SySS
THE PENTEST EXPERTS.

IT SECURITY KNOW-HOW

Moritz Bechler

Oracle Native Network Encryption

Breaking a Proprietary Security Protocol

December 2021

Schaffhausenstraße 77, 72072 Tübingen, Germany
+49 (0)7071 - 40 78 56-0
info@syss.de
www.syss.de

1 Introduction

To protect the network communication between database clients and servers, Oracle offers two different options, an SSL/TLS-based as well as a proprietary security protocol called "Native Network Encryption" (NNE). These protocols both are meant to provide confidentiality and cryptographic integrity protection.

While SSL/TLS is standardized, well understood, and has undergone security analysis for decades, there is no public technical documentation of NNE and SySS GmbH is not aware of any published analysis results.

Originally part of the "Advanced Networking Option", both transport security and additional authentication mechanisms became part of the general feature set starting with Oracle Database version 12.1. Prior to that, they had to be licensed separately.

From a user and administrator perspective, NNE's easier setup may appear beneficial, as no certificates need to be configured, and therefore is often chosen over TLS.

Interested whether the promised security properties hold and live up to modern standards, SySS GmbH analyzed the cryptographic protocol for security issues.

2 Protocol Analysis

SySS GmbH analyzed the protocol based on the published client code and observed client and server behavior. Based on the gathered protocol information, a proxy server to manipulate various aspects of the data exchanges was implemented in Python using Scapy[1]. As this implementation is quite crude and incomplete regarding the application level protocol, it is not released at this point.

The following description reflects the protocol as it was before the changes made to address the findings reported to Oracle in our corresponding security advisories [1] and [2].

2.1 Parameter negotiation

The protocol stack is cleanly divided into several layers. The initial protocol negotiation traffic has multiple phases which correspond to these layers and are briefly described in this section.

2.1.1 Transparent Network Substrate (TNS)

This first layer negotiates basic network protocol parameters. It handles listener selection for clustering/failover and aliasing. NNE support/usage is also signaled on this layer. However, recent server versions enforce its usage and the Java Thin client does not even support non-NNE connections. Figure Figure 1 shows the initial exchange, captured in Wireshark.

