

Thick Client Application Security

Arindam Mandal (arindam.mandal@paladion.net)

Paladion Networks (<http://www.paladion.net>)

January 2005

This paper discusses the critical vulnerabilities and corresponding risks in a two tier thick client application along with the measures to mitigate risks. Thick client is defined as an application client that processes data in addition to rendering. An example of thick client application can be a Visual Basic, JAVA or VB.NET application that communicates with a database.

The risks observed in thick client applications generally include information disclosure, unauthorized access, authentication bypass, application crash, unauthorized execution of high privilege transactions or privilege escalation. It is interesting to note that most of the Open Web Application Security Project¹ (OWASP) Top 10 vulnerabilities are as applicable to Thick client applications as they are to web applications. The table below provides a mapping.

Serial No.	OWASP Top Ten Most Critical Web Application Vulnerabilities	Thick Client Most Critical Application Vulnerabilities
1.	Unvalidated Input	Unvalidated Input
2.	Broken Access Control	Broken Access Control
3.	Broken Authentication and Session Management	Weak Authentication and Session Management
4.	Cross Site Scripting (XSS) Flaws	Not Applicable
5.	Buffer Overflows	Buffer Overflows
6.	Injection Flaws	Injection Flaws
7.	Improper Error Handling	Improper Error Handling
8.	Insecure Storage	Insecure Storage
9.	Denial of Service	Denial of Service
10.	Insecure Configuration Management	Insecure Configuration Management

In the following sections, we shall discuss in detail few of the critical vulnerabilities in thick client applications including unvalidated input, weak authentication method, sensitive data in memory, critical data in files & registry and impersonating a high privilege user.

¹ OWASP Top Ten Most Critical Web Application Vulnerabilities details is available at:
<http://www.owasp.org/documentation/topten.html>

Unvalidated Input

User requests are not validated before being used by the application. Attackers can use this flaw to launch SQL injection attack. SQL injection attack is possible when application does not validate user input for special characters and codes. An attacker can exploit this vulnerability by supplying specially crafted SQL statement to the application.

Risk

This attack can result in authentication bypass, viewing unauthorized data, application malfunction, data deletion or malicious command injection including database shutdown.

Sample Exploit – Authentication Bypass

An application uses the following SQL query to validate user credentials.

```
SELECT *  
FROM USER_TABLE  
WHERE USER_ID=' ' AND PASSWORD=' ';
```

Application will authenticate a user if the database returns a row of data against this query. A valid user supplies the following credential to the application:

```
USER_ID: ADMIN  
PASSWORD: test123
```

Then the query will be:

```
SELECT *  
FROM USER_TABLE  
WHERE USER_ID='ADMIN' AND PASSWORD='test123';
```

The attacker knows the **USER_ID** i.e. **ADMIN** and he crafts the query:

```
USER_ID: ADMIN'--
```

So the new SQL query will be:

```
SELECT *  
FROM USER_TABLE  
WHERE USER_ID='ADMIN'--' AND PASSWORD=' ';
```

This query will comment out password checking logic and database will execute only uncommented query portion. This will return entire row of user **ADMIN**. Application assumes it as valid user and will grant the access.

Sample Exploit – Application Malfunction

Many two tier applications have the following authentication process: When a user enters the Login ID and password, application sends a query to the database to retrieve user's credentials.

The application sends following query to retrieve the password hash.

```
SELECT HASHED_PASSWORD
FROM USER_TABLE
WHERE USER_ID= ' ';
```

The user enters his **USER_ID**

USER_ID: ADMIN

Application will send following query to the database:

```
SELECT HASHED_PASSWORD
FROM USER_TABLE
WHERE USER_ID='ADMIN';
```

Normally, database will respond with following data.

```
..... HASHED_PASSWORD ..... f85fcb8456edb3824f4e32a56cdb76cd .....
```

Now, instead of giving proper **USER_ID** in this field, malicious user gives the following as input.

USER_ID: ' '

So application will now send following query to the database:

```
SELECT HASHED_PASSWORD
FROM USER_TABLE
WHERE USER_ID=' ';
```

Database will not be able to understand the query and will send a syntax error to the application. Now, if the application is not designed to handle this kind of database error, it will crash.

To avoid this type of application crash; developers usually restrict users to type-in ' (single quote) in any form field of the application. But a malicious user can type special characters like ' (single quote) in a text file and copy & paste them in the input form field of the application and crash it.

