute any stored procedures as needed and would be a very easy fix. However, this would still provide too much access for hackers if they were able to gain access. A better practice, though it takes a good deal of work, would be to create a custom database role tailored explicitly for you applications. This approach would adhere the industry best practice of 'least privilege' (only giving users the bare minimum access that they need to accomplish a task – to avoid potential mistakes/blunders, and to lower attack surface/exploits should access to the account become compromised.)

Warnings

There were a number of issues that should require immediate attention:

- **SQL Server Running as Local System.** SQL Server is currently set up to run on the LOCAL SYSTEM account.

The Problem: Running SQL Server with LOCAL SYSTEM credentials means that it effectively runs as administrator or as part of the TCB (Trusted Computing Base – i.e. has full reign over system settings, access, resources, etc. – in effect, it's part of the Operating System). If hackers are able to gain access to SQL Server via a sysadmin account, they can easily create a back door on to the server – and thereby completely take it over.

The Solution: Restricting access, i.e. fixing the two critical issues listed above, will seriously diminish the impact of this concern – down to the point where it won't be an issue. (A best practice would be to run it as a normal user account and restrict access – but that can result in side effects against an already operating server (i.e. it's best done during setup and installation),

and at this point won't really provide much tangible benefit if access to sysadmin accounts is restricted.)

- Database File Placement.** DEDICATED currently has two IDE drives, C: (the system drive), and D: (a data drive). Currently all databases are on the system drive.

The Problem: Neither drive is RAID\redundant, so from a reliability standpoint, there's not too much of an issue on where the databases are placed. From a scalability standpoint, however, it would be better to place the databases on the data drive as this would give the OS a dedicated drive for normal system operations, and would give SQL Server a dedicated drive for data intensive operations.

Mitigating Factors: Currently there's just not enough data for this to be an issue. This won't be an issue until total database sizes cross a few Gigabytes in size. (It won't become a problem until the databases become MUCH larger (in terms of 10s of Gigabytes).

The Solution: Move the database files to the data drive. This is a very simple operation – but will require a few seconds (well, usually a minute or two) of downtime for each database as it is moved from one disk drive to the other.
- Database File Growth.** SQL Server databases are stored in normal 'operating system' files. As more and more data is placed into these files one of two things will occur once all available space is used up: Either the database will 'grow' the files (either the data file, or the log file) as needed using preconfigured settings, or the preconfigured settings will prohibit growth, the data file will not be allowed to grow, and attempts to add new data will fail (and the data will likely be lost – unless 'help' by the calling application).

The Problem: Data files (both database (.mdf) and log files (.ldf)) are pre-configured to grow by the defaults values supplied by SQL Server – at 10%. In other words, each time space becomes low, each database grows by 10% to keep enough disk available for future additions. The problem with this is that this can lead to a fair amount of fragmentation – which decreases performance.

Mitigating Factors: There isn't a ton of data currently, so fragmentation won't be an issue – but as the databases grow and as load increases, this can affect performance on heavily trafficked systems.

The Solution: DEDICATED currently has plenty of disk available. A good idea would be to set each of the primarily used databases (<dbname>, <dbname>, Vault, FogBuz, etc.) to a predetermined size (say 100 or so MB). Then set them to grow by 50 to 100 MB. If traffic increases considerably and lots more data starts getting stored, then increasing the size of the 'gulps' SQL Server uses when it adds new file space will be a good idea.
- Database Backups.** Database backups are currently only being done for the <dbname> database. (Nothing is being done for <dbname>, Vault, or FogBuz.) Furthermore, backups are just being appended to existing backups – meaning that every backup made currently exists in the backup set. While this is good from a recoverability standpoint, it means that as the <dbname> database becomes larger (it's currently < 10MB in size), the backups will become larger and larger - eventually becoming completely unwieldy as this database approaches a size measurable in 100s of MBs.

