



# Danger From Below:

## The Untold Tale of Database Communication Protocol Vulnerabilities

---

---

*Written by:*  
**Amichai Shulman**  
Co-founder, CTO  
Imperva, Inc.

---

Prepared for  
BlackHat DC 2007

### Abstract

A new attack vector is being used on database servers. Attackers are taking advantage of vulnerabilities in the database communication protocols. These refer to the proprietary communication protocols created by database vendors to convey data and commands between database client software and database servers.

The database communication protocols are proprietary, and many pre-date the Internet. While developers strive for backwards compatibility to ease new product integration and solve compatibility issues between versions, this practice also fuels the fire for potential vulnerabilities. Until recently, researchers weren't focused on this class of vulnerabilities. For many software engineers and DBAs, the existence of potential issues with database communication protocols is relatively unknown.

This paper delves into the background of database communication protocol development and testing and explains how these vulnerabilities continue to proliferate. Based on extensive research and testing, this paper describes areas of vulnerability and presents potential methods for mitigating the risk associated with this new class of attacks

## A Brief History of Database Security

Until a few years ago, database server security assessment took into account two classes of remote attacks: infrastructure-level attacks and unauthorized access to database objects.

The first class includes infrastructure attacks aimed against low level components of the network stack (IP, TCP) and attacks (or plain simple unauthorized access) against basic services installed with the operating system that hosts the database server (FTP, Telnet, SNMP, etc.). These attacks were not database vendor specific and had actually little to do with the fact that a database is actually installed on a given machine. Common mitigation tactics for this type of attacks include the use of network firewalls and intrusion detection and prevention systems (IDS / IPS). Given their generic (i.e. non-database specific) nature and the relative maturity of mitigation solutions, database security officers have learned to disregard these attacks and leave them to be handled by general network security personnel.

The second class of attacks relates to unauthorized individuals using the standard SQL query language for accessing or changing information within the database, or even changing the structure of the database itself. This is a type of risk that is indeed inherent to the functionality of a database server. It too is not a risk that is vendor specific. All vendors use roughly the same object model (the relational model) and same basic query language (Structured Query Language – SQL). Traditional mitigation techniques rely on the built-in access control mechanisms within the database server. While not a perfect solution, these mechanisms provided reasonable protection against unauthorized access through the use of standard SQL queries and statements. Setting and monitoring the internal access controls in a database server quickly became the number one concern of database security personnel.

About six years ago, security researchers started to publish their findings regarding database vendor-specific vulnerabilities that can be exploited through the use of standard SQL query language to bypass some of the built-in security mechanisms of the database server. Attack types include buffer overflow (see [ 1 ],[ 2 ],[ 3 ],[ 4 ],[ 5 ],[ 6 ]), SQL injection through stored procedures (see [ 7 ],[ 8 ],[ 9 ],[ 10 ],[ 11 ]) and a multitude of access control bypass techniques (see[ 12 ],[ 13 ],[ 14 ]). The introduction of this new breed of vulnerabilities sent the database security officers into a never ending “patch” pursuit. Once a vulnerability became known a new software patch had to be applied to the database server machine. However, as it turns out, in all but the rarest cases these risks could have been mitigated using a combination of built-in security mechanisms, other database configurations and IDS / IPS solutions. Moreover, by keeping a reasonable audit trail of database activities attacks and violations could be detected and analyzed for the purpose of future remediation. Thus, while database security became a (mostly) reactive practice, solutions could be found within the existing security framework and to some extent proactive security practices are possible.

Approximately two years ago, a new breed of attacks started surfacing. Although, in retrospect, we began to see the first glimpses 6 or 7 years ago. Slightly above the long forgotten infrastructure level attacks and roughly an inch below the SQL level attacks, researchers uncovered vulnerabilities related to the high level (i.e. above TCP) network communication protocols. These vulnerabilities and attacks are tightly related to the use of a database server and are vendor specific. They operate below the SQL level apparent to the DBA, yet at the same time they allow a much higher level access to internal database mechanisms than infrastructure level attacks. In the past year, we have witnessed a dramatic increase in the number of vulnerabilities of this type.

The next sections describe the nature of the database network communication protocols and introduce the vulnerabilities related to these protocols. Then, we will review some examples and mitigation techniques.

## An Introduction to Database Communication Protocols

### *The origin of database communication protocols*

While the syntax and semantics of data access and management commands is mostly defined by a well known standard called ANSI SQL (last version was issued in 2006) other important aspects of the client-server interaction are not. These aspects include the method for creating a client session, conveying the commands from a client to a server, the method for returning data and status to a client, the structure of the returned data and the implementation of mechanisms such as cursors, prepared statements and transactions. Since all these details are required for the correct (not to say basic) functioning of a database server, we can only assume that the gap is filled by vendor specific technology.

Thus, each database vendor has devised a proprietary protocol consisting of a set of messages, interactions and semantics that would provide for these functions. The vendors usually chose to implement these protocols as an independent application messaging layer that can be transported on top of a multitude of transport level protocols such as TCP/IP, Named Pipes and even SNA and DECNET. This allows the vendors to maintain most of the implementing code platform and transport layer independent. It means for example that the code that implements the DRDA protocol on OS/390 platform (mainframe machine) is the exact same code that is used for a Microsoft Windows platform.

Traditionally, the vendors have chosen to proceed with a proprietary implementation and offer little or no public documentation. Examples include SQL\*NET from Oracle, TDS from Sybase, yet another strand of TDS from Microsoft and DRDA from IBM (the latter is to some extent an “open” protocol). By keeping the protocols proprietary and undocumented, each vendor effectively became the sole provider of basic client software (i.e. drivers). While there are some known exceptions to this statement (see [ 15 ],[ 16 ],[ 17 ]) they draw from very partial information supplied by the vendors under strict conditions.

### *Complexity brought to its max*

Most of the protocols include a number of semi-independent layers. The TDS protocol consists of two layers with approximately 10 different messages constituting the lower level and ten times that making the second layer. SQL\*NET and DB2 use a 3-layer protocol with tens of messages in the highest layer.