Sample Exploit – Database Shutdown using Command Injection

An application uses the following SQL query to validate user credentials.

```
SELECT *
FROM USER_TABLE
WHERE USER_ID=' ' AND PASSWORD=' ';
```

Application will authenticate a user if the database returns a row of data against this query. A valid user supplies the following credential to the application:

USER_ID: ADMIN
PASSWORD: test123

Then the query will be:

```
SELECT *  
FROM USER_TABLE  
WHERE USER_ID='ADMIN' AND PASSWORD='test123';
```

The attacker only knows the **USER_ID** i.e. **ADMIN** and he crafts the query:

```
USER_ID: ADMIN'; SHUTDOWN WITH NO WAIT; --
```

So the new SQL query will be:

```
SELECT *  
FROM USER_TABLE  
WHERE USER_ID='ADMIN ' ; SHUTDOWN WITH NO WAIT; -- ' AND PASSWORD=' ' ;
```

This query will comment out password checking logic and database will execute only uncommented query portion. After executing the SELECT query database will move to next query and execute the statement i.e. **SHUTDOWN WITH NO WAIT** and will immediately shutdown the database.

Solution

The solution to this problem is input validation. The application should check for any special character and code in the user input. In case any special character like ' (single quote) or command code is present; application should ignore the input and throw appropriate error message to the user.

Weak Authentication Method

Attackers can compromise the application if account credentials and session IDs are not protected. Our experience shows that many applications use weak authentication method. For example, when a user enters his Login ID and password, the hashed or clear text password of corresponding Login ID is retrieved from the database and brought to the client. The entered password is hashed at the client and the two passwords are compared. Therefore, even on entering a wrong password for a known login ID, the actual hashed or clear text password comes to the client. The attacker can sniff hashed or clear text password on transit and use it for man in the middle attack or to login into the application.

Risk

Attackers can steal passwords, gain access to the application and escalate privilege by sniffing and man in the middle attack.

Sample Exploit – Password Compromise

An example of password compromise by sniffing attack using Ethereal² is shown below:

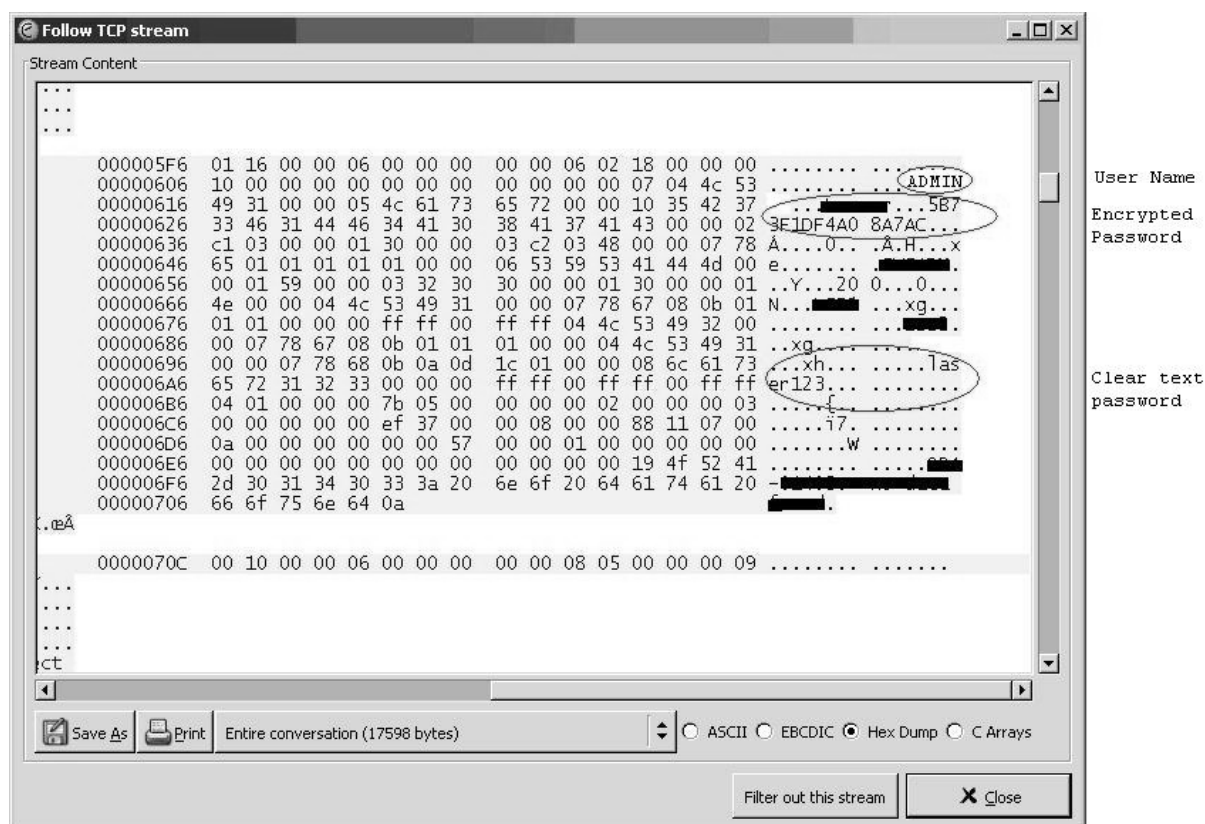


Figure 1: Sniffing password using Ethereal

Using this technique an attacker can get the username and password of other users.

Sample Exploit – Privilege Escalation

A low privileged user, who has the valid password and its hash for his account can intercept the traffic using Interactive TCP Relay³ (ITR) for man in the middle attack, can replace the actual hash of the high privileged user which is coming from the database server with known hash, usually hash of his own password. Then he can supply any high privilege user account name but with his own password to the application. The application will generate the hash for his password and compare it with the modified hash. The authentication will be successful since both the hash values are now the same and authenticate him as a high privilege user.

A snapshot of man in the middle attack using ITR is shown below.

² Ethereal is a sniffer that can capture the traffic in a network and also decode certain protocols. It is available for download at <http://www.ethereal.com/download.html>

³ Interactive TCP Replay is a tool that acts as a proxy for non-HTTP applications and also allows modifying the traffic. It allows editing of the messages in a hex editor. ITR also logs all the messages passing between the client and the server. It can use different types of character encoding like ASCII or EBCDIC for editing and logging. More information on this can be found at http://www.webcohort.com/web_application_security/research/tools.html

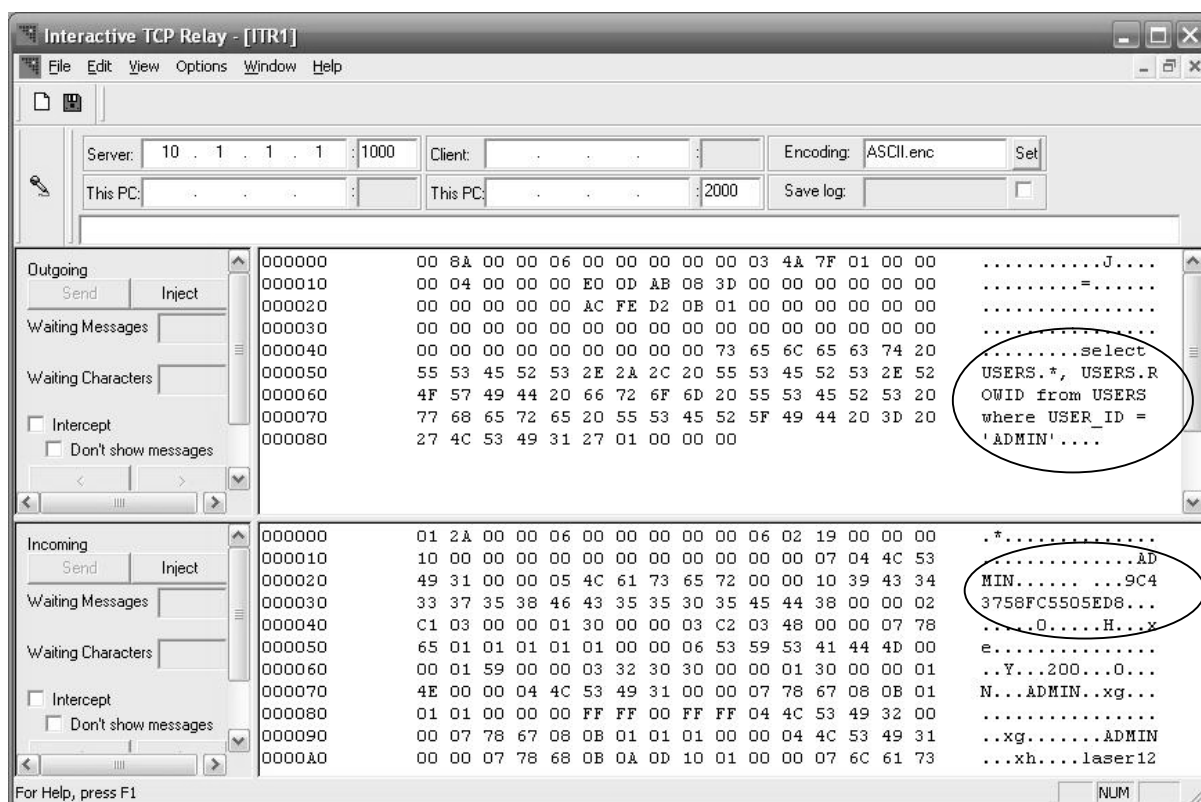


Figure 2: Man in the middle attack using ITR

Let's see an example of privilege escalation:

User **ADMIN** has password **sensitiveaccount**. Its MD5 hash is stored in the database as **f85fcb8456edb3824f4e32a56cdb76cd**. The malicious user knows that MD5 for password **hackaccount** is **b2eb5537c26694bbb8e7ea80ceb398ca**. He supplies username as **ADMIN** and password as **hackaccount** to application's authentication page and starts intercepting the traffic using ITR. The application will send a query to database like:

```
SELECT HASHED_PASSWORD
FROM USER_TABLE
WHERE USER_NAME='ADMIN';
```

Database will send the response:

```
..... HASHED_PASSWORD ..... f85fcb8456edb3824f4e32a56cdb76cd .....
```

Now, the malicious user will intercept the traffic and alter the hash with a known hash, i.e. hash of **hackaccount**. The modified response will be:

```
..... HASHED_PASSWORD ..... b2eb5537c26694bbb8e7ea80ceb398ca .....
```

Application will now convert the user supplied password **hackaccount** to MD5 hash, compare it with modified database response and authenticate the malicious user as admin to the application.

Solution

The solution to this problem is the application should send the Login ID and the password hash from the client to the database server. The server should compare the received hash with the one stored in the database using a stored procedure and then authenticate the user. To prevent man in the middle attack the traffic between client and server should be encrypted.

For Oracle database the following solution can be applied for data confidentiality and integrity:

Oracle advanced security feature provides data encryption and integrity facility. It supports DES, 3DES, AES and RC4 algorithms for data encryption and SHA-1 & MD5 for data integrity. It uses Diffie-Hellman Based Key Management.

To use 3DES and MD5 algorithms for confidentiality and integrity between client and server, following settings can be used:

In `sqlnet.ora` file of the Oracle Server following lines can be added:

```
SQLNET.CRYPTO_CHECKSUM_TYPES_SERVER= (MD5)
SQLNET.ENCRYPTION_TYPES_SERVER= (3DES168)
SQLNET.ENCRYPTION_SERVER = required
SQLNET.CRYPTO_CHECKSUM_SERVER = required
SQLNET.CRYPTO_SEED = serverkeyseed
```

`SQLNET.CRYPTO_SEED` can be changed with a suitable value.

In `sqlnet.ora` file of the Oracle Client following lines can be appended:

```
SQLNET.CRYPTO_CHECKSUM_TYPES_CLIENT= (MD5)
SQLNET.AUTHENTICATION_SERVICES= (NTS)
SQLNET.CRYPTO_CHECKSUM_CLIENT = required
SQLNET.ENCRYPTION_TYPES_CLIENT= (3DES168)
SQLNET.ENCRYPTION_CLIENT = required
SQLNET.CRYPTO_SEED = clientkeyseed
```

`SQLNET.CRYPTO_SEED` can be changed with a suitable value. But it must not be same with the Server's `SQLNET.CRYPTO_SEED` value.

`sqlnet.ora` file in client and server should be protected against any unauthorized use. Except administrator, no body should have permission to view or access the file.

Similarly in MSSQL database environment; data can be protected using SSL security at client and server end. To know more about "How to Use SSL to Secure Communication with SQL Server 2000" check the following link:

<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/secmod/html/secmod33.asp>

Critical data in files & registry

Application stores critical information in files and registry. These files generally contain cryptographic keys, username, password of application and database. If these files are not secured, a malicious user can delete, modify or read data from the files. Assume that the application fetches the data from a file to connect to a remote database using service name, IP address, port name, user ID and password. Now, if the permission on this file is not secured, anybody can access the file to get user ID and password of the database. Again only an access control list (ACL) cannot prevent a clear text stored password from disclosure. A malicious user can use username and password to connect the database directly with admin privileges.

Risk

An attacker can retrieve user passwords and cryptographic keys from files and registry. Attacker can use the same to decrypt password, gain access to the application and access sensitive data in the database.

Sample Exploit – Database credential in file

An example of database credential is stored in a file in clear text format is shown below.