The Solution: My suggestion would be for me to just meet with you for a few minutes, figure out your backup needs for all databases currently on the box, and then implement a custom backup strategy. Doing so won't be a very time-consuming effort (a matter of a couple hours – tops (mostly just me assessing your needs)) and will help you sleep better at night – knowing that your data is being backed up to meet your needs.

As part of this approach, I also recommend running the occasional test against random backups to ensure that they are viable – failure to do this frequently results in 'nightmare' scenarios where organizations find out that all of their backup efforts have been in vain (the nightmare aspect comes from finding this out when they go to do recovery from a disaster).

- Non Normalized Entities.** Normalization, in effect, defines the process of storing data for easier updates and extensibility (often at a slight cost to the speed of querying data). Failure to properly normalize items can lead to serious problems with extensibility later on.

The Problem: A couple of, mostly, 'dynamic design' related tables look like they are not normalized to typical standards (such as CustomColorsAndImages, ScreenStrings, and EmailTemplates) – but I don't think they're much of an issue as they are just being used as tabular data stored for application functionality – NOT for storing data which will be queried/sliced/and diced, etc.

The Surveys table in both databases, however, does worry me a bit. It too could just be considered as a serialization mechanism for application functionality (around surveys), but treating this table in such a fashion will lead to a certain degree of 'brittleness' in associated application logic and changes.

The Solution: My thought on the issue of the Survey tables is that we should probably dialogue about them to a certain degree. What you've got obviously works – and there's nothing truly wrong with it. From a normalized perspective, it could be done differently –in a fashion that would allow greater flexibility (but at the cost of increasing complexity). The biggest factor in evaluating this is the degree to which the structure of this table needs to change, and how often evaluating data from this table presents a problem. (As a rule of thumb, a quick way to 'spot' problems with normalization in many databases, is to look at the number of comments surrounding stored procedures that access data in non-normalized tables – it usually has to explain in depth how data is stored, what is needed to get data of flavor x, what is needed to get to data of flavor y, etc.)

As part of the discussion surrounding this table, I'm sending along a script that hopefully shows an alternative way of organizing survey data – which you can evaluate as a possibility. (It's a bit more complex, will turn some existing logic somewhat on it's head, and won't make everything perfect – but will allow for greater ease of extensibility.)

Important

The following findings are important – and should all be taken under advisement. My recommendation is that each of these issues be evaluated and addressed – though there is no major urgency involved.

- Failure to use OUTPUT and RETURN parameters in Stored Procedures.** SQL Server provides 3 parameter directions for stored procedures: INPUT, OUTPUT and INPUT/OUTPUT. It also provides a special RETURN parameter which is explicitly intended to return information about the success or failure of the stored procedure (many developers will attempt to overload this RETURN parameter as a 'short-cut,' but that is a major 'no-no' though SQL Server has no mechanism to enforce this). OUTPUT parameters and RETURN parameters are an excellent, and VERY light-weight, mechanism for returning status information from stored procedures, as well as row counts, new object ids, etc.

The Problem: Stored Procedures in <dbname> and <dbname> do not make use, at all, of these standard interfaces for exchanging data. Instead a common strategy seems to be to SELECT a 'message' on the 'way out' of the stored procedure (in the form of a 1 or a 0) – effectively hijacking SQL Server's Tabular Data Stream – designed as a way to return BULKY result sets.

Mitigating Factors: This isn't a major issue – just a departure from a best practice (and frankly, one that I see all over). None of the code using this SELECT 1|0 paradigm is broken – and works fine – in fact it shows a great degree of care and concern for ensuring that calling code knows how the requested operation concluded. Furthermore, .NET code allows for the ExecuteScalar (which I believe is a culprit in terms of helping encourage this paradigm), which helps mitigate much of the performance concern involved with this problem. The only issue is that it skirts the mechanism explicitly provided by SQL Server to get this information back. It also

tends to lead to more tightly coupled interfaces (where results are output in result sets) than the use of the standard interface mechanism provided by SQL Server which are designed to lead to better coupling.