[1] Scapy: `https://scapy.net/`

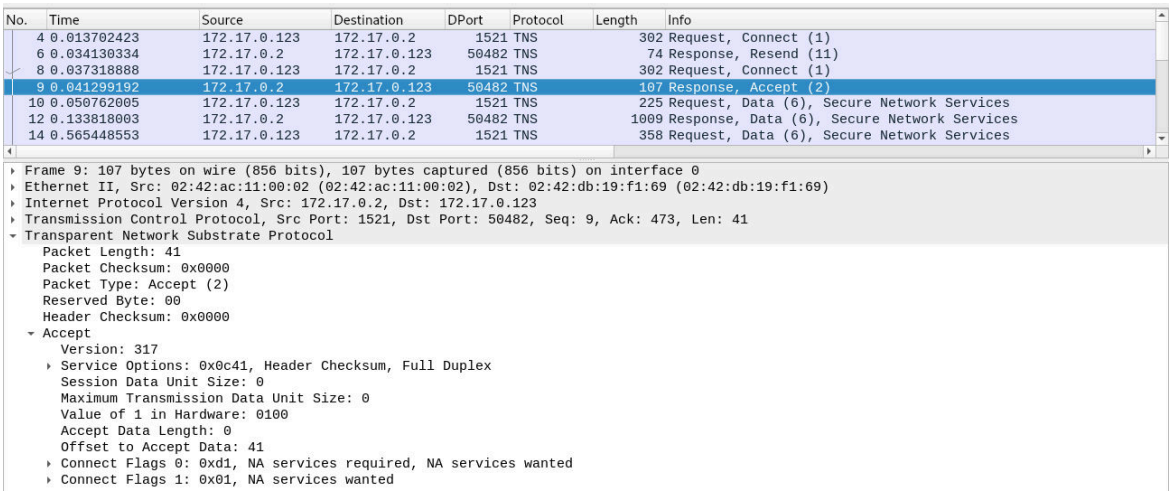

Figure 1: Initial TNS handshake shown in Wireshark

2.1.2 Native Network Encryption (NNE)

The following handshake phase negotiates the security protocol algorithms and parameters. The client first announces cryptographic algorithms it supports, the server then picks its preferred one. If the client configuration disables integrity and encryption, empty algorithm fields will be sent. Listings 2.1, 2.2 and 2.3 show a typical exchange with a strong configuration. They were captured and decoded using the developed custom proxy server.

```
###[ Secure Network Services Req ]###
   dataId    = 0xdeadbeef
   dataLength= 149
   clientVersion= 0
   numServices= 4
   unknown1  = 0
   \services  \
    |###[ Service ]###
    |  serviceId = supervisor
    |  numParameters= 3
    |  unknown1  = 0
    |###[ SupervisorReq ]###
    |     version   = 13000000
    |     S         = 0000780195BFDDFF
    |     T         = DEADBEEF00030000000400040001000100002
    |###[ Service ]###
    |  serviceId = authentication
    |  numParameters= 3
    |  unknown1  = 0
    |###[ AuthenticationReq ]###
    |     version   = 13000000
    |     S         = E0E1
    |     T         = FCFF
    |###[ Service ]###
    |  serviceId = encryption
    |  numParameters= 2
    |  unknown1  = 0
    |###[ EncryptionReq ]###
    |     version   = 13000000
    |     algos     = (,AES256,RC4_256,AES192,3DES168,AES128,RC4_128,
```

```
              3DES112,RC4_56,DES56C,RC4_40,DES40C)
    |###[ Service ]###
    |  serviceId = integrity
    |  numParameters= 2
    |  unknown1  = 0
    |###[ IntegrityReq ]###
    |     version   = 13000000
    |     algos     = (,SHA1,MD5,SHA521,SHA256,SHA384)
```

Listing 2.1: Initial SNS handshake (client → server)

Depending on the configuration, the server may decide to enable integrity and/or encryption. If disabled, an empty algorithm field will be returned. Within the same exchange, the Diffie-Hellman algorithms parameters are exchanged. The server specifies a Diffie-Hellman group and generator to be used.

```
###[ Secure Network Services Resp ]###
   dataId    = 0xdeadbeef
   dataLength= 933
   clientVersion= 0
   numServices= 4
   unknown1  = 0
   \services  \
    |###[ Service ]###
    |  serviceId = supervisor
    |  numParameters= 3
    |  unknown1  = 0
    |###[ SupervisorResp ]###
    |     version   = 12000000
    |     n31       = 001F
    |     T         = DEADBEEF00030000000200040001
    |###[ Service ]###
    |  serviceId = authentication
    |  numParameters= 2
    |  unknown1  = 0
    |###[ AuthenticationResp ]###
    |     version   = 12000000
    |     algo      = FBFF
    |###[ Service ]###
    |  serviceId = encryption
    |  numParameters= 2
    |  unknown1  = 0
    |###[ EncryptionResp ]###
    |     version   = 12000000
    |     algo      = AES128
    |###[ Service ]###
    |  serviceId = integrity
    |  numParameters= 8
    |  unknown1  = 0
    |###[ IntegrityResp ]###
    |     version   = 12000000
    |     algo      = SHA256
    |     len1      = 0800
    |     len2      = 0800
    |     generator = [...]002
    |     prime     = <2048-bit prime>
    |     public    = 0[...]1
    |     data      = 666F6F206261722062617A206261742071757578
```

Listing 2.2: Server SNS response (server → client)

If encryption and/or integrity was chosen and therefore a shared key needs to be agreed upon, the client makes another request to submit its Diffie-Hellman public value/key. Then both parties are able to compute the key.

```
###[ Secure Network Services Req ]###
   dataId    = 0xdeadbeef
   dataLength= 281
   clientVersion= 0
   numServices= 1
   unknown1  = 0
   \services  \
    |###[ Service ]###
    |  serviceId = integrity
    |  numParameters= 1
    |  unknown1  = 0
    |###[ IntegrityReq ]###
    |     public    = B[...]2
```

Listing 2.3: Client SNS response (client → server)

No validation of the negotiated parameters occurs later in the protocol, therefore downgrade attacks within the permitted configuration parameters will go undetected.