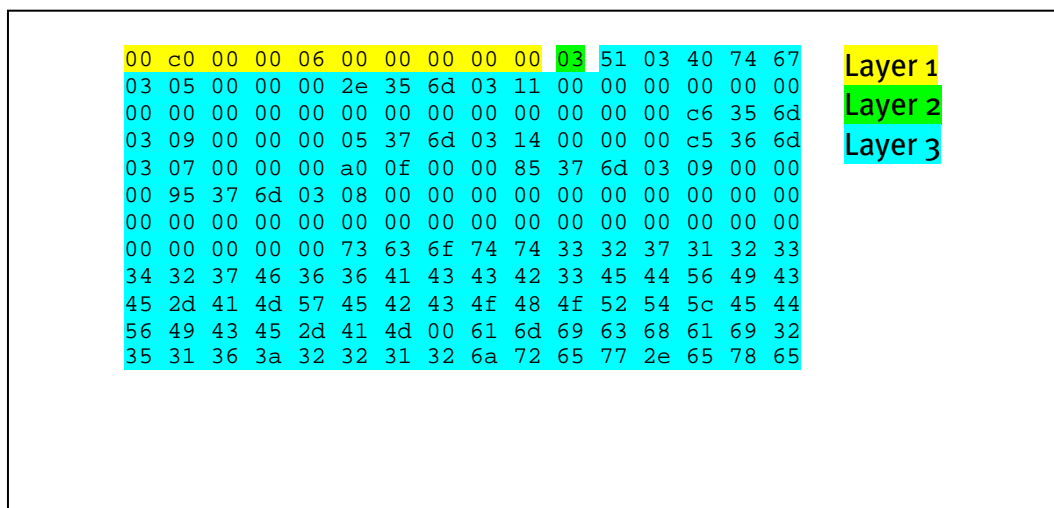


Figure 1: Sample layered message (Oracle)

The code implementing messages in the different layers is developed by different programmers groups not always in coordination. As a consequence, the assumed trust level between the various layers is sometimes misleading, and occasionally messages show information redundancy that is not always validated through all layers (e.g. an independent size field appears in various layers of the same message).

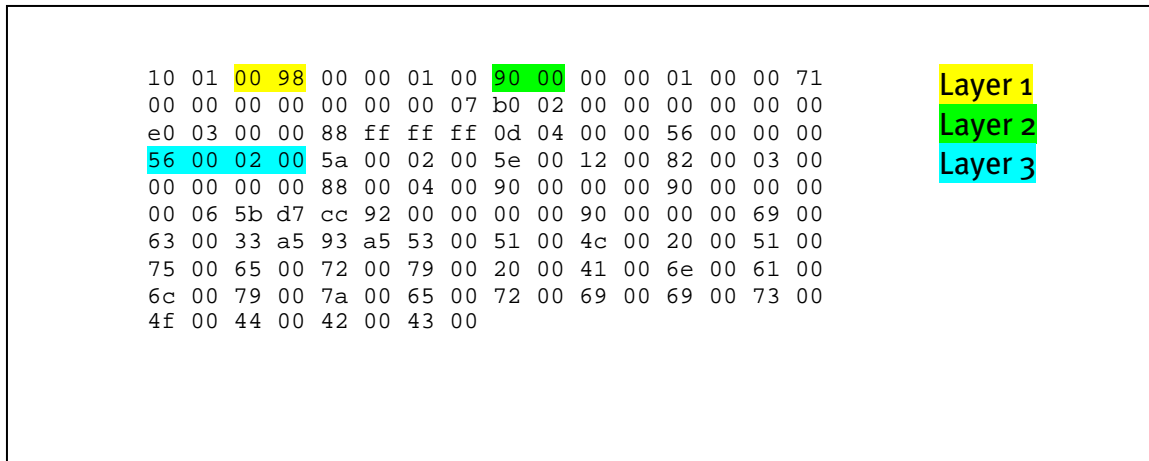


Figure 2: Redundant size information across layers (MS SQL Server)

Protocols also carry a long history of backwards compatibility. For example, an Oracle 8 client can communicate with an Oracle 10g R2 database server while an Oracle 10g client can communicate with an Oracle 8 server. This implies that some of the code that implements the protocol is more than 10 years old while at the same time it is rarely tested in real environments.

Complexity concerns goes even higher as protocols try to maintain the highest efficiency while conveying structured information between systems with different CPU architecture, different byte orders and different character representations. One apparent example is that for most of the protocols there is one layer of messages that has a preset data representation (mostly network order), while the Endianess (as well as the size of a word) of other layers is negotiable. Oracle SQL\*NET for example uses special message structures to eliminate multiple network transmissions of a value if it appears in more than one row in a result set. IBM’s DRDA protocol used for the DB2 database defines no less than 8 different character code pages that can be used in a single session.

The arguments above testify to the sheer complexity of creating and maintaining a software package that implements these protocols.

***The place where vulnerabilities grow***

Because of their proprietary nature, the complete specifications of these protocols were never put to scrutiny by the public eye. Needless to say, that the source code, implementing the protocols, was never available for careful review by researchers.

Moreover, the vendors became the sole producers of basic client software (i.e. drivers). Third parties never had to care about the protocol details or the protocol behavior. Anybody who wanted to implement database client software relied on APIs supplied by vendor packages. The vendors on their side tested the robustness of their implementations not against the protocol specification but rather against the behavior imposed by their own APIs. Thus, if the server side code failed to validate the size of a field in the message, but the driver APIs did not allow a message to be generated with an illegal size, the vulnerability would not show up. In fact looking at the differences between different drivers produced by the same vendor it becomes apparent that some of the details of the protocols were not apparent even to vendor programmers.

Thus, for years database network communication protocols lurked in darkness, undisturbed by security researchers, their growing number of security vulnerabilities unreported and not being fixed. Those few protocol level vulnerabilities that were reported and fixed (see [ 18 ],[ 19 ],[ 20 ],[ 21 ]) were concerned only with the first protocol message sent from a client to the server. This actually indicated a second type of barrier for researchers – the lack of a proper research tool. There was no tool available for a researcher to construct arbitrary protocol message exchanges containing mostly legitimate messages mixed with a few tampered messages.

## Breaking the Code

In the past couple of years, researchers from database security vendors started to reconstruct the specifications for the various protocols. They had to do it as part of their effort to create network based security solutions for databases. This resulted in the elimination of the first barrier to discovering security vulnerabilities – the knowledge barrier.

Motivated by this new opportunity, researchers developed tools to help them overcome the technical barrier of creating tampered message exchange sequences. The common packet editing tools (see [ 22 ], [ 23 ], [ 24 ], [ 25 ]) simply were not enough. These tools allow one-way injection of a single, specially crafted, packet into the network. Using these tools to selectively modify payload of TCP segments within an exiting connection is a nearly impossible task. A useful tool we created in our labs is a generic TCP proxy called TCPirate. This tool relays TCP streams between a client and a server allowing full visibility, in real time, into the stream and providing the capability of selectively changing parts of the stream as well as injecting external data into the stream. Due to the new knowledge and new tools, database communication protocol vulnerabilities are starting to surface from the deeps.

