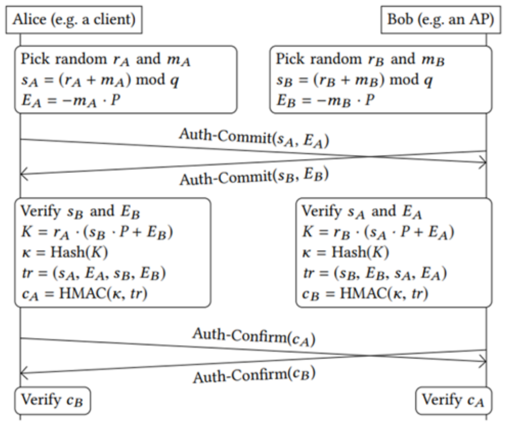
WPA3 Timing CTF Guide – Level 3

Introduction

Welcome to the WPA3 Timing CTF! In this CTF we will see how we will use a side-channel to find out a Wi-Fi password using an offline dictionary attack. We will attack access points that use the dragonfly handshake and see how by measuring the time it takes for the AP (Access Point) to respond to a request we can get the password.
The handshake looks like this: both parties send a commit frame, then calculate a shared key for communication and send a confirmation message.


Alice (e.g. a client)
Bob (e.g. an AP)
Pick random r_A and m_A
$s_A = (r_A + m_A) \bmod q$
$E_A = -m_A \cdot P$
Pick random r_B and m_B
$s_B = (r_B + m_B) \bmod q$
$E_B = -m_B \cdot P$
Auth-Commit(s_A, E_A)
Auth-Commit(s_B, E_B)
Verify s_B and E_B
$K = r_A \cdot (s_B \cdot P + E_B)$
$\kappa = \text{Hash}(K)$
$tr = (s_A, E_A, s_B, E_B)$
$c_A = \text{HMAC}(\kappa, tr)$
Verify s_A and E_A
$K = r_B \cdot (s_A \cdot P + E_A)$
$\kappa = \text{Hash}(K)$
$tr = (s_B, E_B, s_A, E_A)$
$c_B = \text{HMAC}(\kappa, tr)$
Auth-Confirm(c_A)
Auth-Confirm(c_B)
Verify c_B
Verify c_A

During the derivation of the key, the client uses a hash_to_group function that looks something like this in python-like pseudo code:

```
1 def hash_to_group(password, id1, id2, token=None):
2   label = "EAP-pwd" if token else "SAE"
3   for counter in range(1, 256):
4     seed = Hash(token, id1, id2, password, counter)
5     value = KDF(seed, label + " Hunting and Pecking", p)
6     if value >= p: continue
7
8     P = $\text{value}^{(p-1)/q} \bmod p$
9     if P > 1: return P
```

The AP hashes the password, both MAC addresses and an increasing counter to produce a random group element, and then checks if the result is indeed a group element in line 6.
Notice that if the value is bigger than p we will do another iteration, and so on until a value smaller than p is produced (while increasing the counter every time). The number of iterations executed is therefore dependent on the password.

This attack uses the timing of the response from the AP to figure out the number of iterations executed, and by spoofing as different addresses for the client we can get a unique fingerprint of the password!
Using the fingerprint, we can get the actual password using offline brute-force.
To complete the attack, you will need to complete the missing functions!
You can find the full documentation below.

datareader.py

Inside the file “datareader.py” you will find functions that can be used to get a fingerprint of the password based on the measurements acquired from “dragontime”.
The goal is to estimate as accurately as we can the real number of iterations that the server performed during the hash-to-group method for each spoofed MAC address.

1. extract_data(path)

Arguments:

- path: a path to the file containing the measurements from dragontime

Returns:

- a pandas dataframe of the measurements

The measurements are inside a text file that has some information in the first few lines after which we have the measurements themselves. We want to parse them to a pandas DataFrame so it is more convenient.

This function gets the path to the text file containing the measurements and a DataFrame where one column titled "STA" has the last byte of the spoofed address of a single measurement, and another column titled "Time" containing the time this measurement took to perform.
For example:

STA	Time
01	1234
02	4321
03	5678
…	….
13	1234

Each row corresponds to a single measurement (meaning there are several rows for each unique address).
Note: from now on, the last byte of the spoofed address will be referred to as the index of the address (since the other bytes are identical for the spoofed addresses).

The next few functions are going to estimate the number of iterations performed for each address, using a statistical test we call Crosby’s Box Test.

