# Evil Corp [UNSOLVED :-( ]

Hack.lu
2019

Mathias Tausig (TwentyOneCool)

# Task I

You were called by the Incident Response Team of Evil Corp, the urgently need your help. Somebody broke into the main server of the company, bricked the device and stole all the files! Nothing is left! This should have been impossible. The Hacker used some secret backdoor to bypass authentication. Without the knowledge of the secret backdoor other servers are at risk as well! The Incident Response Team has a full packet capture of the incident and performed an emergency cold boot attack on the server to retrieve the contents of the memory (its a really important server, Evil Corp is always ready for such kinds of incidents. However they were unable to retrieve much information from the RAM, what's left is only some parts of the "key_block" of the TLS server. Can you help Evil Corp to analyze the exploit the attacker used? (Flag is inside of the attackers' secret message).

# Task II

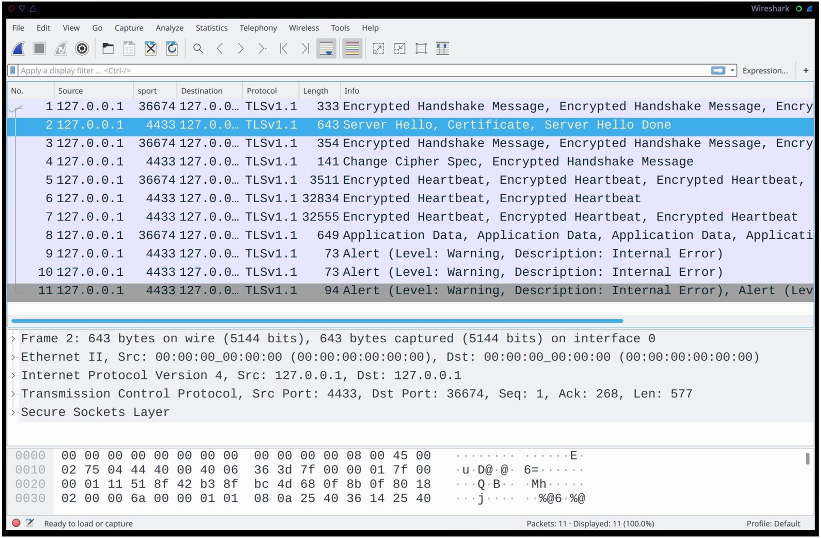
## Keyblock

TT = Could not recover

```
6B 4F 93 6A TT TT TT TT TT TT 00 D9 F2 9B 4C B0
2D 88 36 CF B0 CB F1 A6 7B 53 B2 00 B6 D9 DC EF
66 E6 2C 33 5D 89 6A 92 ED D9 7C 07 49 57 AD E1
TT TT TT TT TT TT TT TT 56 C6 D8 3A TT TT TT TT
TT TT TT TT TT TT TT TT 94 TT 0C EB 50 8D 81 C4
E4 40 B6 26 DF E3 40 9A 6C F3 95 84 E6 C5 86 40
49 FD 4E F2 A0 A3 01 06
```