If enforced on neither the client nor on the server side by configuration, an attacker in a man-in-the-middle position can remove the algorithms from the initial client request. The server will accept that, and no encryption and integrity protection will be used. Configurations need to enforce encryption and integrity at least on one side, preferably on both.

If the algorithms are not manually restricted on the client/server side by configuration, a man-in-the middle attacker also can change the list of supported algorithms sent by the client. That weaker algorithm will then be used, even though both parties actually support stronger ones.

Even with the patches that resulted from this analysis, the default configuration permits all supported cryptographic algorithms to be used. This includes 40-bit (!) key length truncated/export-grade RC4 and DES ciphers. Such a key could conceivably be cracked with little effort.

To mitigate against these downgrade attacks, SySS GmbH recommends using the client and server configuration shown in Listings 2.4 and 2.5. Encryption and checksum algorithms should explicitly be set to strong ones.[2]

```
SQLNET.ENCRYPTION_CLIENT = REQUIRED
SQLNET.CRYPTO_CHECKSUM_CLIENT = REQUIRED
SQLNET.ENCRYPTION_TYPES_CLIENT = (AES128,AES256)
SQLNET.CRYPTO_CHECKSUM_TYPES_CLIENT = (SHA256,SHA384,SHA512)
SQLNET.ALLOW_WEAK_CRYPTO = FALSE
```

Listing 2.4: Recommended client configuration (`sqlnet.ora`)

```
SQLNET.ENCRYPTION_SERVER = REQUIRED
SQLNET.CRYPTO_CHECKSUM_SERVER = REQUIRED
SQLNET.ENCRYPTION_TYPES_SERVER = (AES128,AES256)
SQLNET.CRYPTO_CHECKSUM_TYPES_SERVER = (SHA256,SHA384,SHA512)
SQLNET.ALLOW_WEAK_CRYPTO_CLIENTS = FALSE
```

Listing 2.5: Recommended server configuration (`sqlnet.ora`)

[2] Setting `ALLOW_WEAK_CRYPTO=FALSE` in new version should make this redundant.

2.1.3 Application level

The next stage performs negotiation of application protocol details, including basic client/server capabilities and data representation.

This exchange is already encrypted and integrity-protected according to the previously negotiated parameters and keys derived from the Diffie-Hellman key exchange. As noted in Section 2.4, Diffie-Hellman alone cannot prevent active man-in-the-middle attacks. Therefore, the data exchanged here can be inspected and manipulated by an active attacker.

2.1.4 Oracle logon

The authentication exchange follows, using the same encryption and integrity protection as the previous stage.

The proprietary O5Logon challenge response protocol avoids disclosure of the plaintext password and should protect against replay attacks. Previously, it was found to be vulnerable to offline password cracking[3], but it appears that the protocol level flaw has been fixed. Further analysis of the logon protocol was not part of this research effort.

The individual parameters are encoded using a key-value format. Apart from the challenge response data, various other information is exchanged. Listings 2.6 through 2.9 show an example of an authentication exchange, again intercepted using the developed custom proxy server.

As with the application level exchange, this data communication can be observed and manipulated by an active attacker. As long as the authentication protocol itself is suitably protected, this should not be a major issue,[4] and avoiding it in a design would likely require a more involved key exchange algorithm including authentication, for example the Secure Remote Password protocol (SRP).

Regarding O5Logon and Kerberos, that assumption seems to be in line with the protocol's respective design goals. However, for the RADIUS-based authentication mode, this may depend on the exact mechanisms in use. RADIUS authentication should therefore only be used with TLS protection. The RADIUS scenario was not further analyzed, though.

```
###[ TTIoauthenticate ]###
     funCode   = 118
     nextSeq   = 1
###[ OSESSKEY ]###
        userPtr   = (1,)
        userLen   = 6
        logonMode = 1
        kvPtr     = (1,)
        kvLen     = 5
        unkPtr    = (1,)
        unkPtr2   = (1,)
        user      = 'SYSTEM'
        \kv        \
         |###[ KVPair ]###
         |  keyPtr    = 13
         |  key       = b'AUTH_TERMINAL'
         |  valuePtr  = 7
         |  value     = b'unknown'
         |  unknown   = 0
         |###[ KVPair ]###
```

