Global Information Assurance Certification Paper

GIAC Security Essentials Certification (GSEC)
Practical Assignment – Version 1.4b Option 1

Frank Ress

SQL Server Email – vulnerability issues and prevention strategies

Contents

I. Abstract.

One of the optional features available to users of the Microsoft SQL Server® database is the ability to send and receive Email messages programmatically from the database. Email is often an ideal way to send administrative alerts to system staff and/or end users when unusual conditions are detected, to distribute various kinds of routine reports, etc. Many modern databases share this ability to send Email, and SQL Server Email capabilities have been available at least as early as version 6.0. The SQL Server implementation is somewhat unique among relational databases, however, in its ability to accept and process incoming as well as outgoing messages. This aspect of the Email feature in SQL Server – the ability to send a message to the database and have it automatically execute a query and reply to the sender – presents a potential vulnerability that could easily be overlooked.

This paper will explore some of the ways this feature could be used by both legitimate users and intruders. Installation and configuration of the utility will be briefly described in enough detail to support the ensuing discussion of the vulnerability. Finally, a number of strategies will be suggested that could be used to minimize the vulnerabilities exposed by use of this feature.

II. SQL Mail – How it works.

SQL Mail, as the SQL Server Email utility is commonly referred to, is capable of accepting an incoming message that contains a 'query' in the message body. It will execute the query and return the results to the sender.

There are any number of legitimate uses for such a capability.

- A database administrator could query the database remotely, to check on the availability of various resources, the status of scheduled processes, or to check for errors or alerts.
- Employees could inquire on the status of their benefits.
- Sales staff could check current prices and inventory levels.
- Customers could check the status of their orders.

Almost any query that could be satisfied with a standard database report or form could be delivered in an Email message. Furthermore, the flexibility of such an approach is excellent. Changing the information delivered is as simple as changing the query submitted. The only limitation in the basic implementation of this feature for SQL Server is that the requesting message must contain a single SQL statement. Of course, even this limitation could be removed through relatively simple custom programming.

The three central objectives of information security are confidentiality, integrity, and accessibility. From an accessibility perspective, the ability to simply send an Email message to the database and receive the response in a reply provides a very convenient delivery system. Email is available to nearly every computer user, and for most users Email is the application with which they're most skilled.

Given the Email address used by the database and the ability to construct valid SQL, virtually anyone could use SQL Mail to interrogate a database. Even users unfamiliar with SQL could be supplied with standard queries. Furthermore, the specific Email client used is irrelevant, since gateways exist between all major Email systems and the Internet. SQL Mail queries could be submitted from any type of client system, anywhere on the Internet, without any additional software installation. Aside from Email applications, only web browsers approach this degree of ubiquity and interoperability.

III. The vulnerability.

So it's possible to configure a SQL Mail enabled database to respond to queries. Since the query is run by the database itself, by default it runs with the privileges of the account used by the underlying service. In order to support SQL Mail, the MSSQLServer service must be run from a domain account with local admin privileges (at minimum). Thus the query is executed with fairly extensive privileges.

Using a basic configuration, the database will respond to any query, from any requestor. That's a little generous, to say the least. In almost any database system, we'd like to limit the information that can be retrieved by the average user (protecting system information, if nothing else).

But it's actually a little more complicated than that. Most of Microsoft's descriptions of this feature mention SQL Server processing a 'query' delivered in an Email message. But what exactly is a 'query'? One might be tempted to assume that 'query' implies a read-only request.

In fact, the database will respond to any valid SQL statement. So, in addition to queries, it's possible to process insert, update, and delete commands, create, modify or drop database objects (such as tables), etc. SQL Server also a rich set of built-in SQL routines, known as system stored procedures and extended stored procedures, that could be invoked through an Email message.

System stored procedures are used to perform operations within the database (as opposed to interacting with external processes, for example). They're written and stored in the database in T-SQL, the Microsoft dialect of SQL. Since they're stored as source code, rather than in a compiled form, they could conceivably be modified through an email-delivered payload (assuming sufficient privilege).

Extended stored procedures are able to execute external functions by mapping through DLLs. For example, there's an xp_cmdshell routine that can accept and execute Windows command-line functions. For an excellent discussion of the kinds of operations that can be performed through these built-ins and how they might be abused by an intruder, see McDonald [1].

On the basis of our other two information security goals – confidentiality and integrity – a trivial SQL Mail implimentation leaves much to be desired.

The simple reply-to-sender model makes no attempt to verify the identity of the requestor (authentication) or to check whether the requestor has rights for the information requested (authorization). Unless mechanisms to implement authentication and authorization are included in SQL Mail query/reply systems, confidentiality and integrity are difficult to guarantee.