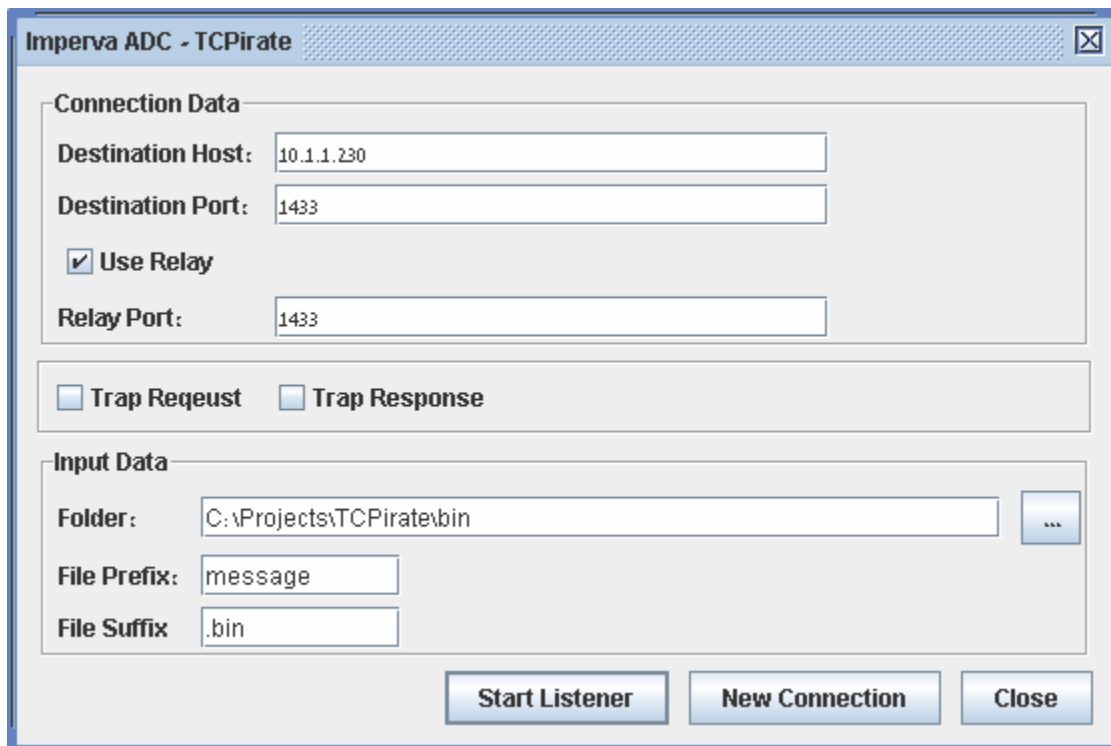


Figure 3: Configuration screen of TCPirate

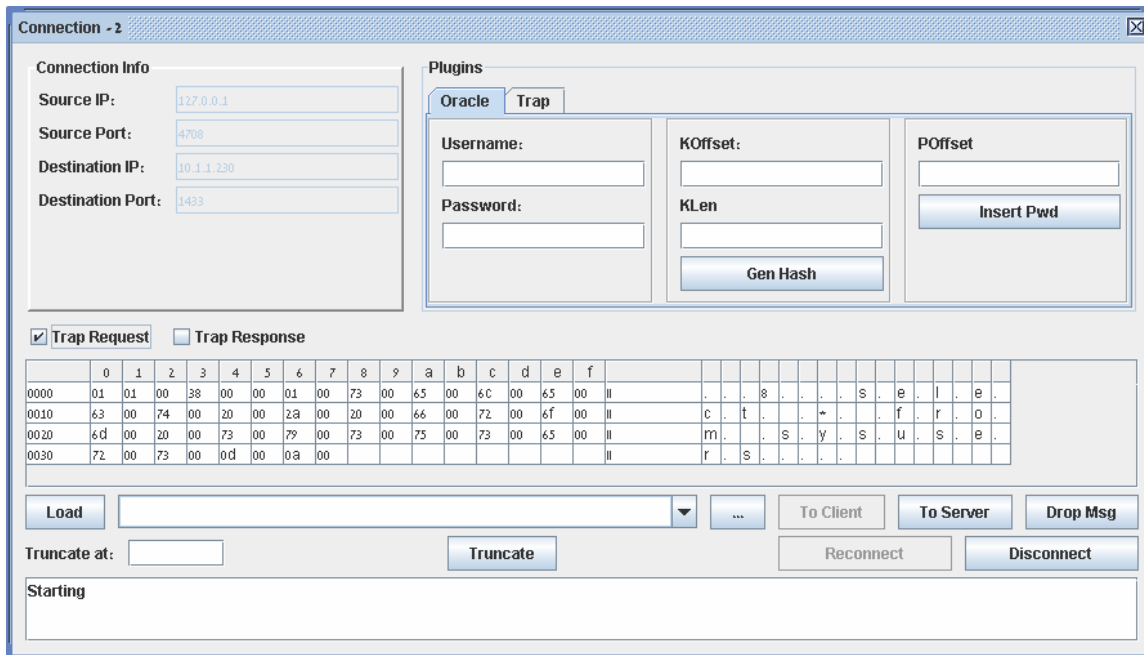


Figure 4: MS SQL Server query trapped in TCPirate

### Vulnerabilities Explained

Before we dive into the details of some vulnerabilities, it is important to understand the types of vulnerabilities and their potential impact. Here we suggest a classification method that is based on the type of manipulation that exposes the vulnerability:

- Message structure tampering. Mostly these vulnerabilities yield attacks against the parsing mechanism that typically result in memory corruption.
- Field size tampering. These vulnerabilities yield buffer overflow attacks against the basic parsing mechanism or against higher level mechanisms that have full trust in the robustness of the parsing mechanism.
- Field content manipulation. These vulnerabilities yield different type of attacks against higher level mechanisms including privilege elevation and audit evasion.
- Message sequence tampering. These vulnerabilities yield different types of attacks against various levels of database mechanisms.
- The following sub-sections discuss each category in more details, but the first example is of a vulnerability that does not require any type of tampering, but rather a network sniffer.

In order to maximize the strength of username/ password based authentication mechanism, a reply on to failed attempt should not disclose whether failure is due to bad username or incorrect password. The Oracle-API calls for login are compliant with the above behavior. However, looking at the actual message exchange generated by the client and the server as a consequence of using these API calls reveals that one message sequence is used when the username is unknown and a different message sequence is used when the password is incorrect. Thus, by using a network packet analyzer on his client machine, an attacker can launch effective brute-force attack on the database, first finding valid usernames and then looking for the password for each.