[3] Compare CVE-2012-3137 — http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-3137.
[4] Apart from some information leakage.

```
         |  keyPtr    = 15
         |  key       = b'AUTH_PROGRAM_NM'
         |  valuePtr  = 16
         |  value     = b'JDBC Thin Client'
         |  unknown   = 0
[...]
```

Listing 2.6: Decoded OSESSKEY request (client → server)

```
###[ RPA ]###
     outNbPairs= 3
     \nbPairs   \
      |###[ KVPair ]###
      |  keyPtr    = 12
      |  key       = b'AUTH_SESSKEY'
      |  valuePtr  = 96
      |  value     = b'1EF63A15F672C6FFC[..]'
      |  unknown   = 0
      |###[ KVPair ]###
      |  keyPtr    = 13
      |  key       = b'AUTH_VFR_DATA'
      |  valuePtr  = 20
      |  value     = b'2324DF8DD3FA0E470ED8'
      |  unknown   = 6949
      |###[ KVPair ]###
      |  keyPtr    = 26
      |  key       = b'AUTH_GLOBALLY_UNIQUE_DBID\x00'
      |  valuePtr  = 32
      |  value     = b'2C73B05B0AC21DAD7AB9EA794FA378E6'
      |  unknown   = 0
[...]
```

Listing 2.7: Decoded OSESSKEY response (server → client)

```
###[ TTIoauthenticate ]###
     funCode   = 115
     nextSeq   = 2
###[ OAUTH ]###
        userPtr   = (1,)
        userLen   = 6
        logonMode = 257
        kvPtr     = (1,)
        kvLen     = 14
        unkPtr    = (1,)
        unkPtr2   = (1,)
        user      = 'SYSTEM'
        \kv        \
         |###[ KVPair ]###
         |  keyPtr    = 13
         |  key       = b'AUTH_PASSWORD'
         |  valuePtr  = 64
         |  value     = b'95C8E3F1[...]'
         |  unknown   = 0
         |###[ KVPair ]###
         |  keyPtr    = 22
         |  key       = b'AUTH_PBKDF2_SPEEDY_KEY'
         |  valuePtr  = 160
         |  value     = b'4A60662F013D[...]'
```

```
         |  unknown   = 0
[...]
```

Listing 2.8: Decoded `OAUTH` request (client → server)

```
###[ RPA ]###
     outNbPairs= 45
     \nbPairs   \
      |###[ KVPair ]###
      |  keyPtr    = 19
      |  key       = b'AUTH_VERSION_STRING'
      |  valuePtr  = 12
      |  value     = b'- Production'
      |  unknown   = 0
      |###[ KVPair ]###
      |  keyPtr    = 16
      |  key       = b'AUTH_VERSION_SQL'
      |  valuePtr  = 2
      |  value     = b'24'
      |  unknown   = 0
[...]
```

Listing 2.9: Decoded `OAUTH` response (server → client)

2.2 Weak encryption mode

AES data encryption is performed in CBC mode with an all-zeros initialization vector (IV). This scheme leaks some information about the encrypted data, as messages with the same prefix result in the same prefix of the encrypted data. Its use is generally discouraged and could enable other cryptographic attacks.

The other encryption algorithms and key derivation routines were not analyzed in detail, as all these ciphers are considered weak and should no longer be used anyways.

2.3 Weak integrity key

When integrity protection is enabled, a cryptographic checksum is appended to each request/response packet. The checksum is calculated using the negotiated cryptographic hash algorithm over the original packet data and a per-packet secret key. If encryption is enabled as well, the checksummed plaintext data is then encrypted with the session master key[5].

The sequence of packet keys is derived using one of two key generation schemes: an AES-based one for SHA-2 family hashes, and an RC4-based one for legacy ones. These generators rely on iterative encryption of a block or the keystream produced by the cipher respectively. A separate generator for each communication direction is used and seeded by a primary generator.

Listing 2.10 shows the initialization and the client-to-server key generation routine as Python code. A similar construction is used for rekeying.

The generator initial state completely depends on the session master key `key` and `state['iv']`. The `iv` is the `data` value exchanged during SNS negotiation in Listing 2.2 and therefore transmitted in cleartext.