The Solution: This solution is a tough one. The biggest issue with changing the current code is that the code is part of an interface between ADO.NET and SQL Server – so a change of the existing code would require changing code in both ‘tiers’ – which would also cause a lot of testing. Given how entrenched this paradigm is in the current code I’d suggest leaving it in place. But I’d strongly recommend discontinuing the practice for future code.

If you need an example of how to use RETURN or OUTPUT parameters, don’t hesitate to let me know – I’ll gladly whip some up free of charge.

- **Temp DB Fragmentation.** SQL Server uses the tempdb to store temporary tables that it generates during the processing of large and complex queries (such as reports). These temporary tables, called work tables, can take up a considerable amount of disk if the data being manipulated is large, or being aggressively processed. If SQL Server needs more room to complete operations, the tempdb will grow according to its preset growth settings in order to accommodate user queries.

The Problem: The tempdb is also highly susceptible to fragmentation when left to auto-grow at default levels.

Mitigating Factors: Again, there really currently isn’t enough data to worry about this.

The Solution: Another simple solution – just bump up the size of the DB to 2GB, and set it to grow in 250MB chunks. This is a simple operation and will keep the tempdb in good standing for a very long time.

- **Log File Sizes.** SQL Server uses log files for two primary purposes: to keep tabs on transactional operations as they transpire, and to provide a record of changes for databases that wish to track them (so that the database can be rolled back to a specific point in time should data corruption or a hideous mistake be detected.)

The Problem: A general rule of thumb is to keep log files at ¼ the size of the database files that they accompany. This prevents them from getting overly fragmented when large operations occur that require large amounts of log file space. Currently, most of the databases on DEDICATED just have default sizes, and have been ‘cobbled’ together by various ‘growth spurts.’

Mitigating Factors: There really isn’t enough data to really worry about this too much – currently. But as more and more data gets stored, this can and will become a concern.

The Solution: This is a very simple fix, and can be done in a matter of minutes – we just need to direct the settings for each database to adjust the size of the log file, and set it to grow at a rate where it won’t fragment.

- **Missing Clustered Indexes.** Tables without clustered indexes are known as Heap Tables, and can be VERY performant for storing oodles and oodles of INSERTS quickly, but carry a huge number of sinister penalties – one of which is that they are very difficult to keep defragmented.

The Problem: <dbname>..Accounts, <dbname>..Surveys, and <dbname>..FeedbackCategories are all missing clustered indexes. Not having a clustered index on the <dbname>..Surveys table will allow it to become heavily fragmented – and if lots of data (rows) get stored in this table, it will be difficult to defragment.

Mitigating Factors: <dbname>..FeedbackCategories isn’t likely to grow more, and <dbname>..Accounts won’t likely grow to millions of rows (though I’m sure that would be a nice problem to have).

The Solution: I would strongly recommend adding a clustered index to all of these tables. Doing so is very simple, and is something I could add very easily.

Performance Considerations

While I only performed an 'overall' audit, I did notice several key things about performance:

- **Statistics:** A perusal of the statistics shows that aren't being updated regularly. SQL Server uses complex statistics that it will create automatically if needed, to help ensure index precision and accuracy – better statistics means better query and index choices, which makes for faster queries with less CPU, and disk overhead. While SQL Server will modify these statistics by default – it tends to wait a little too long between updates (most likely out of a concern for negatively impacting performance while the statistics are being generated). A regularly scheduled job can painlessly and effectively update statistics during non-peak hours. I'll add the creation of this job to my punch-list of issues to address in the future.
- **Indexing Concerns.** SQL Server makes use of indexes to more quickly find data - much as the index in the back of a book helps you find data you are looking for. The better the index, the more likely that SQL Server will use the index. If SQL Server doesn't use the index, then it has to scan for the data – a very expensive operation (similar to scanning a book for quote or section you know is there but doesn't appear in the index). It is also possible to place information that satisfies individual queries entirely into composite, or covering indexes – meaning that some core queries can be sped up considerably.