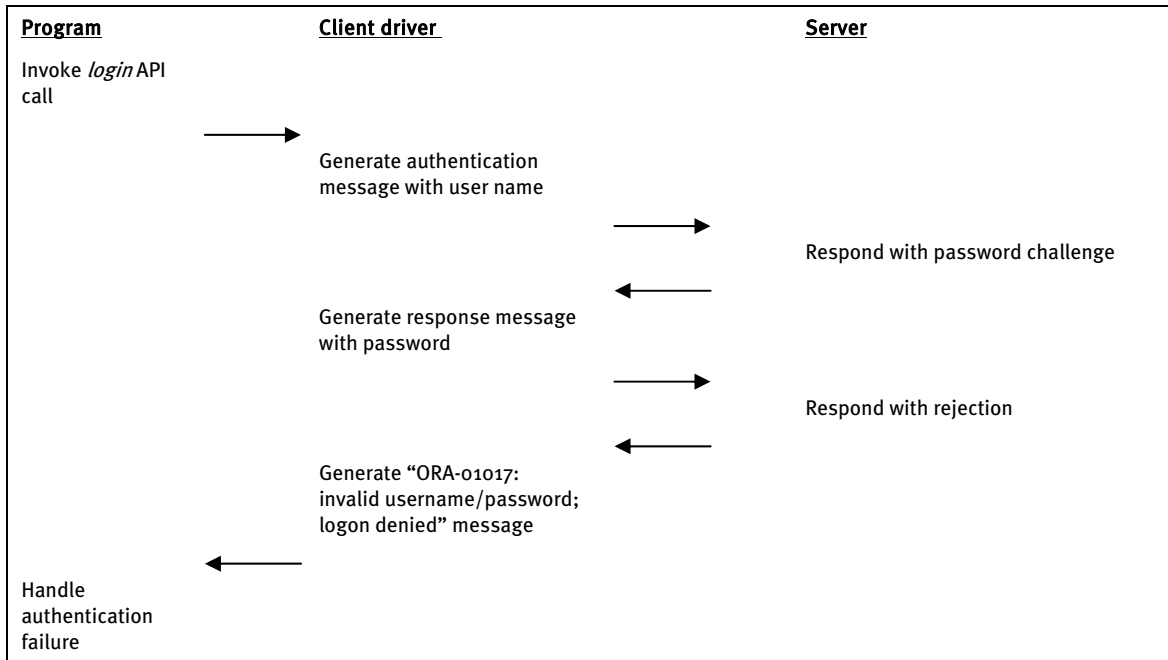


Figure 5: Oracle authentication message exchange - bad password

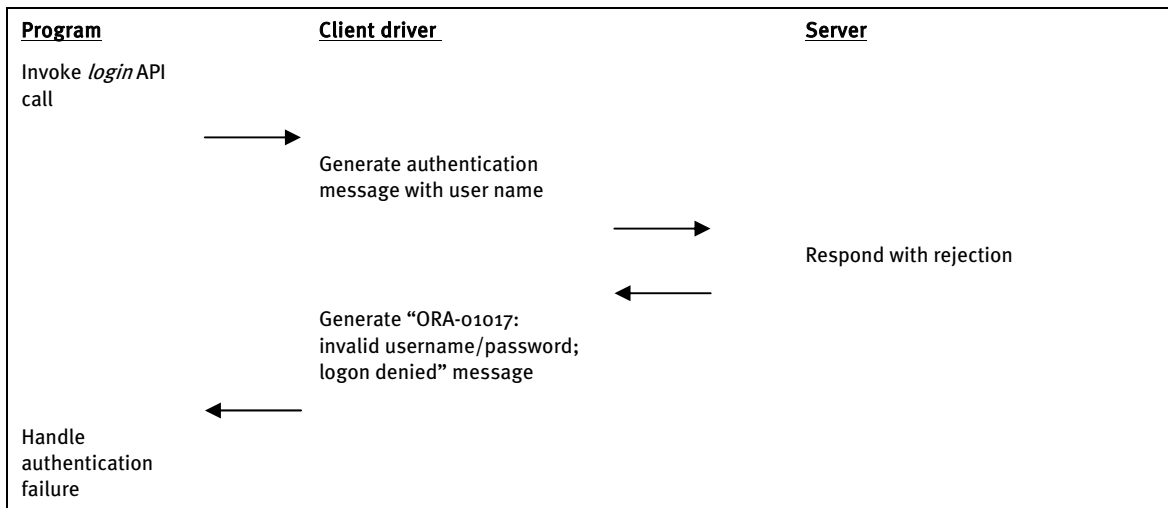


Figure 6: Oracle authentication message exchange - bad user name

The example above teaches us two important things: API programmers were aware of the need to produce a uniform response for the different types of authentication failure. Aware as they were, no one thought that an adversary would have used a local network sniffer to observe traffic. Of note, using encryption is of no avail since the size of the traffic discloses the nature of the response.

## Message Structure Tampering Vulnerabilities

A protocol message has a well defined, yet rarely documented, structure. The structure of a message can be described as a list of fields, where each field has a specific role and expected format. Some messages have a variable structure where the list of actual fields is determined by the state of the protocol or by the discretion of the sending party. The main tampering techniques for message structure are therefore the following:

- Removing fields from a message
- Adding fields to a message or duplicating fields in a message
- Combining fields in an unexpected manner

An excellent example of the first type of vulnerability is shown in an IBM DB2 vulnerability published in September 2006 (See [ 26 ]). One of the connection establishment messages is said to contain an optional database-name field. In practice, IBM produces clients that always include this field. As it turned out, when the message is sent without the “optional” database-name, an unhandled exception condition occurs on the server, yielding the database inaccessible to all clients.

