Name: Jonathan Alter (ja3943)

Team Name: SendIt

Team Members: Spencer Yost (swy273), Jan Masztal (jm8512), Matthew Thibodeaux (mt4228)

CTF: RITSEC CTF 2021 (https://ctf.ritsec.club/challenges)

Challenge Name: IFTPP

Challenge Completed Date: 4/11/2021

Challenge Value: 500

Challenge Description: Dang that's a big ping

Challenge Author: -degenerat3

Challenge Category: Forensics

Challenge File: iftpp_challenge.pcap
https://ctf.ritsec.club/files/db6544c6a11ecbb6fd580dc2668b57e4/iftpp_challenge.pcap?token=eyJ1c2VyX2lkIjoxOTIsInRlYW1faWQiOjc5LCJmaWxlX2lkIjoxM30.YHM-AQ.nvyXuZs-krP4W7ZjVa_Xqv59WEU

Challenge Summary: The challenge provides a packet capture (PCAP) file for analysis. A simple gander at the capture (as well as a look at the description) shows that there is much more to the capture than initially meets the eye. The client uses an HTTP GET request to obtain the file `rfc.txt`. This file outlines a custom file transport protocol that runs atop ICMP Echo requests (a.k.a ping) dubbed Insecure File Transfer Protocol over Ping (IFTPP). The subsequent IFTPP traffic found in the data field of ICMP communication in the capture file can be analyzed using the provided RFC to extract and decrypt the file requested by the client (`flag.jpg`).

Challenge Walkthrough:

Getting your bearings:

11	0.001519	192.168.206.189	192.168.206.136	TCP	66
12	0.001600	192.168.206.136	192.168.206.189	HTTP	817
13	0.001608	192.168.206.189	192.168.206.136	TCP	66
14	0.001773	192.168.206.136	192.168.206.189	TCP	66
15	0.002145	192.168.206.189	192.168.206.136	TCP	66
16	0.002305	192.168.206.136	192.168.206.189	TCP	66
17	10.502035	192.168.206.189	192.168.206.136	ICMP	66
18	10.502652	192.168.206.136	192.168.206.189	ICMP	60
19	10.502737	192.168.206.189	192.168.206.136	ICMP	74
20	10.503020	192.168.206.136	192.168.206.189	ICMP	74
21	10.503241	192.168.206.189	192.168.206.136	ICMP	66
22	10.503732	192.168.206.136	192.168.206.189	ICMP	1083

An initial look at the PCAP file in Wireshark, shows three main protocols in use: TCP, HTTP and ICMP. The client sends an HTTP GET request for a file called `rfc.txt`, and the server promptly replies with the file requested. All remaining communications in the capture are ICMP echo requests to and from the server.

RFC Time:

The `rfc.txt` file can be viewed in the Wireshark window, but can also be exported as such:

> `File` > `Export Objects` > `HTTP`
>
> Select packet #`12`, and click `Save`

The RFC defines a protocol named the Insecure File Transfer Protocol over Ping (IFTPP), and the basic structure of an IFTPP message is defined as such.

```
message IFTPP {
  int32 XXXXXXXX = 1;     // thonk
  bytes XXXXXXXX = 2;        // thonk
  bytes XXXXXXXX = 3;       // thonk

  enum Flag {
      SESSION_INIT = 0;     // client to propose session ID
      ACK = 1;              // generic ack, multiple uses
      CLIENT_KEY = 2;       // client proposed key
      SERVER_KEY = 3;       // server proposed key
      FILE_REQ = 4;         // client requesting file
      FILE_DATA = 5;        // requested file data
      FIN = 6;              // transfer is complete
      RETRANS = 7;           // request retrans of prev packet
  }
  Flag type_flag = 4;       // describe payload type
}
```

The RFC also defines how a shared (symmetric) key is exchanged and calculated in Go.