File `evilcorp.pcapng`

# Capture


Wireshark
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>
Expression...
No. Source sport Destination Protocol Length Info
1 127.0.0.1 36674 127.0.0... TLSv1.1 333 Encrypted Handshake Message, Encrypted Handshake Message, Encry
2 127.0.0.1 4433 127.0.0... TLSv1.1 643 Server Hello, Certificate, Server Hello Done
3 127.0.0.1 36674 127.0.0... TLSv1.1 354 Encrypted Handshake Message, Encrypted Handshake Message, Encry
4 127.0.0.1 4433 127.0.0... TLSv1.1 141 Change Cipher Spec, Encrypted Handshake Message
5 127.0.0.1 36674 127.0.0... TLSv1.1 3511 Encrypted Heartbeat, Encrypted Heartbeat, Encrypted Heartbeat,
6 127.0.0.1 4433 127.0.0... TLSv1.1 32834 Encrypted Heartbeat, Encrypted Heartbeat
7 127.0.0.1 4433 127.0.0... TLSv1.1 32555 Encrypted Heartbeat, Encrypted Heartbeat, Encrypted Heartbeat
8 127.0.0.1 36674 127.0.0... TLSv1.1 649 Application Data, Application Data, Application Data, Applicati
9 127.0.0.1 4433 127.0.0... TLSv1.1 73 Alert (Level: Warning, Description: Internal Error)
10 127.0.0.1 4433 127.0.0... TLSv1.1 73 Alert (Level: Warning, Description: Internal Error)
11 127.0.0.1 4433 127.0.0... TLSv1.1 94 Alert (Level: Warning, Description: Internal Error), Alert (Lev
Frame 2: 643 bytes on wire (5144 bits), 643 bytes captured (5144 bits) on interface 0
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
Transmission Control Protocol, Src Port: 4433, Dst Port: 36674, Seq: 1, Ack: 268, Len: 577
Secure Sockets Layer
0000 00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00 ········ ······E·
0010 02 75 04 44 40 00 40 06 36 3d 7f 00 00 01 7f 00 ·u·D@·@· 6=······
0020 00 01 11 51 8f 42 b3 8f bc 4d 68 0f 8b 0f 80 18 ···Q·B·· ·Mh·····
0030 02 00 00 6a 00 00 01 01 08 0a 25 40 36 14 25 40 ···j···· ··%@6·%@
Ready to load or capture
Packets: 11 · Displayed: 11 (100.0%)
Profile: Default

# Skills

Skills needed for this challenge:

- Basic cryptography knowledge
- Basic TLS knowledge
- **RFC reading**

# Skills

Skills needed for this challenge:

- Basic cryptography knowledge
- Basic TLS knowledge
- **RFC reading**

**RFC 4346[1]**

The Transport Layer Security (TLS) Protocol Version 1.1
April 2006

[1] https://tools.ietf.org/html/rfc4346

# What is a key_block?

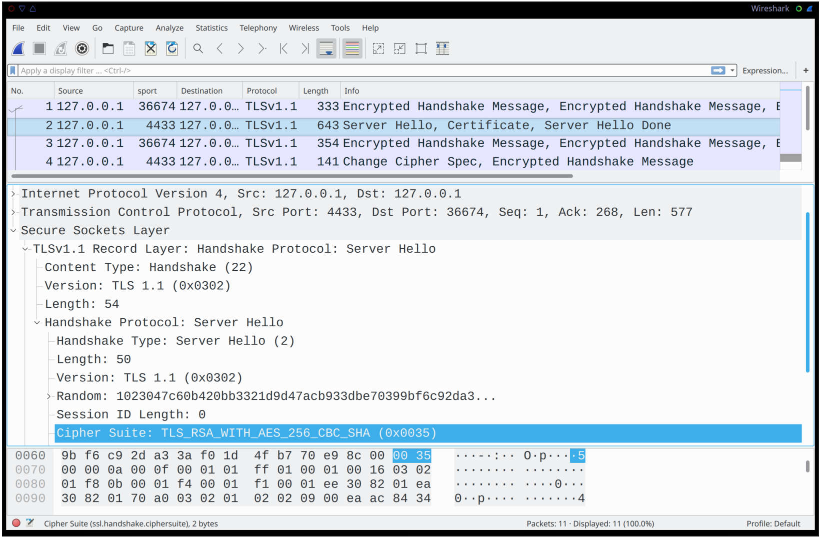
## 6.3. Key Calculation

*The Record Protocol requires an algorithm to generate keys, and MAC secrets from the security parameters provided by the handshake protocol.*
*The master secret is hashed into a sequence of secure bytes [...]*
*Then the **key_block** is partitioned as follows:*

```
client_write_MAC_secret[SP.hash_size]
server_write_MAC_secret[SP.hash_size]
client_write_key[SP.key_material_length]
server_write_key[SP.key_material_length]
```

SP = *SecurityParameters*

# SecurityParameters



# key_block I

## Security Parameters

- **Cipher:** AES-256 $\rightarrow$ 32 byte key
- **MAC:** HMAC-SHA1 $\rightarrow$ 20 byte hash size