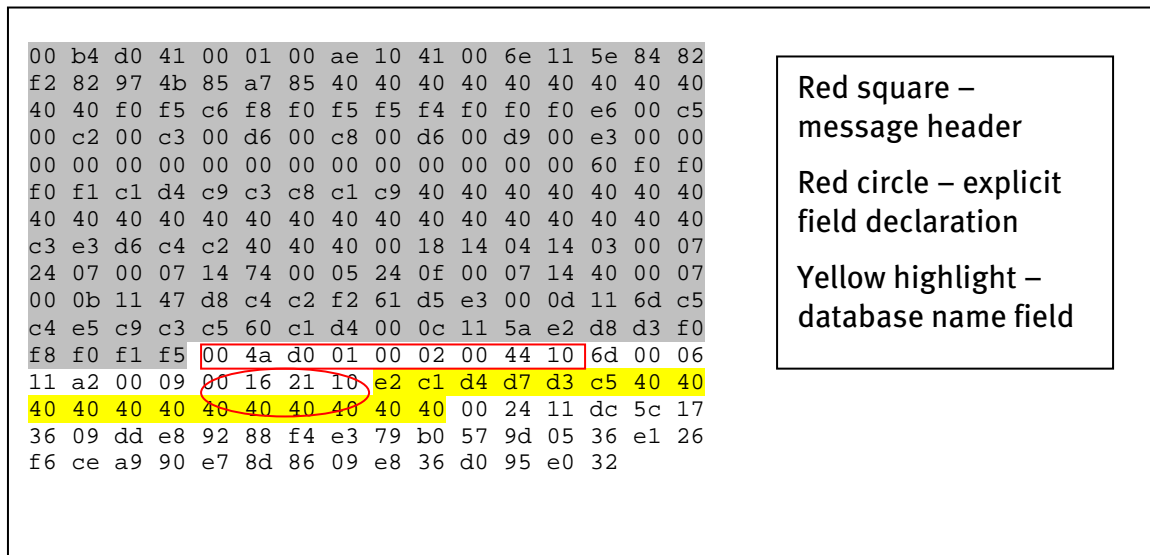


Figure 7: Original message with database name field

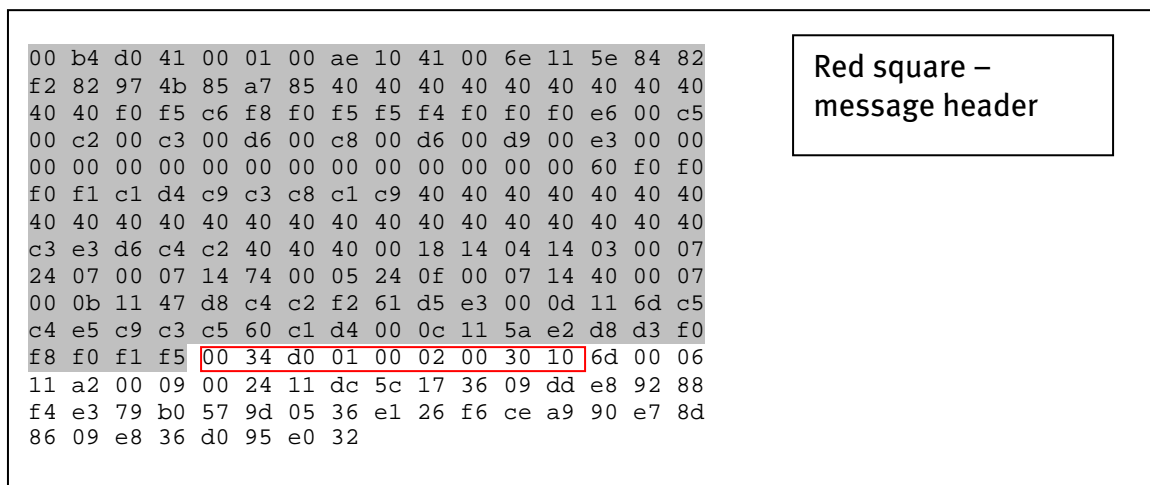


Figure 8: Tampered message with no database name field



While there is a very good example of the second type of tampering technique, at this time it can only be discussed in high level terms since it has not been patched. It is a vulnerability related to the Oracle database network communications protocol (SQL\*NET) in which adding fields to a message in the wrong (or “right”) context of the protocol state, allows unauthenticated access to the database server.

Exploiting the vulnerabilities discussed above requires more than just an editor and a telnet client. However, the exploit process is not rocket science either. The attacker could write a short program to send the tampered messages and process the server’s replies up to the point where the exploit takes effect, or use a standard, vendor provided, client software and redirect the traffic through a local, interactive TCP proxy (like TCPirate), tampering with the messages as they flow from the client to the server.

### ***Field size manipulation***

Occasionally, fields in a message have their sizes explicitly declared using another dedicated field. For example, the field declaring the type of message is a constant sized field in all protocols. However, the field containing the name of the database to connect to tends to have a variable size. The size is given by a fixed size field in another part of the same message. Thus tampering field sizes is actually a specific instance of the next type of vulnerabilities where the field being manipulated is the length field.

Tampering field sizes is mostly used for buffer overflow attacks yielding execution of arbitrary code. This type of vulnerability is the result of having the length indicator capable of expressing larger data sizes than actually supported by the server software. Some known example include [ 20 ] and [ 19 ]. A recent vulnerability uncovered in IBM’s DB2 database ([ 27 ]) can be used to demonstrate this issue in details. The session establishment message for the DRDA protocol includes a field called MGRLVLLS that holds an array with two integer columns. The number of rows in the array is expected to be a small constant, representing the number of main software modules that make up the client and server software. Most fields in DRDA protocol messages have their size explicitly defined using an integer field preceding the actual value. Thus, it is possible to create an array of a very large number of rows embedded within this field. As expected, when sending a large enough array (approx. 400 rows) a buffer overflow condition occurs in the server. This vulnerability can be easily exploited for bringing down the server by setting the value of array rows arbitrarily. The vulnerability can also be exploited for executing arbitrary code on the server by carefully constructing the contents of the array. Actually, it is one the more difficult code execution attacks to exploit. The tools needed for creating and launching an attack are a text editor and a Telnet client. Moreover, the attacker needs no initial access credentials for the database server. This vulnerability, at the time of discovery affected the DB2 database installed on many platforms including what many people would consider the most robust security platform – OS/390.

Yet there are other types of size tampering attacks. One interesting type is related to the tendency of protocol programmers to include redundant size information within a message without validating consistency between all size related fields. For example, in one of the TDS messages used by MS SQL Server there are three (3!) size related fields within the same message (see Figure 2). The first field describes the size of the entire message with the common protocol header fields. The second size field describes the size of the message without the common protocol header fields, yet another third size indicator is attached to each data field, indicating its individual size. This type of redundancy and inconsistency can result in various vulnerabilities. One amusing instance of such vulnerability actually concerns another TDS message (TDS “Hello” message). When the size indicator of an individual field is set to a number larger than the size of the message itself, an unexpected behavior by the server causes arbitrary memory buffers to be dumped to the network connection, thus exposing sensitive information (including passwords).

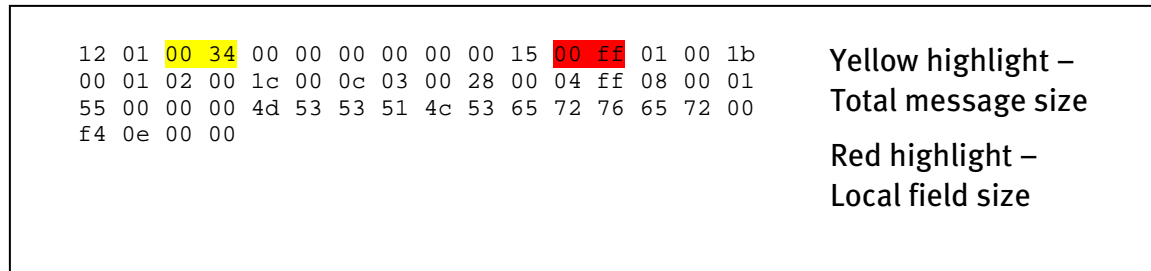


Figure 9: TDS Hello message - Field size larger than total message size

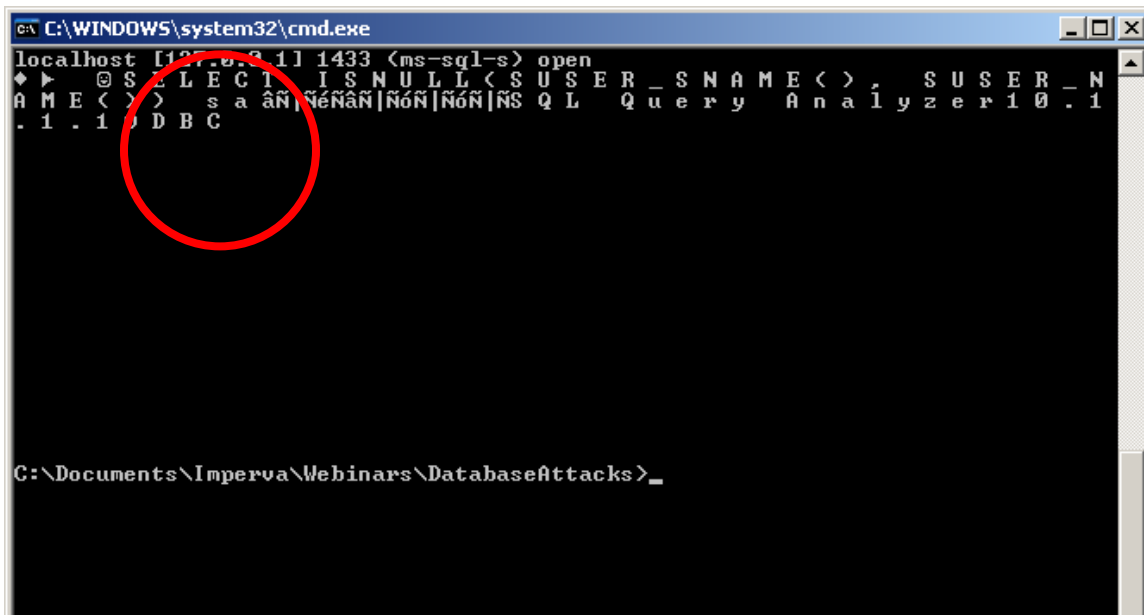


Figure 10: Sample buffer dumped by the server showing names of connected users (sa)