The Problem: Several key stored procedure and function queries are missing indexes that would make them much more performant. There are also some composite indexes that don't match some commonly used queries – if these indexes aren't beneficial, they are actually imposing extra overhead (in terms of maintenance and slowed INSERT/UPDATE operations) and should be removed. Also, while most developers and SQL Server users have a good idea of the benefits of indexes, properly tuning them can be some what of a 'black art' as changes to indexes to accommodate one query can have negative impact on other queries (what I like to call the 'whack-a-mole effect'). In other words, index tuning can be a difficult and tricky operation – even for experienced users.

Mitigating Factors: Currently there's not a lot of data to scan – so scanning tables row by row will not be terribly expensive – but as tables become loaded with tens and hundreds of thousands of rows, this extra effort will take its toll on the CPU, memory, and hard drive. (In the past I've saved a few companies from having to spend \$30,000 to \$40,000 on 'better' hardware by merely taking a few hours to tune queries to make better use of indexes – the results can be VERY dramatic – so having these issues out of the way early on will make a lot of sense.)

The Solution: I've already compiled a list of areas to evaluate – and will just need a few hours to pull down local copies of your databases upon which I can tune and test. I'd also like to coordinate a time with you when the server is under decent load, to trace activity on the box and then analyze it and tune things for performance. Three to Four hours should be all I need to do this – and it can be done now, or over the ensuing months as desired.

- **Cursors.** A handful of functions and stored procedures currently use cursors. Cursors are procedural constructs by nature which are generally used by programmers in lieu of set-based operations (which can usually accomplish the same functionality in a much more performant manner - but at a cost of increased complexity in the mind of most developers). It is very common for non-database-savvy developers to severely abuse cursors and impose major performance problems. Because of this abuse, it is also common for database 'snobs' to assume the use of any cursor is bad (which isn't true). Happily, I didn't see any abuse of cursors in the existing code – though it's likely that some of the existing stored procedures using cursors can be rewritten to avoid using them. Doing so would take a good deal of time though, and should only be considered if performance on the box becomes an issue. That said, the existing cursors could be quickly and easily optimized for better performance though some slight modifications.
SQL Server allows for a number of powerful options during cursor creation – most of which aren't

needed in the majority of cases/uses, and add measurable overhead. Failure to be exact and explicit with these options results in default options that sadly come at the price of adding unnecessary and expensive overhead. To correct this, simply make existing CURSORS explicitly LOCAL, READ ONLY, and FAST FORWARD. This will allow them to be created without the need for expensive pointers needed for rewinding or modification – making them considerably faster and more performant. To implement this change, simply change the extant

```
DECLARE <cursorName> CURSOR FOR SELECT ...
```

syntax to

```
DECLARE <cursorName> LOCAL READ_ONLY FAST_FORWARD FOR SELECT ...
```

This will explicitly set light-weight cursor options, and will also change the scope of the cursor to the current process, meaning that other threads won't need to have pointers created to this cursor as well.

This is a fix that I highly recommend, as it is easy to implement, and will provide immediate benefit (not that it's needed currently, but as more and more load is placed on the server, this will be nice to have out of the way). As such, here is a list of all udfs and stored procedures currently using cursors:

- <dbname>..**udf_is_GetAllGroupsOfAdmin (UDF)**
- <dbname>..**is_UsersInGroupTree (UDF)**
- <dbname>..**isGetUserLineageFunction (UDF)**
- <dbname>..**isAddSeedUsersToGroup (Stored Procedure)**
- **Rebuilding of indexes.** As data in a table is INSERTED, UPDATED, and DELETED indexes that track information about the data can frequently become fragmented – which slows performance (both for INSERTS/UPDATES and SELECTs). Indexes can be defragmented, which can have a DRAMATIC impact on performance. It is therefore recommended that key indexes be fragmented as frequently as needed – even daily if necessary on heavily used tables. None of the tables or databases on DEDICATED currently have enough rows or data in them to become fragmented – though this is something you'll want to keep an eye on more and more data starts getting logged. I've got a number of scripting options that can be put in place to help rebuild key indexes on a regular basis to keep data performant as the system becomes more and more used.