[5] First n bytes of the computed shared key, where n is the algorithm key length. Diffie-Hellman shared secret, potentially with authentication key folded in.

However, the key input is truncated to 5 bytes length, even for the modern algorithm. Therefore, only 40 bits (5 bytes) of secret information are necessary to recover the complete generator state, significantly weakening the scheme. This is likely in the range to allow for practical brute-force attacks, at least when only integrity protection is applied and a long-running session is attacked.

```
class AESIntegrityKeyGen(IntegrityKeyGen):
    def __init__(self, key, state):
        mk = key[0:5]+ b'\xFF' + b'\x00' * 10
        self.m = AES.new(mk, AES.MODE_CBC, iv=state['iv'][0:16])
        self.ms = b'\x00'*32
        self.ms = s = self.m.encrypt(self.ms)
        self.m = AES.new(s[0:16], AES.MODE_CBC, iv=s[16:32])

        k1 = s[0:5] + b'\xB4' + s[6:16]
        self.s2c = AES.new(k1, AES.MODE_CBC, iv=s[16:32])
        self.s2cs = b'\x00' * 32

        k2 = s[0:5] + b'\x5A' + s[6:16]
        self.c2s = AES.new(k2, AES.MODE_CBC, iv=s[16:32])
        self.c2ss = b'\x00' * 32

    def genc2s(self):
        self.c2ss = k = self.c2s.encrypt(self.c2ss)
        return k
```

Listing 2.10: Original AES key generator initialization implemented in Python, only 40 bits (5 bytes) of the key are used

2.4 Authentication key fold-in

While the Diffie-Hellman key exchange allows establishing a shared secret, even if the exchange is observed by an attacker, neither party can be sure that the other party is the intended communication partner. Without further verification, an active attacker can perform two independent key exchanges, one with each party. This way, two different keys are established, both of which are known to the attacker who can then translate between the two original parties. This is a well-known man-in-the-middle attack pattern against Diffie-Hellman key exchanges.

Therefore, an additional mechanism to properly identify the party that provided the Diffie-Hellman key parameters needs to be implemented, e.g. cryptographic signatures, as used in TLS, or a shared secret.

Oracle's documentation states that NNE addresses such attacks by mixing in a shared secret derived from the authentication protocol.

> You can use Authentication Key Fold-in to defeat a possible third-party attack (historically called the man-in-the-middle attack) on the Diffie-Hellman key negotiation algorithm key negotiation. It strengthens the session key significantly by combining a shared secret, known only to the client and the server, with the original session key negotiated by Diffie-Hellman.
>
> The client and the server begin communicating using the session key generated by Diffie-Hellman. When the client authenticates to the server, they establish a shared secret that is only known to both parties. Oracle Database combines the shared secret and the Diffie-Hellman session key to generate a stronger session key designed to defeat a man-in-the-middle attack.
>
> – https://docs.oracle.com/database/121/DBSEG/asoconfg.htm

This authentication key mentioned here is a result of the O5Logon challenge response authentication. The `AUTH_SESSKEY` values are exchanged in `TTIoauthenticate` calls and decrypted using a key derived from the user password (hash). Therefore, it is a secret shared between the legitimate server and the client. An attacker who does not know the user password should be unable to determine the key used for encryption and integrity protection.

However, SySS GmbH found out that no such fold-in is performed by the JDBC Thin client variant, and a classic man-in-the-middle attack succeeded by just replacing the Diffie-Hellman (DH) parameters on either side. The same behavior was later achieved by forcefully skipping the authentication key fold-in using a debugger with an OCI client.

This server-side behavior is broken from a security perspective, as packets encrypted with the "wrong" key are accepted and processed by the server. It looks like a fallback to the other insecure key is performed when processing with the correct key fails.