## Available data

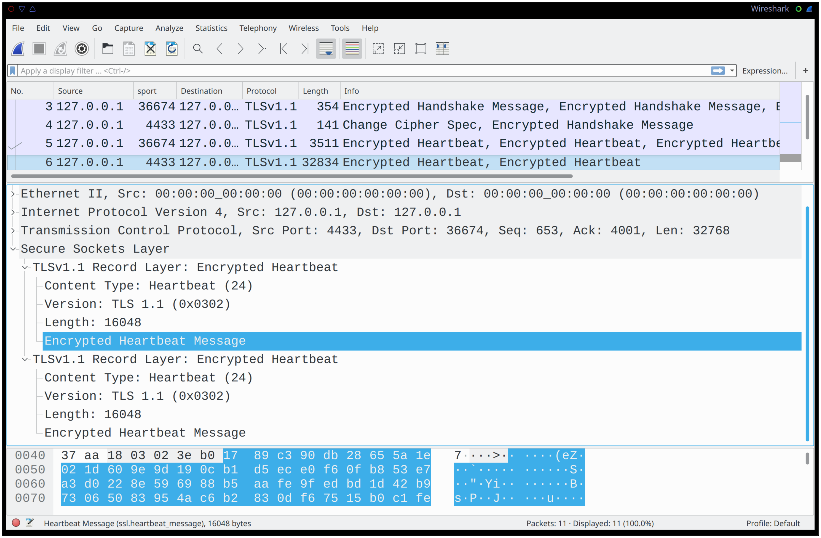
**6B 4F 93 6A TT TT TT TT TT TT 00 D9 F2 9B 4C B0**
**2D 88 36 CF** B0 CB F1 A6 7B 53 B2 00 B6 D9 DC EF
66 E6 2C 33 5D 89 6A 92 **ED D9 7C 07 49 57 AD E1**
**TT TT TT TT TT TT TT TT 56 C6 D8 3A TT TT TT TT**
**TT TT TT TT TT TT TT TT** 94 TT 0C EB 50 8D 81 C4
E4 40 B6 26 DF E3 40 9A 6C F3 95 84 E6 C5 86 40
49 FD 4E F2 A0 A3 01 06

# key_block II

## Secrets

- **client_write_MAC_secret:** 6B 4F 93 6A TT TT TT TT TT TT 00 D9 F2 9B 4C B0 2D 88 36 CF
- **server_write_MAC_secret:** B0 CB F1 A6 7B 53 B2 00 B6 D9 DC EF 66 E6 2C 33 5D 89 6A 92
- **client_write_key:** ED D9 7C 07 49 57 AD E1 TT TT TT TT TT TT TT TT 56 C6 D8 3A TT TT TT TT TT TT TT TT TT TT TT TT
- **server_write_key:** 94 TT 0C EB 50 8D 81 C4 E4 40 B6 26 DF E3 40 9A 6C F3 95 84 E6 C5 86 40 49 FD 4E F2 A0 A3 01 06

# Encrypted data



# TLS records

```
struct {
    ContentType type;
    ProtocolVersion version;
    uint16 length;
    select (CipherSpec.cipher_type) {
        case stream: GenericStreamCipher;
        case block: GenericBlockCipher;
    } fragment;
} TLSCiphertext;

block-ciphered struct {
    opaque IV[CipherSpec.block_length];
    opaque content[TLSCompressed.length];
    opaque MAC[CipherSpec.hash_size];
    uint8 padding[GenericBlockCipher.padding_length];
    uint8 padding_length;
} GenericBlockCipher;
```

# Plan

1. Brute force the *server_write_key*
2. Decrypt all messages sent by the server

# Plan

1. Brute force the *server_write_key*
   - Validate decryption
2. Decrypt all messages sent by the server

# Plan

1. Brute force the *server_write_key*
   - Validate decryption
     - Validate padding
     - Verify MAC
2. Decrypt all messages sent by the server

# Plan

1. Brute force the *server_write_key*
   - Validate decryption
     - Validate padding
     - Verify MAC

2. Decrypt all messages sent by the server

### 6.2.3. Record Payload Protection

The MAC is generated as:

```
HMAC_hash(MAC_write_secret, seq_num +
 TLSCompressed.type + TLSCompressed.version +
 TLSCompressed.length + TLSCompressed.fragment));
```

# Plan

1. Brute force the *server_write_key*
   - Validate decryption
     - Validate padding
     - Verify MAC

2. Decrypt all messages sent by the server

**6.2.3. Record Payload Protection**

The MAC is generated as:

```
HMAC_hash(MAC_write_secret, seq_num +
 TLSCompressed.type + TLSCompressed.version +
 TLSCompressed.length + TLSCompressed.fragment));
```

Problem: `seq_num` is implicit, not sent over the wire.