The execution time for each address is not deterministic (due to noise and other factors) but we know it is dependent on the number of iterations performed for each address. To get a

fingerprint of the passwords, we first want to find out which addresses cause the same number of iterations and find “groups” of addresses that share the iteration count. Then, we can sort these groups from shortest to longest time.
To do so we need a way to tell if two addresses share the same iteration count.

This test uses the fact that some quantile ranges have less noise so are better for our analysis. Our goal here is for each pair of samples (of two different spoofed addresses) to be able to tell if they were sampled from the same distribution, and therefore hash-to-curve has done the same number of iterations for them.

We get as parameters two quantiles i, j
For each sample, we define an interval between the two quantiles.
For example, if for STA “0C” the i-th quantile is 5000 and the j-th quantile is 6000 then our interval is [5000,6000]

Given two samples we want to find out if they are taken from the same distribution. So, we simply check if their intervals intersect.
The next functions will use this test to estimate the number of iterations performed.

2. qplot(df, addrs=20)

Arguments:
- df: a DataFrame of Dragontime measurements
- addrs: number of spoofed addresses

Plots a quantile plot of the addresses in the measurement data. plt.legend() to add a legend to the graph, plt.show() to see the graph.

3. get_address_quantiles(df, addrs=20, low, high)

Arguments:
- df: a DataFrame of Dragontime measurements
- addrs: number of spoofed addresses
- low: the lower quantile requested (a number between 0 and 1)
- high: the higher quantile requested (a number between 0 and 1)

Returns:
- a pandas dataframe of each spoofed address with the low and high quantiles found

This function should create a DataFrame where each row corresponds to a spoofed address. The first column title is "STA" which will have the index of the address, the second column is "Low Quantile" and has the quantile of the “low” parameter supplied, and the third column is "High Quantile" and has the quantile of the “high” parameter supplied.
You are encouraged to used pandas quantile() method for the calculations.
The result should look like this:

STA	Low Quantile	High Quantile
01	1234	5678
02	1234	5678
03	1234	5678
…	….	….
13	1234	5678

Here each row corresponds to a unique spoofed address.

4. min_iterations(df, addrs=20, low, high)

Arguments:

- df: a DataFrame of Dragontime measurements
- addrs: number of spoofed addresses
- low: the lower quantile for the Box Test (a number between 0 and 1)
- high: the higher quantile for the Box Test (a number between 0 and 1)

Returns:

- a pandas dataframe of each spoofed address with the low and high quantiles found and the grouping based on Crosby's Box test

This function groups the addresses to groups of addresses which we estimate come from the same distribution, based on the Box Test described above. We sort those groups and assign each one an index, which is also a lower bound on the number of iterations performed during the handshake with the host.
To the dataframe we created in get_address_quantiles we add a new column "min_iters".

The algorithm is as follows:

a. Keep a counter i that starts with 1
b. Inside a loop:
    i. Sort all addresses by their 'Low Quantile' values. Pick the smallest one that has not yet been assigned to a group.
    ii. Find all of the addresses that have intervals overlapping with the interval if the address from (a) (of the ones we didn't mark yet)
    iii. Mark those addresses as group “i” in the "min_iters" column
    iv. Increase i
c. Until all addresses have a group assigned. Notice that the group number is a lower bound.

What we did here is grouping the addresses using Crosby’s box test and sorting the groups.

The final stage will be to use the groupings we made and the lower bounds to get a more accurate estimation of the actual number of iterations performed for each group.
We can assume with good probability that the number of iterations performed for the groups with “min_iters” <= 3 is equal to their “min_iters” value. Using this assumption, we can estimate the time it takes to perform a single iteration, and then get a more accurate estimation for the remaining groups.

5. estimate_iter_time(addr_quantiles)

Arguments:

- addr_quantiles: a DataFrame of the quantiles of each spoofed address

Returns:

- An estimation of the time it takes to execute one iteration of the hash_to_group method

This function estimates the time it takes to perform single iteration
This estimation can be calculated as follows: $Avg(\frac{{q^s}_i - {q_i}^t}{s-t})$
Where ${q^s}_i$ is the i-quantile of an address with min_iters == s <=3
And ${q_i}^t$ is the i-quantile of an address with min_iters == t < s
Notice, (s, t) can be (2,1), (3,1), (3,2).

You can do this calculation for both the high and low quantiles and return an average of them both.