What are the barriers to abuse of the minimal SQL Mail implementation by intruders? In essence, the only default safeguards are 1) lack of awareness – on the part of the intruder - of the vulnerability itself, 2) lack of awareness of the existence of a database that has implemented the facility, and is thus vulnerable, 3) lack of awareness of the name of the account to which SQL Mail can be addressed and 4) demotion of the default privilege level of the process. Essentially, security relies on ignorance - always a dangerous assumption - and careful attention to the privilege of the Email handler by the database administrator. If the first three hurdles are overcome, the problem for a potential intruder becomes one of privilege elevation, at worst.

Assuming would-be intruders are aware of the vulnerability itself, how great a safeguard do the other 2 awareness barriers provide?

From an intruder's perspective, if she/he could determine the proper account, the target Email system would route the penetration message to the appropriate database server. So if the intruder has knowledge of the SQL Mail account, or can guess the name of the account, a simple Email message (one that could be guaranteed to run for even the least privileged account) would be enough to determine whether incoming SQL Mail has been implemented in the target domain! Once the intruder has gathered this intelligence, she/he's ready to tackle the privilege issue – assuming there is one!

There are some logical candidate names that might be chosen for the SQL Mail account. Remember, if the utility is to have legitimate value, it would tend to be given a name that's relatively descriptive, to make it easy to remember. There are also more-or-less well known names for this account, and an intruder aware of the vulnerability would probably know these, as well. These would include standard domain admin accounts like 'administrator', SQLAdmin (or SQLAdmin1, SQLAdmin2… the names used in course materials for SQL Server administration from Microsoft [7]), and a few other logical candidates (SQL, SQLServer,

SQLOperator, etc.) Remember that it only takes an Email message to test each possibility. Also, sites that wish to implement this utility for multiple database instances might need a separate account for each one (to allow messages to be routed to the proper server), further increasing the probability of a hit.

Still, how likely is the interest of potential intruders in SQL databases? We might make an inference from another technique used to compromise SQL databases, SQL injection [1,10].

SQL injection relies on trial-and-error modifications to parameters passed to a database via the URL on a web page. By modifying the parameters, the intruder is eventually able to discern data structures and retrieve data. To be effective, the intruder has to be fairly fluent in relational database concepts and SQL (as well as to find a database vulnerable to SQL injection).

SQL injection can be relatively slow and labor intensive. The intruder has to invest some time to compromise even a vulnerable database, and even after some degree of penetration has been achieved, exploiting the intrusion tends to be tedious, since queries still have to passed through URLs. By comparison, a mail-enabled SQL Server database, where SQL can be passed in clear text is easier to use.

One advantage of SQL injection over a SQL Mail attack is that public web pages offer a clue where a vulnerable database might be found. By observing displayed pages and inspecting URLs, web pages that access SQL databases can be detected. Tools that could be used to perform reconnaissance also exist, such as Wpoison [1]. Note, however, that the same clues could be used to perform reconnaissance for SQL Mail attacks.

In fact, SQL Mail and SQL injection may be very complimentary intrusion techniques. Intruders familiar with one would probably find the other useful and easy to learn. The two techniques could be combined in attacks on any database that was vulnerable to both, or to use one vulnerability to implement the other.

IV. Standard Setup.

In order to discuss techniques to harden SQL Mail to prevent intrusions, it helps to know a little more about how it's configured.

There are a number of excellent and detailed references that discuss the configuration of SQL Mail in great detail [2,3,4,6,8]. The following brief description of the configuration process is not intended to provide an alternative to these resources. What's described here is the basic information necessary to understand and explore the security implications of the various configuration options. Some or all of these references should be consulted before pursuing

use of SQL Mail in your environment, in order to more thoroughly understand how SQL Mail is installed and used.

Note, also, this discussion reflects the current production version of SQL Server (SQL Server 2000), rather than any of the earlier versions of the database that have supported SQL Mail. Information on specifics of earlier versions can be found in many of the references cited and other resources on the Microsoft website.

It's also worth mentioning that both of the SQL Server services (the database service and the SQL Agent service) can make use of Email. Server mail, as noted earlier, is referred to as 'SQL Mail', and the agent mail as 'SQL Agent mail'. SQL Agent mail is only used for outgoing mail. It doesn't process incoming messages, and we'll largely ignore it for the remainder of this discussion.