Both client and server exchange 16 bytes of random data, and each perform the following calculation:

```
// calculate the shared key by combining keys, sort by
// descending, then taking sha1
func calcSharedKey(key1 []byte, key2 []byte) []byte {
    combined := append(key1, key2...) // put two keys together
    sort.Slice(combined, func(i int, j int) bool {
        return combined[i] > combined[j]
        }) // sort descending
    hasher := sha1.New()
    hasher.Write(combined)
    sha := base64.URLEncoding.EncodeToString(hasher.Sum(nil))
    return []byte(sha)
}
```

Other parts of the RFC note that sessions are initiated with a payload of `newSession` followed by the session ID (SID). This can be seen in the plaintext of the PCAP. (Here, SID=`1a08`)

```
Data (24 bytes)
    Data: eb04120a6e657753657373696f6e1a0877684e59506a6534
    [Length: 24]
```

```
00 0c 29 02 a5 bc 00 0c  29 9b 22 06 08 00 45 00   ··)····· )·"···E·
00 34 e2 bf 40 00 40 01  39 72 c0 a8 ce bd c0 a8   ·4··@·@· 9r······
ce 88 08 00 aa 4c 08 a1  84 96 eb 04 12 0a 6e 65   ·····L·· ······ne
77 53 65 73 73 69 6f 6e  1a 08 77 68 4e 59 50 6a   wSession ··whNYPj
65 34                                              e4
```

The request for a file can also be seen in plaintext:

```
Data (24 bytes)
    Data: eb041208666c61672e6a70671a0871397267517562302004
    [Length: 24]
```

```
00 0c 29 02 a5 bc 00 0c  29 9b 22 06 08 00 45 00   ··)····· )·"···E·
00 34 e2 c3 40 00 40 01  39 6e c0 a8 ce bd c0 a8   ·4··@·@· 9n······
ce 88 08 00 35 c3 08 a1  84 96 eb 04 12 08 66 6c   ····5··· ······fl
61 67 2e 6a 70 67 1a 08  71 39 72 67 51 75 62 30   ag.jpg·· q9rgQub0
20 04                                              ·
```

It is important to note that the last bytes are often the checksum and the flag field (separated by a space character `0x20`). This, too, is defined by the RFC and was is implemented in *Python3* by my team (see `gen_checksum` function).

Key E><change:

In frames #19 and #20, the client and server exchange random data for symmetric-key generation. The file transferred in subsequent messages will be chunked and then XORed with key before sending.

Client's random bytes: `4f163f5f0f9a621d729566c74d10037c`

Server's random bytes: `52fdfc072182654f163f5f0f9a621d72`

The shared key was generated using a slightly *modified* version of the Go function provided by the RFC. It can be reached via:

https://play.golang.org/p/dKh6Qr19_eZ

The symmetric key: `68597258426b324350694649544a3374394e435675584e6f6a4c6f3d`

File Transfer:

After the client requests `flag.txt`, the server and client exchange messages wherein the server repeatedly sends messages chunks (flag=`05`) of the file and the client responds with a `fDataAck` to each one accordingly.

```
    28 10.505302    192.168.206.136    192.168.206.189    ICMP    1083
    29 10.505510    192.168.206.189    192.168.206.136    ICMP    56
```

```
    Data: eb0412086644617461416 36b2001
    [Length: 14]
```

```
0000  00 0c 29 02 a5 bc 00 0c  29 9b 22 06 08 00 45 00   ··)····· )·"···E·
0010  00 2a e2 cb 40 00 40 01  39 70 c0 a8 ce bd c0 a8   ·*··@·@· 9p····
0020  ce 88 08 00 c1 54 08 a1  84 96 eb 04 12 08 66 44   ·····T·· ····· fD
0030  61 74 61 41 63 6b 20 01                          ataAck ·
```

This transpires until the server sends the last chunk, not limited by the maximum packet size of ICMP. This packet (#70) has a total size of 871.

67	10.517808	192.168.206.189	192.168.206.136	ICMP	56
68	10.518142	192.168.206.136	192.168.206.189	ICMP	1083
69	10.518342	192.168.206.189	192.168.206.136	ICMP	56
70	10.518641	192.168.206.136	192.168.206.189	ICMP	871
71	10.518896	192.168.206.189	192.168.206.136	ICMP	56

Solving the challenge:

Before the challenge could be solved, a better way to extract packet data was required. Although a Wireshark packet dissector *could* be manually created, nobody on the team claimed to be familiar enough with C/Wireshark to do so. As such, the packets that required analysis, were exported from Wireshark in an easily manageable JSON format.