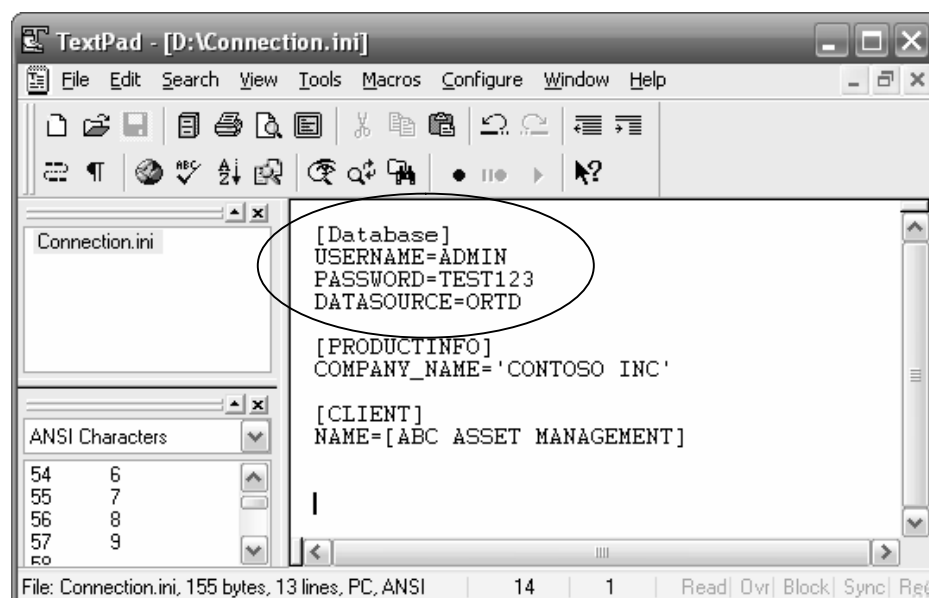


Figure 3: Database credential stored in clear text format

Solution

There are various solutions for these problems:

- § Secure the permission on application program files. Give full control permission only to application service account and administrator account. Other users and groups can be given only read permission.

- § Restrict the permission on user and password file (if present). Store the password in one way hash format (e.g. MD5⁴) in the file.
- § Restrict the permission on database credential file. If database username and password are stored in this file then store them in encrypted or one way hashed format (e.g. MD5).
- § If the application stores database and user credentials in the OS registry, secure the registry as well. Access permission to the sensitive registry hive should be restricted to normal user and user password should be stored in one way hashed format.

Sensitive data in memory

Application temporarily stores data into the memory from different environments like users or network for further processing. This type of data includes user passwords, cryptographic keys or sensitive data. Thick client application generally has a user authentication form, where user supplies username and password as credentials. These credentials are stored in the memory for future processing by the application. To validate these credentials, application fetches correct username & password from the database. This actual username & password is also stored temporarily in the memory. These credentials stay in the memory until they get overwritten by any other data. A malicious user can run tool like WinHex⁵ in the machine and can see the entire memory content used by the application process. From this dump he can find out the username, password and other sensitive data used by the previous user.

Risk

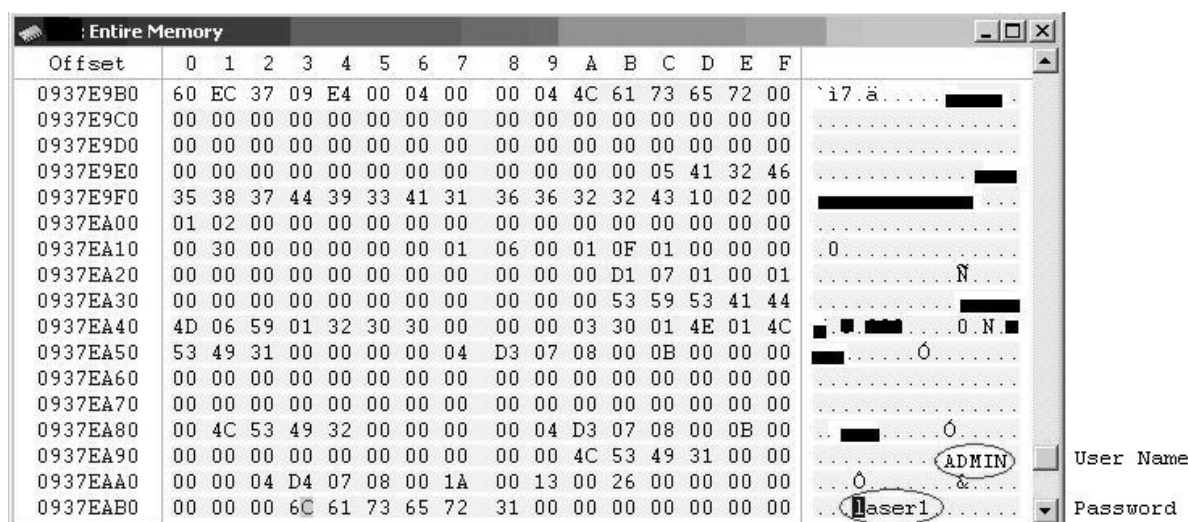
An attacker can retrieve user passwords and cryptographic keys from memory. Attacker can use the same keys to decrypt password, gain access to the application and access sensitive data in the database.