That broken behavior enables a number of attacks that break the protocol's fundamental security promises:

- A trivial man-in-the-middle attack is successful against the JDBC Thin client, as both parties use and accept the original keys.
- A man-in-the-middle attacker can negotiate an unprotected connection with the client and a secured one with the server. After authentication, the man-in-the-middle attacker can still use the original DH-derived keys when communicating with the server. Therefore, an attacker does not need any information about the user credentials to successfully launch the attack. This vector is mitigated when encryption/integrity is enforced on the client side.
- Most notably, it is also possible to perform an attack to hijack an authenticated connection. The attacker can launch a man-in-the-middle attack on the DH exchange. Then he waits for the authentication exchange to complete (server responds to `OAUTH` call). At this point, the client can be ignored/dropped.[6] However, the connection with the server is successfully authenticated, and the server still accepts the original, DH-derived key material for integrity and encryption purposes. The man-in-the-middle attacker can simply continue to use these keys to encrypt requests and decrypt responses. Access to the database as the original victim user is gained.

This last attack is successful independent of server and client configuration and affects all client types.

3 Conclusion

This research identified a fundamental flaw in a proprietary security protocol of a widely used product that has been around for many years. This once again shows that such protocols need to be scrutinized, even though their typically undocumented nature makes this an expensive, time-consuming exercise in reverse engineering.

In general, relying on proven, standardized TLS-based security instead of the custom proprietary protocol is advised. This analysis should not be considered an in-depth crypto analysis, and more subtle security flaws may be present.

The identified issues were responsibly disclosed to Oracle in March 2021 and addressed in the July 2021 Critical Patch Update, although some related patches were slightly delayed to August 7. It is tracked as `CVE-2021-2351`[1].

[6] At this point, an OCI client will no longer accept responses, as the wrong key is used.

[1] See `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-2351`

The announcement and associated support document for the official fix is somewhat obfuscating the core vulnerabilities and the extent of the fixes applied, as it is mostly talking about the restriction of legacy cipher usage implemented at the same time.

The documentation, however, reveals some more details for the new parameter `SQLNET.ALLOW_WEAK_CRYPTO`:

> If you set this parameter to FALSE, then you can specify only supported algorithms so that clients and servers can communicate in a fully patched environment. The server enforces key fold-in for all Kerberos and JDBC thin clients. This configuration strengthens the connection between clients and servers by using strong native network encryption and integrity capabilities.
>
> – `https://docs.oracle.com/en/database/oracle/oracle-database/12.2/netrf/parameters-for-the-sqlnet-ora-file.html`

If both the client and server in use are patched, a new security protocol version is negotiated. That version features various improvements addressing the security weaknesses presented here:

- No more truncation/weakening of the integrity key – now, 120 instead of 40 bits of secret key material are used.
- No more static all-zero IVs – part of the DH secret is now used as a per-session IV for encryption.
- Proper authentication key fold-in is performed by the JDBC Thin client.

In addition and most significantly, the authentication key fold-in can be enforced on the server side. As the JDBC Thin client up to this point relied on the original, insecure server behavior, fixing the issue requires the upgrade of all such clients, before this option can be enabled. It should be noted that, while the full man-in-the-middle attack scenario could have been mitigated (leaving just the hijacking attack) in the Java client even for older servers by always performing the authentication key fold-in, this was not implemented.

SySS GmbH strongly recommends applying the supplied patches and upgrades, including the necessary JDBC client updates and then setting the new configuration option `SQLNET.ALLOW_WEAK_CRYPTO_CLIENTS=FALSE`[2]. Connections to database servers without this setting are still vulnerable and authenticated connections can be hijacked by a man-in-the-middle attacker. Also, if NNE is used, confidentiality and integrity should be enforced by configuration on both the client and the server side.[3]

[2] Also `SQLNET.ALLOW_WEAK_CRYPTO=FALSE` to disable legacy ciphers on the client side.

[3] Settings: `SQLNET.ENCRYPTION_(SERVER|CLIENT) = REQUIRED` and `SQLNET.CRYPTO_CHECKSUM_(SERVER|CLIENT) = REQUIRED`

References

[1] Moritz Bechler, SySS Advisory SYSS-2021-061, https://www.syss.de/fileadmin/dokumente/Publikationen/Advisories/SYSS-2021-061.txt, 2021 1

[2] Moritz Bechler, SySS Advisory SYSS-2021-062, https://www.syss.de/fileadmin/dokumente/Publikationen/Advisories/SYSS-2021-062.txt, 2021 1

THE PENTEST EXPERTS

SySS GmbH Tübingen Germany +49 (0)7071 - 40 78 56-0 info@syss.de

www.syss.de