# Plan

1. Brute force the *server_write_key*
   - Validate decryption
     - Validate padding
     - ~~Verify MAC~~
2. Decrypt all messages sent by the server

# Pitfall

TLS padding $\neq$ PKCS#7 padding

# Pitfall

TLS padding $\neq$ PKCS#7 padding

**PKCS#7 padding**

`0c ff 3d 5b` $\rightarrow$ `0c ff 3d 5b 04 04 04 04`

# Pitfall

TLS padding $\neq$ PKCS#7 padding

**PKCS#7 padding**

`0c ff 3d 5b` $\rightarrow$ `0c ff 3d 5b 04 04 04 04`

**TLS padding**

`0c ff 3d 5b` $\rightarrow$ `0c ff 3d 5b 03 03 03 03`

# Decryption

```python
def remove_tls_padding(data):
    """Remove TLS padding which contains n bytes of
    value n-1 (n >= 1)"""
    padding_length = data[-1]
    padding = data[-(padding_length+1):]
    for padding_byte in padding:
        if(padding_byte != padding_length):
            raise ValueError("Bad padding")
    unpadded = data[:-(padding_length+1)]
    return unpadded
```

# Decryption

```
def decrypt_block(secret, block):
    try:
        iv = block[:16]
        ciphertext = block[16:]
        cipher = AES.new(secret, AES.MODE_CBC, iv)
        decryption = cipher.decrypt(ciphertext)

        plaintext = remove_tls_padding(decryption)
        # Last 20 bytes of the plaintext are the MAC
        plaintext = plaintext[:-20]
        # No compression is used
        return plaintext
    except ValueError:
        return None
```

# Decryption

```
decrypted = {}
for key_byte in range(256):
    key_cand = key[:byte_to_bruteforce] +
        bytes([key_byte])+key[byte_to_bruteforce+1 :]
    block = application_data[0]
    plaintext = decrypt_block(key_cand, block)
    if(plaintext):
        decrypted[key_candidate] = []
        decrypted[key_candidate].append(plaintext)
        # Valid decryption for first block
        # -> check if we can decrypt the next blocks
        for i in range(1, len(application_data)):
            block = application_data[i]
            plaintext = decrypt_block(key_cand,block)
            if(plaintext):
                decrypted[key_cand].append(plaintext)
```

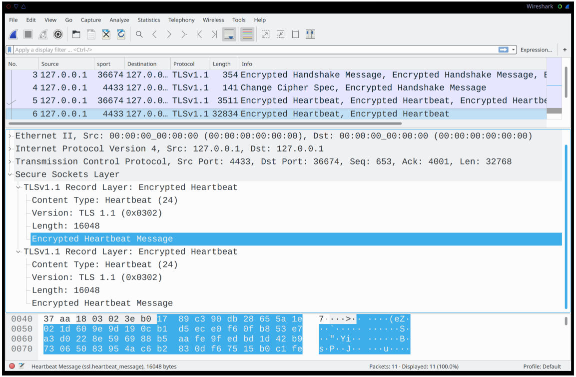
# Decryption

```
# The correct key should decrypt all blocks
valid_keys = list(filter(
    lambda k:len(decrypted[k])==len(application_data)
    decrypted.keys()))
if(len(valid_keys) != 1):
    print('ERROR: Key not found.')
    exit(1)

key = valid_keys[0]

print("Decrpytion worked for key '{0}':".format(key))
for i, plaintext in enumerate(decrypted[key]):
    print('## Record {0} plaintext ##\n{1}'
        .format(i, plaintext))
```