⁴ MD5 one way hashing algorithm is described in RFC 1321. It can be found at <http://www.ietf.org/rfc/rfc1321.txt>

⁵ WinHex is a tool used to view the contents of the memory. It can be used to view the memory content per process basis and search within the content. The trial version is available for download at <http://www.winhex.com/winhex/index-m.html>

Sample Exploit – Password in memory

An example of password in memory is shown below.



Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0937E9B0	60	EC	37	09	E4	00	04	00	00	04	4C	61	73	65	72	00	i7.a.....
0937E9C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0937E9D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0937E9E0	00	00	00	00	00	00	00	00	00	00	00	00	05	41	32	46
0937E9F0	35	38	37	44	39	33	41	31	36	36	32	32	43	10	02	00
0937EA00	01	02	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0937EA10	00	30	00	00	00	00	00	01	06	00	01	0F	01	00	00	00	.0.....
0937EA20	00	00	00	00	00	00	00	00	00	00	00	D1	07	01	00	01N
0937EA30	00	00	00	00	00	00	00	00	00	00	00	53	59	53	41	44
0937EA40	4D	06	59	01	32	30	30	00	00	00	03	30	01	4E	01	4C0.N
0937EA50	53	49	31	00	00	00	00	04	D3	07	08	00	0B	00	00	00Ó
0937EA60	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0937EA70	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0937EA80	00	4C	53	49	32	00	00	00	00	04	D3	07	08	00	0B	00Ó
0937EA90	00	00	00	00	00	00	00	00	00	00	4C	53	49	31	00	00ADMIN
0937EAA0	00	00	04	D4	07	08	00	1A	00	13	00	26	00	00	00	00&
0937EAB0	00	00	00	6C	61	73	65	72	31	00	00	00	00	00	00	00User1

Figure 4: Password in memory

Solution

Clear the memory area that contains critical data after a sensitive transaction. This means application should clear the username and password from the memory after authentication process. This will ensure that later on a malicious user cannot retrieve any sensitive data from the memory.

Impersonating a high privilege user

Normally data from the application are sent in clear text format at default port of the database (e.g. Oracle uses port 1521 for listener service). A malicious user can intercept the traffic using ITR and perform transactions with database with high privileged user's context.

Risk

A malicious user can intercept the traffic using ITR, read and modify the data or password with high privileged user ID and perform transaction with the database on that user's context.

Sample Exploit – Password Change

A low privileged user can change the password of a high privileged user. All SQL statements are sent from client to database server in clear text. When a user changes his password, an UPDATE query of password change with reference of user attribute (e.g. USER_ID) goes to database server.