Briefly, preparation to use SQL Mail requires at least the following steps:

- SQL Server installations create 2 Windows services (the MSSQLServer database service and the SQLSERVERAgent service). By default, the standard installation will create and configure these services to run under the local administrator account. While sufficient for most database operations, this will not support SQL Mail. To use SQL Mail, the relevant service must be run under a Windows domain account with local administrator privileges. Create such an account if you're not going to use an existing account.
- Create an Email account on the mail server for the domain account just described.
- SQL Mail requires an Outlook 2000 or Outlook 2002 mail client to communicate with your Email server (typically Exchange). Log onto the Windows server using the same account that will be used to run the SQL Server service(s), and configure Outlook as the default mail client (as opposed to Outlook Express or some other Email client).
- Create a mail profile on the database server that identifies the mail server, etc. that's appropriate for your environment.
- Open the Outlook client and verify that Email can be sent and received from the domain account on the server that hosts the database.
- Use the SQL Server Enterprise Manager utility on the server to associate the new mail profile with the server service and/or the SQL Agent service[1].

[1] It's worth noting that once you configure a mail profile for either the server service or SQL Agent, it's not possible to 'undo' the profile from Enterprise Manager. You can change to another profile, but you can't return to a null profile. To completely remove a mail profile once you've defined one, you'll have to edit the registry.

- While still in SQL Server Enterprise Manager, create a job that runs the sp_processmail [5] package. Schedule it to run at a suitable interval (e.g. every 5 minutes) to process incoming Email messages.

The sp_processmail package isn't the only option available for processing incoming mail. It simply provides the basics needed for a rather bare-bones utility, and serves as a time-saver to get inbound Email processing up and running. It mostly calls a number of extended stored procedures (xp_openmail, xp_findnextmsg, xp_readmail, xp_sendmail, xp_deletemsg, etc.) to process a single message, expected to contain a single query.

Furthermore, although this doesn't appear to be documented anywhere, the procedure as delivered will process requests with a default privilege of 'guest'. So, by default, sp_processmail will reduce the privilege of the mail handler to a fairly conservative level. Be careful that you don't simply remove this default value if 'guest' is too restrictive. Without any value for the @set_user parameter for sp_processmail, it will run every job with the full privileges of the server.

V. Protective Measures.

Before we even begin to consider protection, note that SQL Mail is not part of the default SQL Server installation. It must be configured manually, after the database itself is installed. Therefore the SQL Mail vulnerability is not one that need be immediately addressed after every database installation.

However, if you do set up SQL Mail to process incoming messages, there are many techniques that can be used to increase its security. The following list isn't exhaustive – there's always room for new ideas – but will certainly be an improvement over a basic SQL Mail installation.

1) **Don't use SQL Mail at all.** This may sound like a ridiculous suggestion. It's a classic "throwing the baby out with the bath water approach". But as we noted earlier, proper attention to security must consider confidentiality, integrity, and availability. Functionality nearly always conflicts with security to a greater or lesser extent. You have to consider the value of the feature (accessibility) and weigh it against the associated risk (integrity and confidentiality). The SQL Server Security Checklist found at [11] simply recommends not using this feature, without further discussion. Do you have a genuine need for this, or is it just a convenience you could live without? Certainly consider some of the following suggestions to mitigate the risk before making this decision, but SQL Mail may not be worth the risk at the end of the day. Avoid the temptation to add a service because 'it might be useful someday'. In our environment, we expect that we will use outgoing mail at some point, and we will configure it when it's needed. But we highly doubt the need for incoming SQL Mail at any point in the future.

2) **Don't accept incoming messages.** SQL Mail requires some sort of 'reader' process to handle incoming mail. Microsoft supplies the sp_processmail package as one way to implement the reader process (as noted, you could also code your own). But you're not obligated to do anything with these messages, and you may only need the outgoing mail features. If this is the case, don't set up an incoming mail handler of any sort.

3) **Screen incoming messages.** Watch for suspicious Email traffic that might be directed toward the SQL Mail service. Look for messages that contain strings like 'SELECT'…'FROM' to detect messages with SQL bodies. Look for messages addressed to the SQL Mail account, or addressed to logical candidate names for the account. Many mail servers allow traffic for particular accounts to be copied and stored, in addition to being delivered. Direct a copy of all messages going to or coming from the SQL Mail account to an administrator or operator account, so that traffic can be monitored with a minimum lag from the time it's delivered to the database.

4) **Intercept (block) incoming messages.** Configure your mail server to reject (or drop, or quarantine) mail addressed to the SQL Mail account from external users (maybe even some internal users), or both, as your needs might demand. Your mail server(s) have to route mail to SQL Server before either legitimate users or intruders can take advantage of the service. Don't deliver anything that's not wanted. Combine this with (2) above, or some of the suggestions that follow, to build up a defense-in-depth strategy.