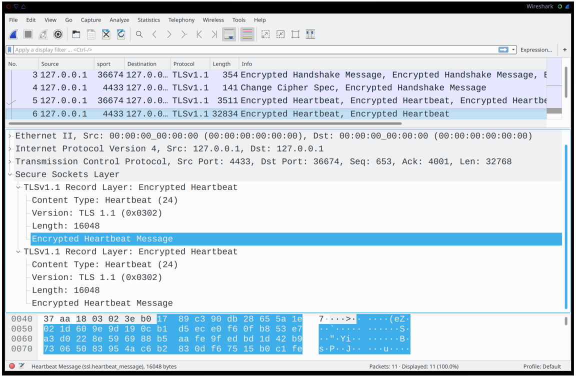
# What’s in it


```
[...]00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00-----BEGIN RSA PRIVATE KEY-----\nMIIB3gIBAA
JlAK2H8Iak4azSVdHXcySgXqfSUPKF86beNbnwfF0IOt1RZmd0J
bgz\nUyglXntWL5RNVcVv8IT0MW/cnj9bAJ/v1lAVpcoijJTj/T
XGq6g+pOIIAKNFSKo2\npdQOPHSWxlvbyGTo8WECAwEAAQJkJj9
5P2QmLb5qlgbj5SXH1zufBeWKb7Q4qVQd\nRTAkMVXYuWK7UZ9W
a9nYulyjvg9RoWOO+SaDNqhiTWKosQ+ZrvG3A1TDMcVZSkPx\nb
XCuhhRpp4j0T9levQi0s8tR1YuFzVFi8QIzANNLrgK2YOJiDlyu
78t/eVbBey4m\nuh2xaxvEd8xGX4bIBlTuWlKIqwPNxE8fygmv4
uHFAjMA0j7Uk1ThY+UCYdeCm4/P\neVqkPYu7jNTHG2TGr/B6hs
txyFpXBlq6MJQ/qPdRXLkLFu0CMwCf/OLCTQPpBiQn\ny5HoPRp
MNW4m0M4F46vdN5MaCoMUU+pvbpbXfYI3/BrTapeZZCNfnQIzAJ
7XzW9K\nj8cTPIuDcS/qpQvAiZneOmKaV5vAtcQzYb75cgu3BUz
NuyH8v2P/Br+RJmm5AjMA\njp9N+xdEm4dW51lyUp6boVU6fxZi
mfYRfYANU2bVFmbsSAU9jzjWb0BuXexKKcX7\nXGo=\n-----EN
D RSA PRIVATE KEY-----\n\x00\x00\x00\x00\x00\x00\x0
0\x00\x00[...]
```

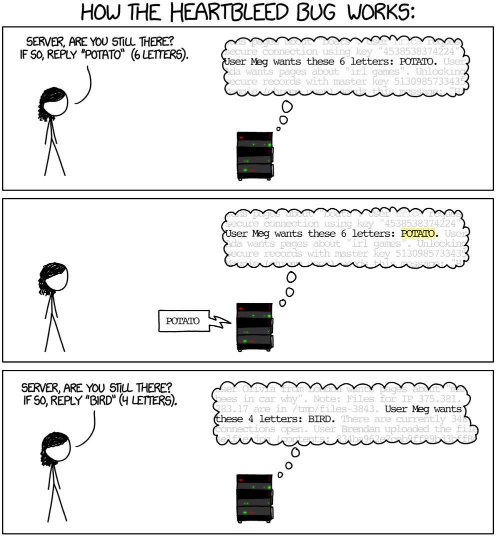
# Why is it there?



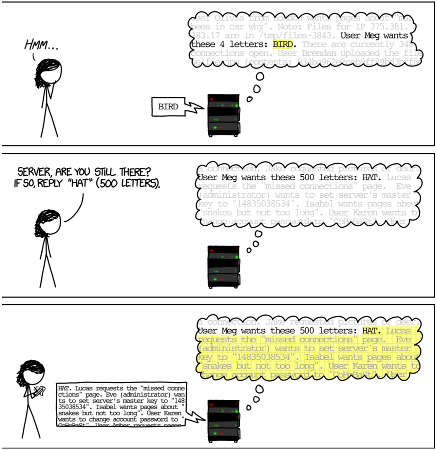
# Why is it there?