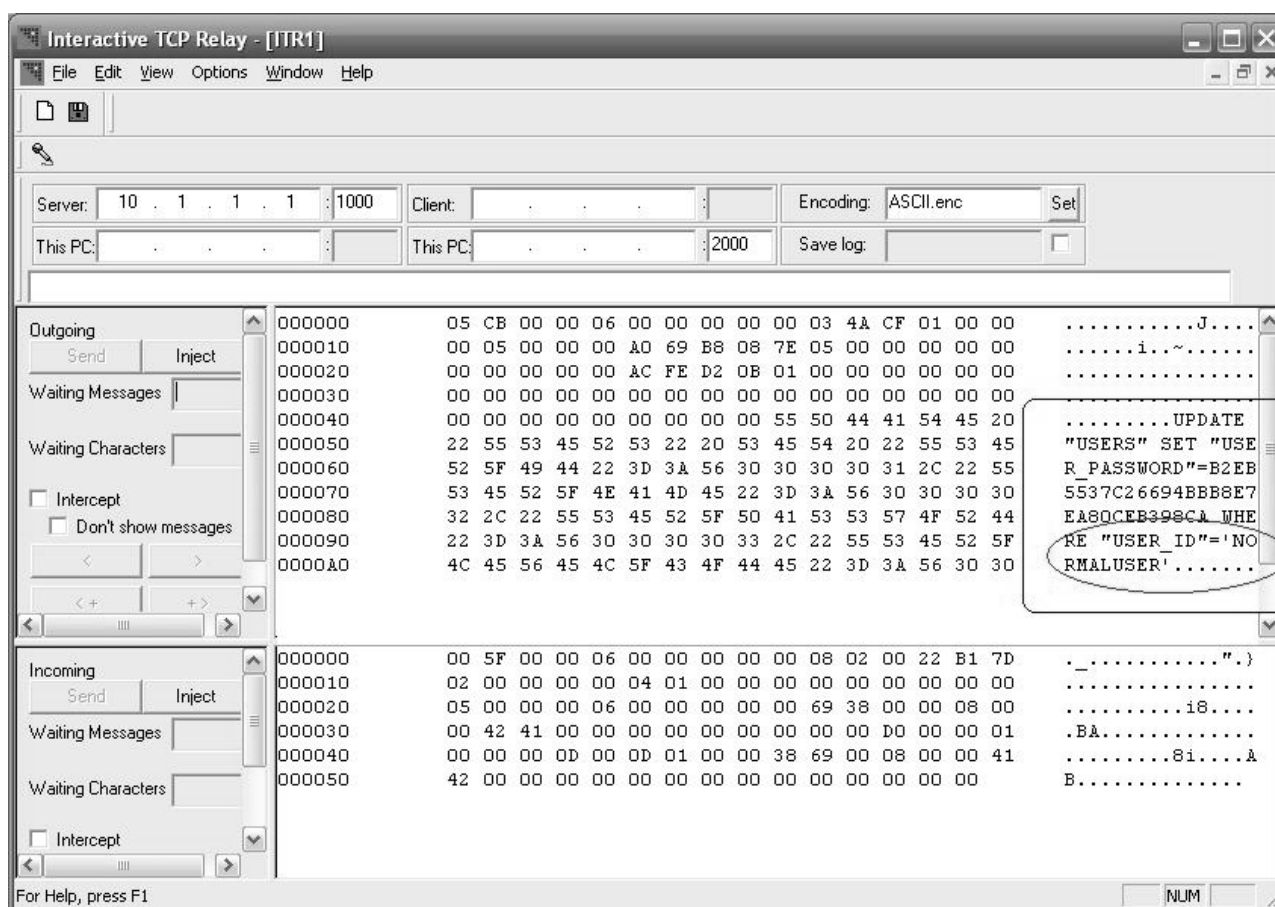


Figure 5: ITR as Interceptor

A malicious low privileged user can intercept the traffic using ITR; modify the query to reflect the reference attribute of high privilege user `USER_ID`. The database will update the password field of high privileged user with the supplied password of the malicious low privileged user. The malicious user can now log in to the application as high privileged user and view/modify important information.

In the following example a malicious user has modified the `USER_ID` portion of the `UPDATE` statement by replacing `NORMALUSER` with `ADMIN` and updated the password of the `ADMIN` with a known password hash.

A snapshot of impersonation is shown below.