### Field Content Manipulation

If you are responsible for the security of a database server, this is the type of vulnerabilities that should make you tremble in fear. Content manipulation is usually straight forward to perform and the effects are directly reflected in the behavior of data processing mechanisms. The details revealed for some vulnerabilities and proclaimed effects of some vulnerabilities whose details were not disclosed are frightening.

Let us take a look at an Oracle SQL\*NET vulnerability disclosed in January 2006 (See [ 28 ]). The authentication message used by the SQL\*NET protocol includes a multitude of fields, whose values should be evaluated in the event of a successful authentication. One of these fields, called AUTH ALTER\_SESSION, includes a command whose primary purpose was to set the proper language support for the client’s session.

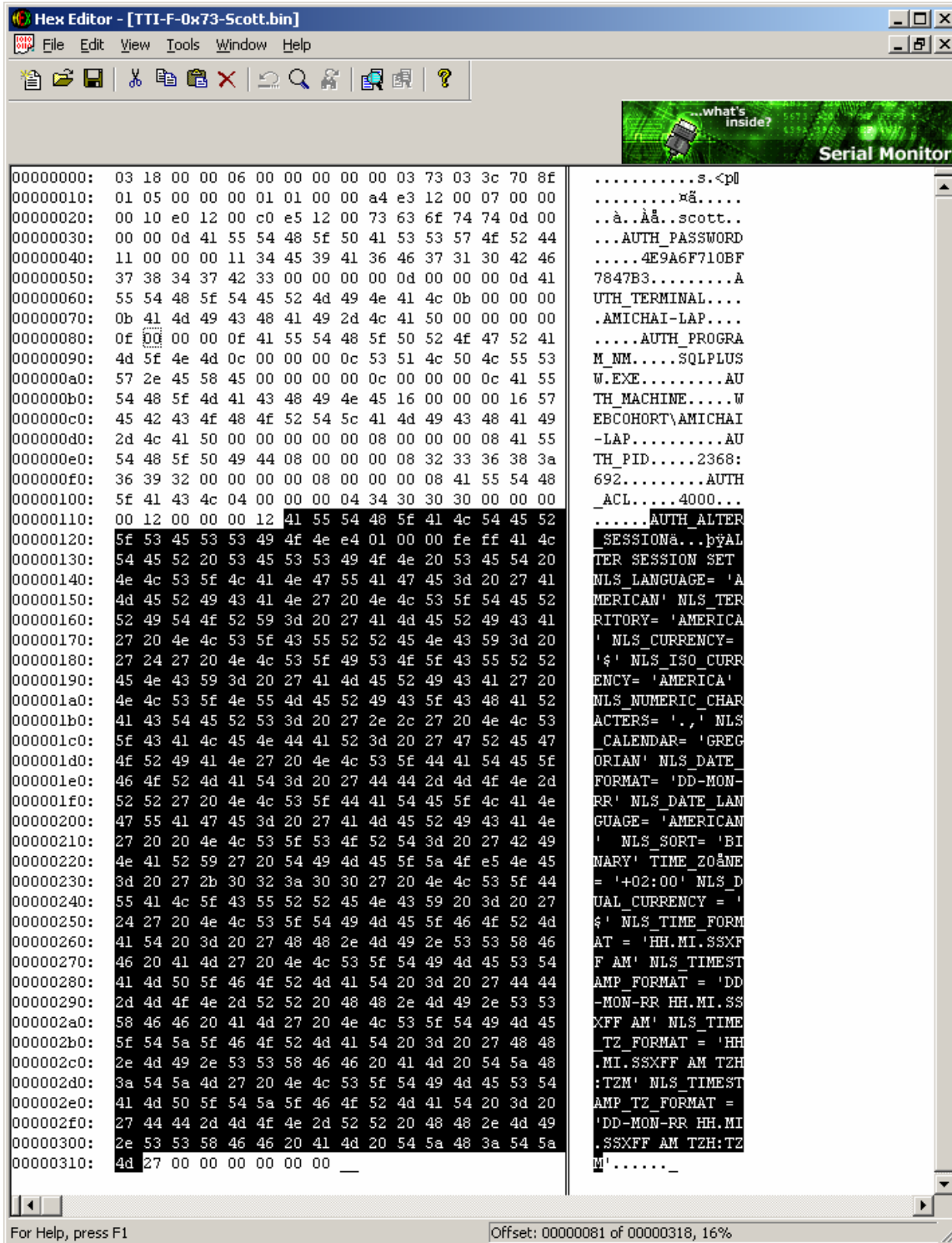


Figure 11: Oracle authentication package - AUTH\_ALTHERR\_SESSION is selected

However, processing semantics of this field go far beyond this simple and benign task and in fact the contents of this field are evaluated as an SQL statement by the database upon successful login. It so happens that this SQL statement is executed in the same security context of the authentication procedure (rather than the security context of the user logging into the system). This security context is actually the highest available in the Oracle server and allows execution of any command without any restriction and without any audit trail. In order to exploit this vulnerability an attacker needs to change the contents of this field into a command of his

liking (e.g. granting administrative privileges to himself). This can be achieved by routing the traffic from a vendor made client software through an interactive TCP proxy or by editing (using a text editor or a more convenient hex-editor) the client software DLLs and then simply using the vendor made client software. An interesting thing to notice about this vulnerability is that when discovered, it applied to all versions of Oracle server ranging from the obsolete Oracle 8i to the most up-to-date Oracle 10gR2. Needless to say that it applied to any operating system platform.