5) **Filter incoming messages.** Exercise some selectivity in the messages that get processed. There are any number of ways to evaluate the incoming message to decide whether to respond to the request.

   a. If you're using the sp_processmail package, consider using the @subject [5] parameter when invoking it. The package will only process messages that match the subject you specify in the call to sp_processmail. Although the primary purpose for this parameter is to distribute the workload over multiple mail handlers, it could be used for a sort of "poor man's passphrase" system. An intruder would need to know the passphrase to have his/her message processed.

   b. If using sp_processmail, you could use the @set_user [5] parameter with an argument of @originator to accept messages only from senders with a valid SQL Server account. (Consider the possibility that the sender address could be spoofed when you evaluate the reliability of such an approach.)

   c. If you code your own routine to process incoming mail (instead of using sp_processmail), you could extend the passphrase technique to include a password/passphrase in any part of the message. The @subject

approach used in the preceding item is a sort of binary filter; you either know the passphrase and your request is processed, or you don't know it and the request is rejected. If you code your own routine, you could scan for a passphrase in the message body as well as the subject. You could also implement a number of different passphrases, each of which keys a different processing behavior/privilege.

d. Another possibility open to you if you develop your own mail processor would be to use certificates or encryption. Possible encryption protocols that could be used include Privacy Enhanced E-Mail (PEM) or Secure Multipurpose Internet Mail Extensions (S/MIME). Incoming messages that lacked the proper certificate or that were unencrypted (or improperly encrypted) would be rejected. This would require a significant programming effort, but if, for example, a consulting organization was using SQL Mail to perform remote SQL Server administration for several clients, the cost might be justified.

6) **Reduce the privilege of the mail handler.** As already noted, the E-mail processing routine will run with the privilege of the server process itself, and sp_processmail will demote the privilege to that of a 'guest' user. Even this may be too generous if you have no other barriers in place to screen potential intruders. The principle of least privilege should always be applied to an incoming mail processor.

   a. If you're using sp_processmail, another advantage of the @set_user=@originator parameter is that the privilege of the process is automatically aligned with the matching database user account (in addition to the previously noted behavior of rejecting requests for requestors who don't have a database account).

   b. If you customize sp_processmail, or if you create your own mail handler, avoid the temptation to start with wide-open security, with the best of intentions for making improvements later. At minimum, follow the sp_processmail model for your base security. Enhance it by filtering by statement type (allow only SELECT statements, not UPDATES or DELETES, for example), or by requestor (only allow employees access to their own records).

7) **Secure your database.** Implement general database hardening [2,11], apply operating system and SQL Server patches, etc. Everything you do to improve general database security will improve the environment for SQL Mail.

VI. Intrusion Detection.

Regardless of your efforts to prevent intrusions, you have to consider the possibility that your installation may be compromised. If that occurred, how would you detect the penetration?

When it comes to detection, I'm always reminded first of one of the lessons from Clifford Stoll's classic book "The Cuckoo's Egg" [9]. His search for an intruder began because of a minor discrepancy between a standard accounting utility and one that had been custom written. The intruder knew about the standard package, and made an effort to cover his tracks. But he was unaware of the custom utility, and comparison of the two exposed his activities. Standard features and utilities are more convenient and easy to use than custom coding, but they're also well known to intruders, and are more vulnerable to tampering for that reason.

1) **Retain processed messages.** By default, sp_processmail will delete messages after they've been processed. While this might save space, it removes a potential audit trail. Not only do you need some way to determine what's happened when you suspect an intrusion, you need to have some way of knowing what your normal activity is. Checking the messages processed by SQL Server is probably one of the most convenient ways of accomplishing both of these objectives. It's also an obvious possibility, which an intruder would check and attempt to sanitize. Saving copies on your mailserver, as already mentioned, is probably less susceptible to tampering by an intruder.
2) **Log everything, everywhere.** If there's one thing Microsoft products do well, it's to provide plenty of checkboxes for 'Create Log' and a field to specify the log name and destination (or to use one of the system logs). Use them. It's hard to think of much downside to having a record of operations on your database. (Aside from the overhead it takes to create the logs and diskspace it takes to save them. So buy another disk for audit information if you need to – it's cheaper than losing evidence of an intrusion because you didn't have room to save the log.) In addition, SQL Server takes logging to another level with the SQL Profiler audit tool. Understanding the Profiler is an absolute must for any database administrator. It's capable of recording things like the creation of new database accounts, changes to user rights, database operations like backups, etc. You can define your own event 'producers' if you'd like. It can direct trace information to a GUI management interface, flat files, tables, or event logs. Don't be too complacent with the standard logs, though. Remember the Cuckoo's Egg. Building application logs is probably one of the easiest forms of custom recordkeeping you can add. Consider things like adding your own code to critical stored procedures (like sp_processmail), that will record each call to the routine, and when, and by whom.
3) **Create a 'tripwire'.** Tripwire is a tool that can generate a 'fingerprint' of a file using a one-way hashing algorithm. It can be used to detect tampering with critical files on a server by comparing the hashes generated at different times. If the fingerprints match, you know nobody has tampered with the files. Unfortunately, database files are large container files that are constantly changing, so the standard file-based tripwire product isn't very useful to