Scalability Considerations

Info about how to prepare for coming growth...

- **Multiple Data Files.** SQL Server allows for database data to be stored in multiple files. Each database requires a primary data file (.mdf) and a log file (.ldf), but for very active databases, the possibility of using multiple log files (.ldf files) exists, and can even allow for log data to 'straddle' multiple drives. The same goes for data files – and the notion of 'straddling' can provide considerable performance improvements as well as allow large databases to span data across multiple drives. A common strategy with larger databases is to break logical operations across multiple data files/drives as well. For example, when a query is run against a table with millions of rows if an index is used, SQL Server can traverse the index on one file, and begin fetching results from another file – providing a considerable performance boost. The databases on DEDICATED are really way too small to consider this type of approach, but if they begin seeing action on the order of thousands of surveys per week, then it would be a good idea to consider using multiple data and log files. My recommendation is to do nothing about this now, but remember it as an option when things begin to really get busy.

- **Server Setup.** DEDICATED is a fairly powerful box – with more than adequate resources for the current load. As load increases, however, there may be resource contention between SQL Server and IIS (the application server for ASP.NET). Both applications are notorious for their in-ability to ‘play nicely’ on a single box under load. Of course, they can be directed to play nicely if given the proper instructions (mostly in the form of dedicating RAM – both applications make heavy usage of RAM/caching to avoid expensive operations). And, should this ever become a problem, DEDICATED currently only has 1GB of RAM – another Gigabyte or three would be fairly cheap – and will do a great job of extending SQL Server performance as more and more data and load become common on the box.

FYI or Further Resources

Things you’re probably aware of, but just in case:

- **ELMAH_Error Table sizes.** Currently, the ELMAH_Errors table accounts for 98% of the total space used in the <dbname>database, and 76% of the total space used in the <dbname>databases. (i.e. of the roughly 80MB of data in database <dbname> – 98% of that is ELMAH data.)
Mitigating Factors: Most likely this just represents lots of development/testing. This space can also be recovered very easily by simply deleting it.
The Solution: May want to look into scripting a weekly job that deletes data older than a few weeks/days/etc old if this information will continue to show up/populate the table on a regular basis. (Doing this would be very easy and I can help out if desired.)
- **SQL Server Logs and Crashes.** Things.
- **Outdated Code.** Looks like the following sproc references non-existent objects, and can therefore be dropped:
 - <dbname>.dbo.isGetGrouIDForPortal
- **Non Standardized Naming Conventions.** In many ways this observation could be considered a ‘nit-pick’ – but there are a number of apparent naming conventions being used in each of the key databases. This is most obvious in the naming of User Defined Functions: some are names *isGetxxxx*, others *is_xxxx*, and others *udf_xxxx*. Unless there is a specific reason or purpose for naming these objects like this, you may want to consider standardizing on a single convention in the future. Having a single convention serves to improve readability, and reduces ‘mental noise’ which can contribute to better development time and fewer errors.
- <dbname>..**Questions.** The ordinal number of each question is included in the question text. This is technically a violation of 1st normal form (database-ese for saying that it repeats data (the ordinal number is also the id of the question)), and will make re-ordering of the questions a bit more difficult. (i.e. if you were to ever swap questions #5 and #10 – you’d have to change the text in addition to changing their placement). This may or may not be an issue depending upon how often question order changes – but removing the numbers from the question text won’t hurt, and can only benefit things (unless you are somehow relying upon this number for business logic). If this number is needed for presentation, I would suggest a simple modification of the query that retrieves the data: `SELECT CAST(QuestionNumber as varchar(3)) + ‘. ’ + Question FROM dbo.Questions...` etc. This, of course, is some what of a nit-pick and should be treated as such.