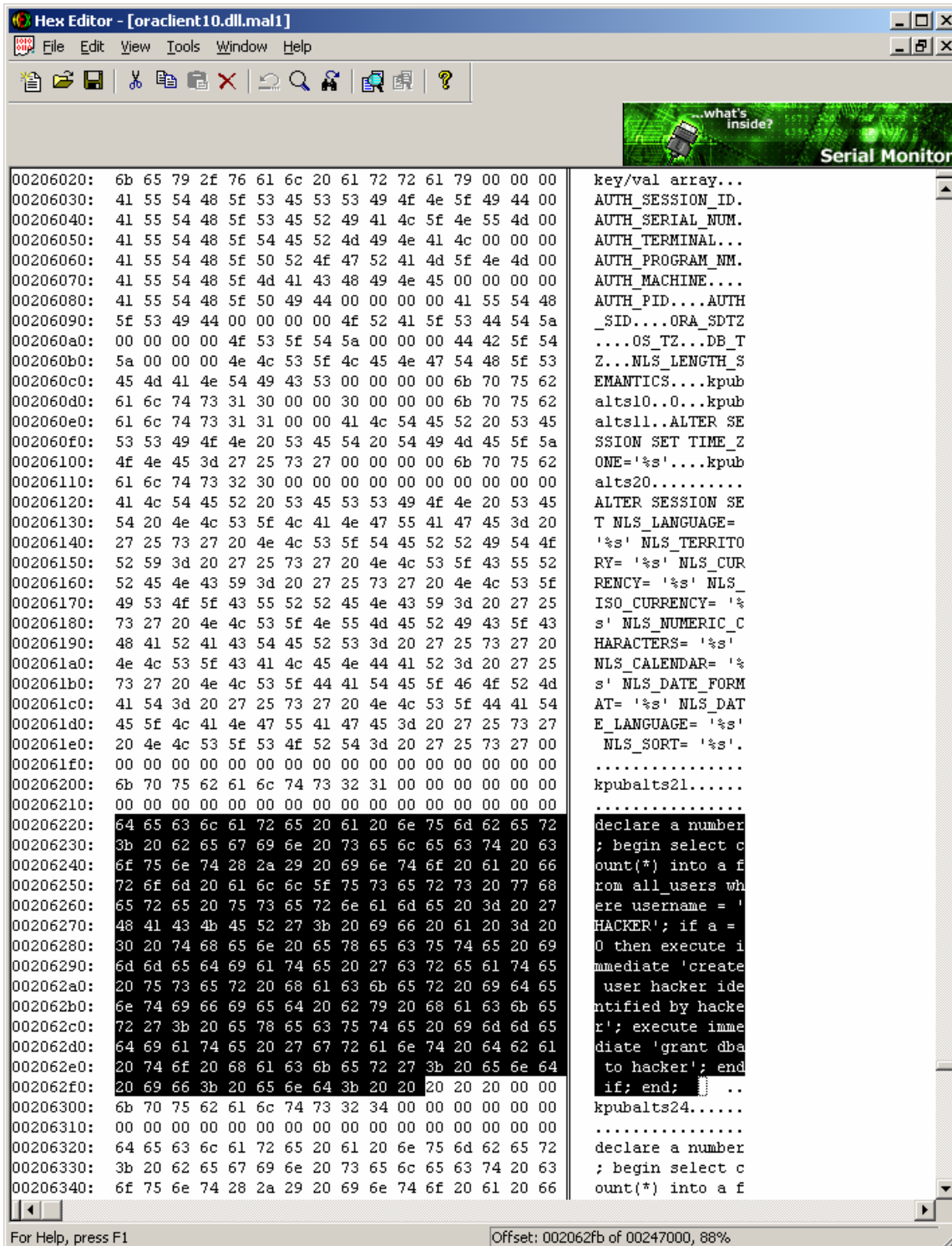


Figure 12: Editing oraclient10.dll to include a privileged command

While the above type of attack relies on processing semantics there are certain attacks that rely on the more basic syntactic parsing issues. Certainly one of the most prominent issues for creating parsing inconsistency is the use of the NULL character (The character whose code is 0x0000). In one vulnerability disclosed in January 2006 (see [ 29 ]) an attacker alters the login message of the TDS protocol to add a leading NULL character to the actual user name. While the authentication mechanism is programmed to ignore this addition, the tracing mechanism does not. As a consequence that attacker is logged into the database but the audit trail cannot associate between the activities performed and the identity of the attacker. This vulnerability is a classic case of a server being protected by the vendor created client software who denied the use of NULL characters in its login API.

```

10 01 00 a6 00 00 01 00 9e 00 00 00 01 00 00 71
00 00 00 00 00 00 00 07 ac 11 00 00 00 00 00 00
e0 03 00 00 88 ff ff ff 0d 04 00 00 56 00 00 00
56 00 02 00 5a 00 02 00 5e 00 12 00 82 00 0a 00
00 00 00 00 96 00 04 00 9e 00 00 00 9e 00 00 00
00 16 41 56 c2 71 00 00 00 00 9e 00 00 00 69 00
63 00 33 a5 93 a5 53 00 51 00 4c 00 20 00 51 00
75 00 65 00 72 00 79 00 20 00 41 00 6e 00 61 00
6c 00 79 00 7a 00 65 00 72 00 31 00 30 00 2e 00
31 00 2e 00 31 00 2e 00 32 00 33 00 31 00 4f 00
44 00 42 00 43 00

```

Red highlight is user name (Unicode)

Figure 13: MS SQL Server login message with user name *ic*

```

10 01 00 a8 00 00 01 00 a0 00 00 00 01 00 00 71
00 00 00 00 00 00 00 07 ac 11 00 00 00 00 00 00
e0 03 00 00 88 ff ff ff 0d 04 00 00 56 00 00 00
56 00 03 00 5c 00 02 00 60 00 12 00 84 00 0a 00
00 00 00 00 98 00 04 00 a0 00 00 00 a0 00 00 00
00 16 41 56 c2 71 00 00 00 00 a0 00 00 00 00 00
69 00 63 00 33 a5 93 a5 53 00 51 00 4c 00 20 00
51 00 75 00 65 00 72 00 79 00 20 00 41 00 6e 00
61 00 6c 00 79 00 7a 00 65 00 72 00 31 00 30 00
2e 00 31 00 2e 00 31 00 2e 00 32 00 33 00 31 00
4f 00 44 00 42 00 43 00