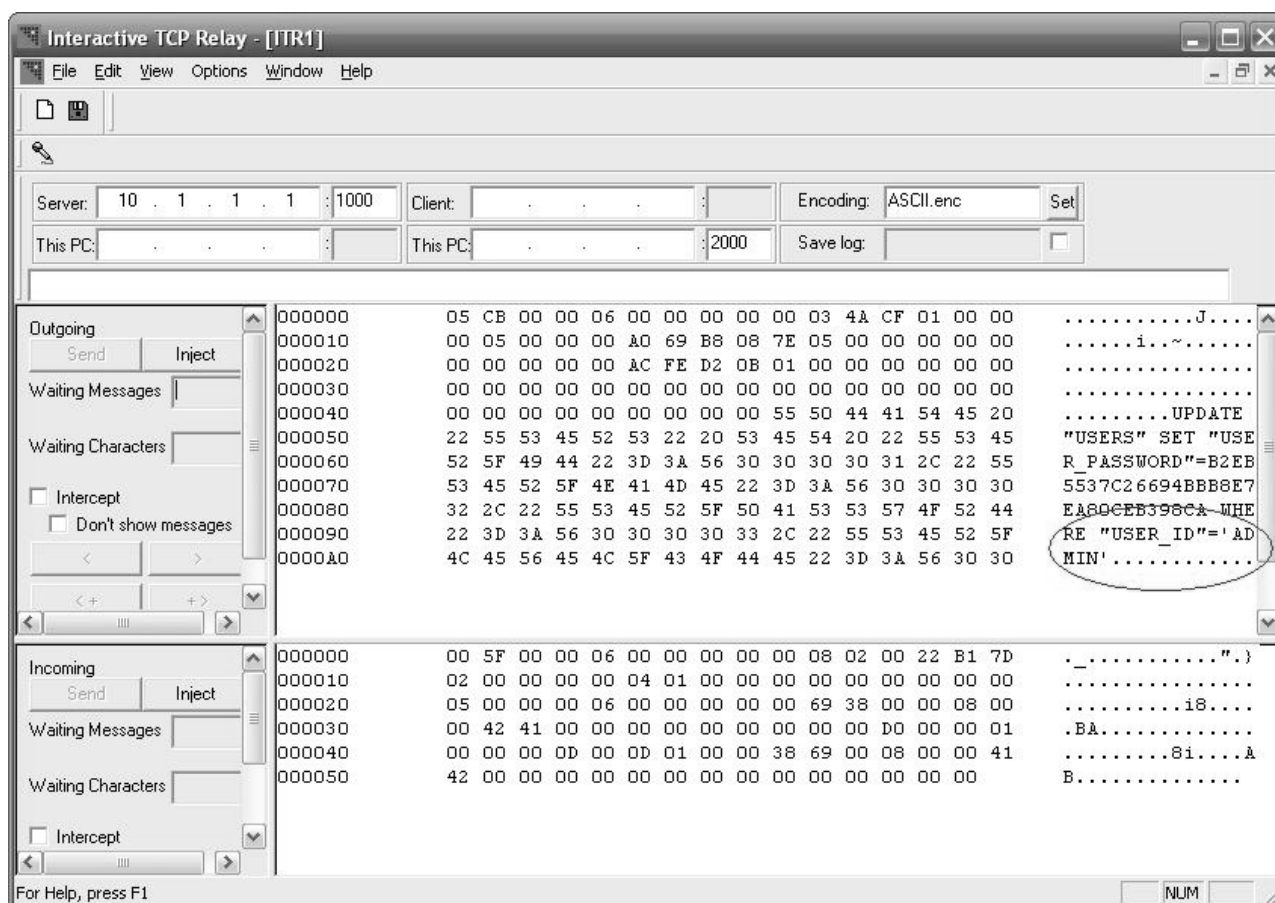


Figure 6: ITR as Interceptor for Impersonation

Solution

To solve this issue, database listener service should not run at default port. It can run at a non guessable port other than the default one. Client server traffic should be encrypted to protect from man in the middle attack. In three tier architecture this can be achieved by maintaining session ID between client user interface and application server. Physical access to the application server should be restricted and only application server should be permitted to communicate with the database. Application server should maintain and keep track of every transaction with highly random & non guessable session IDs.

Mitigating Risks

We have discussed a few critical vulnerabilities associated with a thick client application. To address these issues secure application architecture should be designed. The following security best practices can be considered during development as described in above sections.

- § Thick client application can be conceptualized as three tier architecture. Application user interface (UI) will reside at 1st tier. Application server which will perform the entire processing job will reside at 2nd tier. Database server will come at 3rd tier to store data.
- § Application UI to Application Server and Application Server to Database server traffic should be protected to maintain confidentiality and integrity. Confidentiality can be maintained using encryption algorithm like DES or 3DES and integrity by MD5. SSL can also be used to achieve the same.
- § Application should validate all user inputs for length, special characters & code.
- § Application should have provision to enforce proper authorization level to users on a need to know basis.
- § Adequate audit and logging feature should be present and enabled in the application to log all transaction events including login attempts.
- § Application should not store sensitive information like user password in computer memory, files, registry or database in clear text format. It should rather be stored in one way hash format. Critical file and registry permission must be secured. Application should clear the data in the memory after its usage.
- § Application should have the provision to enforce strong password policy to the users.
- § Application must use strong authentication technique. In this 3-tier structure all the user credentials can be verified at application server itself. It must not pass any credential like actual password to UI.
- § Application should uniquely identify and maintain sessions. Session IDs used should be random and unbreakable.
- § Application should handle the errors without disclosing critical system information.
- § Database should not run on a default port. A non guessable port should be used to connect the database.
- § Database, application server and OS should be configured with secure setting and updated with the latest software or patch levels.
- § Database should accept the connections only from the application server.