6. iterations(df, addrs=20, low, high)

Arguments:

- df: a DataFrame of Dragontime measurements
- addrs: number of spoofed addresses
- low: the lower quantile for the Box Test (a number between 0 and 1)
- high: the higher quantile for the Box Test (a number between 0 and 1)

Returns:

- a list of estimated iteration counts for each address

This function combines everything we did so far.

a. First calculate the average time a 1-iteration execution takes. For this you can average both quantiles for each address, and average across all addresses within the first group
b. For i <= 4 until all addresses were estimated: calculate the average time of the group with min_iters == i in a similar manner to the way you calculated the average time of a 1-iteration execution.
c. Estimate the true number of iterations like so:
   1 + (((avg of group i) - (avg of group 1)) / (time of 1 iteration))
d. Finally return the estimated values as a list.

fingerprint.c

In this file you will need to implement the brute-forcing stage of the attack.

You will need to use the function “sae_derive_pwe_ffc” inside sae.c to get the iteration count for each password and spoofed address pair (modify it so it returns the iteration count).

Then you will need to complete the missing parts with the mac address of the attacked AP (found in MAC_AP) and of the attacker device, the relevant info of the DH groups you are using (can be found in dh_groups.c)

And finally perform the dictionary creation:
Write to the output file in a csv format.
Go through every password in the password list provided.
Each row should begin with the password itself, followed by the iteration count for each spoofed address. For example:

12345678, 1, 1, 2, 4, 2, ..., 1
password, 2, 1, 1, 3, 5 …, 2

For the spoofed addresses you should iterate through the last byte of the address from 0 to 19.
(Use this exact format. Every element that is not the last in its line is followed by “, “)

find_matches.py

In this file you will need to implement a program that transforms the csv file returned by fingerprint.c into a KDTree, and then find the closest matches of new fingerprints to it.

1. create_tree(fp)

Arguments:
- fp: csv filename

Returns:
- passwords: a list of the passwords
- tree: k-d Tree of all the fingerprints, listed in the same order as the passwords

This function creates a KDTree and returns it, together with the passwords.

2. __main__(argv)

Arguments:
- argv[1]: timing measurements filename
- argv[2]: fingerprint csv filename
- argv[3]: optional variable k-number of closest matches to print. default 5.

Prints out the k closest matches to the measurements data in the csv file.

dragontime.c

The program for making fraudulent logins to the Wi-Fi network and taking timing measurements. It sends commit frames from 20 different spoofed addresses, by iterating from 0 to 19 in the last byte of srcaddr.

To complete the missing parts of the code, make sure you understand the following structs and functions (found in drafontime.c and dragontime.h):

static struct **state**

Various information that is maintained through the attack and updated.

static void **calc_prev_commit_diff**(const struct state *state, struct timespec *diff)

state->curraddr holds the last byte for the current spoofed address.

You will need to complete code in some key functions:

1. static void **process_packet**(struct state *state, const unsigned char *buf, const int len)

This function processes the received packets. You need to complete the missing parts that deal with calculating the time difference.

2. static void **check_timeout**(const struct state *state)

Here you need to check if the time elapsed from the previous commit is over the timeout. If so, you need to inject a deauth packet and queue the next commit.

3. static void **event_loop**(struct state *state, char *dev)

This is the main function of the attack. Complete the necessary initialization steps to start the main event loop.

Running the attack

After completing the missing code in all the files, run:

```
$ sh setup_user.sh $(<MAC_AP) wlan0
```

You can replace “wlan0” with your Wi-Fi device, find it with `ip addr` or `ifconfig.`

Use dragon time to gather measurements of the time AP takes to respond to spoofed commit frames:

```
$ sudo ./dragontime -d $(<DEVICE) -a $(<MAC_AP) -o measurements.txt
```

After getting 100 measurements, stop dragontime with CTRL+C.

Use fingerprint to build a dictionary of fingerprints for all the passwords in the password list:

```
$ ./fingerprints passwords.txt out.csv
```

You will generate a CSV file “out.csv”.

In find_matches.py, use datareader.extract_data to convert the data to a DataFrame.
Use datareader.iterations to calculate a fingerprint for the password and use fitting parameters for “low” and “high”.
You can experiment with different intervals to see what works best. You can also graph the quantiles of each measurement using datareader.qplot(), and see what range looks “clean” and without much noise. note that the range shouldn’t be too big.

Finally, once you have your fingerprint, find the ones that are most similar in the dictionary:

```
$ python find_matches.py measurements.txt out.csv
```

If everything works right, the most similar should be the true password!

Testing

In the folder “example” you will find example measurements for a password “12345678q” for your own testing purposes. Provided are also outputs for some of the functions in “datareader” for given parameters, and the file “exemple_out.csv” is the output of “fingerprint“ for the MAC addresses that you can find in 12345678q.txt.

Good luck! ☺