```

Red highlight is user name (Unicode) preceded by NULL character

Figure 14: MS SQL Server login message with user name preceded by NULL character

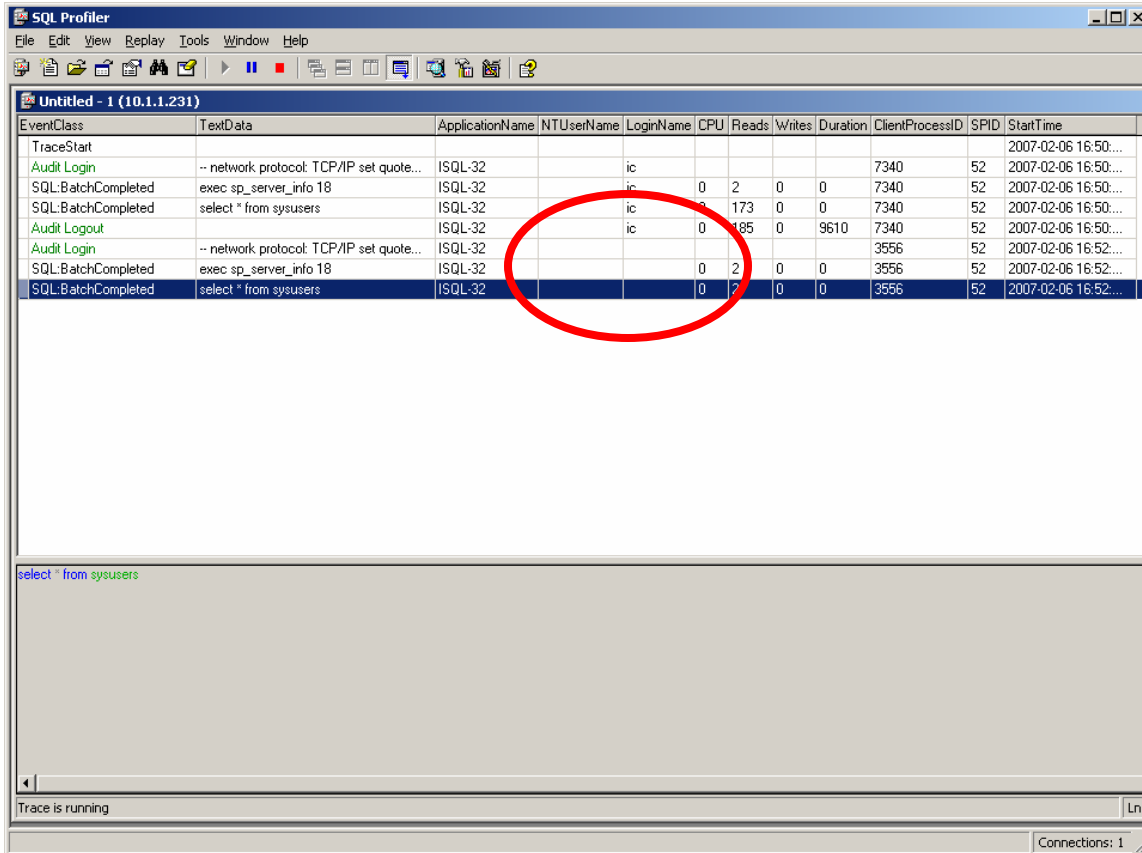


Figure 15: SQL profiler showing audit records of the "invisible user"

### Message Sequence Tampering

This last category of vulnerabilities is the most difficult to demonstrate. While I do have compelling example, its details cannot be exposed until the vendor patches it. In general terms, an attacker can issue an irregular sequence of well formed protocol messages that will effectively yield the server inaccessible to other users. Exploiting this vulnerability requires basic scripting capabilities and can even be pursued manually with no automation.

## **New Attack Types Require New Protection Solutions**

It is evident from the discussion above that database communication protocol vulnerabilities are an emerging threat to database servers of all types. The question that must be asked is can we use the traditional security mechanisms to mitigate the risk associated with them.

We cannot rely on internal server mechanisms to provide for proactive security measures against these types of vulnerabilities. It will simply be as if we expected the programmers not to introduce these vulnerabilities in the first place. This brings us to the question of whether programmers can actually do that. The answer in my opinion is obviously no. Otherwise they would have eliminated any other programming flaws that exist in the software.

Reactive protection is suggested by database vendors through orthodox patching of the software. However, given the time scales required for vendors to actually issue patches (months to years) and the time required for safely deploying the patch in a large production environment (months to years), this clearly should be considered the last line of defense. Traditional IDS / IPS vendors offer partial reactive solution for some of the vulnerabilities. While their timeframe for issuing a solution is usually acceptable their coverage is very low because they lack proper insight into the network protocol used by database servers.

We therefore call for a new breed of solutions that should be coupled with existing database protection mechanisms. The new database IDS / IPS solution must have through understanding and deep insight into the communications protocol used by the database server. Having this type of capabilities allows the database IPS to provide proactive validation of protocol messages as they flow from the client to the server. Any message or message sequence that does not comply with the expected behavior (as defined by analyzing the behavior of vendor produced software client) is immediately flagged and possibly discarded. This mechanism can be used for timely detection of zero-day attacks. The proactive mechanism can be complemented with a reactive mechanism based on frequent signature updates for accurate detection and blocking of known vulnerabilities.

## References

- [ 1 ] [Microsoft SQL Server Xp\\_sprintf buffer overflow](#)
- [ 2 ] [Microsoft SQL Server / Data Engine xp\\_SetSQLSecurity Buffer Overflow Vulnerability](#)
- [ 3 ] [Microsoft SQL Server 2000 Password Encrypt Procedure Buffer Overflow Vulnerability](#)
- [ 4 ] [Multiple Oracle Database Parameter/Statement Buffer Overflow Vulnerabilities](#)
- [ 5 ] [Sybase Adaptive Server DROP DATABASE Buffer Overflow Vulnerability](#)
- [ 6 ] [IBM DB2 Universal Database REC2XML and GENERATE\\_DISTFILE Buffer Overflow Vulnerabilities](#)
- [ 7 ] [Oracle 10g Database SUBSCRIPTION\\_NAME Remote SQL Injection Vulnerability](#)
- [ 8 ] [Oracle Database Server ALTER\\_MANUALLOG\\_CHANGE\\_SOURCE SQL Injection Vulnerability](#)
- [ 9 ] [Oracle Database SYS.KUPV\\$FT Multiple SQL Injection Vulnerabilities](#)
- [ 10 ] [Oracle Database Server ctxsys.driload Access Validation Vulnerability](#)
- [ 11 ] [Microsoft SQL Server 2000 sp\\_MScopyscript SQL Injection Vulnerability](#)
- [ 12 ] [Microsoft SQL Server Non-Validated Query Vulnerability](#)
- [ 13 ] [Microsoft SQL Server 7.0 Stored Procedure Vulnerability](#)
- [ 14 ] [Read-only user can modify data](#)
- [ 15 ] [FreeTDS](#)
- [ 16 ] [iTDS](#)
- [ 17 ] [DataDirect Technologies](#)
- [ 18 ] [Microsoft SQL Server 7.0 NULL Data DoS Vulnerability](#)
- [ 19 ] [Microsoft SQL Server User Authentication Remote Buffer Overflow Vulnerability](#)
- [ 20 ] [Oracle 8i TNS Listener Buffer Overflow Vulnerability](#)
- [ 21 ] [Oracle TNSListener SERVICE\\_NAME Remote Buffer Overflow Vulnerability](#)
- [ 22 ] [Packit – Network Injection and Capture](#)
- [ 23 ] [wINJECT \[drugs for Windows\]](#)
- [ 24 ] [Packetyzer – Network protocol analyzer for Windows](#)
- [ 25 ] [ColaSoft Packet Builder](#)
- [ 26 ] [Details for BID 19586](#)
- [ 27 ] [Details for BID 18428](#)
- [ 28 ] [US CERT Vulnerability Note VU#871756](#)
- [ 29 ] [BUG: Login names that contain leading zero characters are not visible when you use SQL Profiler to audit connections to SQL Server 2000](#)



**US Headquarters**  
950 Tower Lane  
Suite 1550  
Foster City, CA 94404  
Tel: +1-650-345-9000  
Fax: +1-650-345-9004

**International Headquarters**  
125 Menachem Begin Street  
Tel-Aviv 67010  
Israel  
Tel: + 972-3-6840100  
Fax: + 972-3-6840200