> To replicated this:
>
> In Wireshark, filter out only **ICMP** packets by using the `icmp` filter.
>
> Then, `File` > `Export Packet Dissections` > `As JSON…`
>
> Ensure that the `Displayed` radio button is selected, name your file `packet_dissections.json`, and click `Save`

Now with the packets as JSON objects, a Python3 script could be used to analyze them and extract the data. My team used the following script named `IFTPP.py`:

```
"""
Name: Jonathan Alter

Partners:
    > Spencer Yost
    > Jan Masztal
    > Matthew Thibodeaux
"""
import hashlib
import base64
import json

# Client random bytes (as hexstring)
c = bytes.fromhex('4f163f5f0f9a621d729566c74d10037c')
# Server random bytes (as hexstring)
s = bytes.fromhex('52fdfc072182654f163f5f0f9a621d72')
# Shared key - calculated in golang
shared_key = "68597258426b3243506946495544a3374394e435675584e6f6a4c6f3d"

def gen_checksum(hex_data):
    """
    Takes the payload of the IFTPP packet and calculates the checksum.

    hex_data: A hexstring representing the IFTPP payload
```

```
    """
    data = bytes.fromhex(hex_data)
    sha1 = hashlib.sha1(data).digest()
    b64 = base64.b64encode(sha1)
    return b64[-9:-1].hex()


def read_iftpp_data(hex_data):
    """
    Takes a hexstring representing ICMP payload (`hex_data`) and interprets it as a IFTPP pac
ket.
    NOTE: This assumes the packet contains a Flag!!!

    Parameters:
        hex_data:   a hexstring representing the ICMP data payload
    Returns:
        dict: A dictionary representation of the IFTPP packet.
    """
    data = bytes.fromhex(hex_data)
    flags = data[-2:]
    checksum = data[-10:-2]
    sid = data[-12:-10]
    # seems data packets have an extra byte and start at 5 not 4
    payload = data[5:-12] if flags[-1:].hex() == '05' else data[4:-12]
    calc_cksm = gen_checksum(payload.hex())
    return_val = {"payload": payload.hex(),"flags":flags[-
1:].hex(),"sid":sid.hex(),"checksum": checksum.hex(), "calc_cksm": calc_cksm, "checksum_match
es": calc_cksm == checksum.hex()}
    return return_val


def xor_chunk(chunk, key):
    """
    XORs the data supplied in `chunk` with the `key`
    Parameters:
        chunk: hexstring representing the key
        key: hexstring representing the key
    Returns:
        bytes: resulting values from an XOR of `chunk` and `key`
    """
    # Convert to bytestrings
    k = bytes.fromhex(key)
    d = bytes.fromhex(chunk)
    #preform xor and return
    return bytes([d[i] ^ k[i % len(k)] for i in range(len(d))])

if __name__ == "__main__":

    # Open wireshark packet dissection for the selected ICMP packets
    with open('packet_dissections.json','r') as f:
```

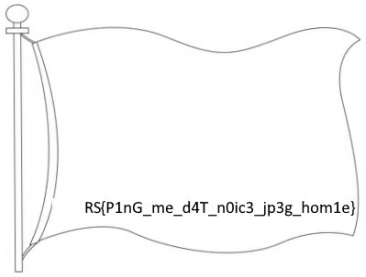
```
        # load it as a json file
        packets = json.load(f)

    # Extract the ICMP data field
    icmp_data = []
    for p in packets:
        # Convert to hexstring by removing ':' delimiter
        icmp_data.append(p['_source']['layers']['icmp']['data']['data.data'].replace(":",""))

    # Open output JPEG
    with open("out.jpg",'wb') as f:
        # Iterate over the icmp data hexstrings
        for p in icmp_data:
            # Interpret each as a IFTPP packet
            interpreted_packet = read_iftpp_data(p)
            # Check if the flag is set to 'FILE_DATA' ('05')
            if interpreted_packet['flags'] == '05':
                # If yes, XOR payload with key and write the binary data to out.jpg
                f.write(xor_chunk(interpreted_packet['payload'], shared_key))
```

Note: Ensure that the *packet_dissections.json* file is located in the **same directory** as this script.

Run the script on Windows: `python IFTPP.py` (Linux/Unix may require the use of `python3`)

Upon the completion of execution, a file named `out.jpg` should have been created.

Here are its contents:



Unsolved Mysteries:

The RFC does not always seem followed.

- There is **no `0` flag** sent by the client in session *initialization* phase
- The number of bytes in the header is **inconsistent** between file data messages and other messages (4-byte header vs a 5-byte header)
- The RFC claims that the original file was read, **base64 encoded**, chunked, and XORed with the symmetric key. In our experience, it seems that the base64 encoding was not performed – as simply XORing the chunks with the key yielded the raw bytes for a JPEG.
- Checksums were **not always accurate** and were often off by several bits
- The headers used (4 or 5 bytes) are not well defined and seemingly mean nothing