detect database tampering. Within the database, however, it might be possible to identify objects like stored procedures that should be fairly static. It would be easy to write SQL scripts that could list these objects, to see if the numbers of these objects change over time[2]. A more ambitious undertaking might be to code your own hash algorithm and 'fingerprint' the objects, to develop a 'tripwire' to run within the database.

VII. Conclusion.

The feature of SQL Mail that allows database requests to be submitted via E-mail is one that should be used with a great deal of caution. The primary advantage of this feature – the fact that it is so easy to widely deploy as an application interface – is also the source of its greatest weakness.

It's useful because E-mail tends to be one of the transport mechanisms that's commonly allowed through firewalls and other network filtering devices. The vulnerability this presents is that virtually all protective measures have to be deployed within the database itself. If you use this tool, you're almost forced to deal with potential attackers when they're at the gate.

The one tool that might help you protect your database at a distance is a mailserver that's capable of screening and filtering E-mail. If at all possible, this should be a prominent part of your defense-in-depth strategy, if you're accepting inbound SQL Mail, sitting between your firewall and your database server. The mailserver is the first place you have an opportunity to detect reconnaissance probes and intrusion attempts. Ideally, that's the place that you will stop attackers, but if not, the last line of defense is the database server itself.

Hopefully, this paper has given you a good idea how this feature can be used and how it operates, what the inherent weaknesses are, how to begin building your defenses, and where to go to get additional information.

References

[1] McDonald, Stuart. "SQL Injection: Modes of attack, defence, and why it matters". (Apr 2003).

[2] Microsoft Corporation. "Microsoft SQL Server 2000 Books Online" v8.00.000. (SQL Server 2000 online documentation, distributed and installed with the database).

[2] Hint: Use osql, the command-line SQL tool, to execute your script from a Windows .bat file. You can run your .bat file on another server (use the Windows scheduler service), and the –S qualifier in your osql command will direct the SQL execution back to your database server. That way, an intruder would be less likely to notice your audit script, because it's run from another server.

[3] Microsoft Corporation. Microsoft Knowledge Base Article 263556; "INF: How to Configure SQL Mail" (May 2003). http://support.microsoft.com/default.aspx?scid=kb;EN-US;263556

[4] Microsoft Corporation. Microsoft Knowledge Base Article 311231; "INF: Frequently Asked Questions – SQL Server - SQL Mail" (May 2003). http://support.microsoft.com/default.aspx?scid=kb;EN-US;311231

[5] Microsoft Corporation. Microsoft Developer Network Transact-SQL Library, "sp_processmail" http://msdn.microsoft.com/library/default.asp?url=/library/en-us/tsqlref/ts_sp_pa-pz_0w4s.asp (2002).

[6] Microsoft Corporation. Microsoft Office Developer, SQL Server E-mail "An Introduction to SQL Mail and SQLAgentMail". http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnovba01/html/sqlservere-mail.asp (2002).

[7] Microsoft Corporation. Microsoft Official Curriculum. "Administering a Microsoft SQL Server 2000 Database." Course no. 2072A. (Oct. 2000). Module 5, pp 5-6

[8] Rankins, Ray; Jensen, Paul; Bertucci, Paul; et al. "Microsoft SQL Server 2000 Unleashed". (2002). SAMS Publishing, pp 1290-1314.

[9] Stoll, Clifford. "The Cuckoo's Egg: Tracking a Spy Through the Maze of Computer Espionage". (1989) Doubleday, 666 Fifth Avenue, New York, NY 10103.

[10] Strawmeyer, Mark. "Secure Your ASP.NET Application from a SQL Injection Attack". (Aug. 2003) http://www.developer.com/net/asp/article.php/2243461

[11] SQL Server security checklist http://www.sqlsecurity.com/DesktopDefault.aspx?tabindex=3&tabid=4