# Heartbleed I



# Heartbleed II




Source: `https://xkcd.com/1354/`

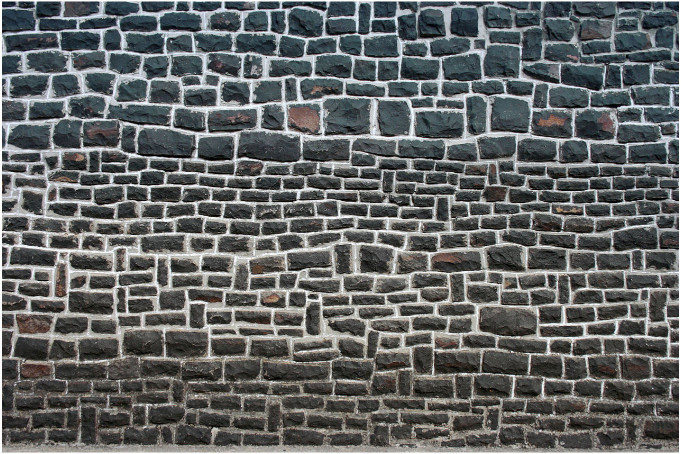
# Heartbleed III

## Facts

- *Heartbeat* is a TLS extension[2]
- Meant to *allow the usage of keep-alive functionality without performing a renegotiation*
- Buffer overread vulnerablility in the openssl implementation
- Published: April 2014
- Fixed: openssl 1.0.1g, 1.0.2-beta2
- Led to numerous openssl forks/reimplementations (libressl, BoringSSL, ...)
- CVE-2014-0160

[2] RFC 6520: Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension (2012)

# And now???



Source: CC-BY-SA, Takkk,
https://de.wikipedia.org/wiki/Datei:Basalt_house-wall_-_Matraszentistvan,_Hungary.jpg

# Ideas I

## Key Exchange

- RSA key exchange was used
- Private key is key for certificate used in the handshake
- This should allow us to decrypt whole conversation
- Unfortunately, we need the *ClientKeyExchange* message from the TLS handshake, and it's not in the capture

# Ideas II

**Missing keys**

Searching for the remaining unknown bytes of the *client_write_key* in the decrypted data was not successful.

# Ideas III

## Secure Renegotiation

- A second TLS extension is used in the handshake: Secure Renegotiation (RFC 5746: Transport Layer Security (TLS) Renegotiation Indication Extension, 2010)
- That's why the handshake messages sent by the client (containing the *ClientKeyExchange*) in the first frame are encrypted
- No decrypted handshake messages found
- Maybe the previous session keys are in the decrypted data?

# Ideas IV

## Finished

- We can decrypt the *Finished* message sent by the server at the end of the handshake
- It contains only the *verify_data* (RFC, 7.4.9, Finished)
- `verify_data = PRF(master_secret, finished_label, MD5(handshake_messages) + SHA-1(handshake_messages)) [0..11];`

# Ideas V

## Other data

- One more block with readable content found in the plaintext
  - (Part of) A certificate?
  - Looking for a DER encoded sequence (`0x8081`, `0x8082`) reveals another 800 bit public key

```
\xad_S\xb8hTG=0\x8a\x1c]/<\xa5\x08\xc0\x04\xday
\xd3\x07k\xab\x19;"‘\xab\x96\xaa8\xb6\xa6j\xae‘
\xe9\xdc\xe2\x89\xc2\xdb.\\\xbc\xd9\x98\x00021
\x120\x10\x06\x03U\x04\n\x0c\tEvil-corp1\x1c0
\x1a\x06\x03U\x04\x03\x0c\x13main-server.evil.
lu0 \x17\r190412131524Z\x18\x0f21190319131524Z
021\x120\x10\x06\x03U\x04\n\x0c\tEvil-corp1
\x1c0\x1a\x06\x03U\x04\x03\x0c\x13main-server.
evil.lu0\x81\x800\r\x06\t*\x86H\x86\xf7\r\x01
```

# Problems & Fixes

- Not upgrading openssl (and the rest of the system?) for 5 years $\rightarrow$ upgrade system
- Use of deprecated RSA key exchange in TLS handshake $\rightarrow$ use ECDHE
- (Use of deprecated TLS version $\rightarrow$ use TLS v1.3)
- (Use of deprecated cipher suite $\rightarrow$ use modern cipher suite with AEAD)

# The End!