WPA3-Attack

Ariel Berant Asaf Malachi Tal Orlansky Shmuel Segal Ishay Yemini

ABSTRACT

We consider a known problem in Wi-Fi and communications over the internet in general, come up with a way to improve the public's knowledge and educate them, demonstrate the problem and implement the attack.

Wi-Fi allows devices to communicate easily, and is supposed to be protected by WPA certificates. However, in this paper we show that even the code meant to protect our privacy can be circumvented, and that caution should always be taken. We implement the attack used to exploit a weakness in WPA3 and show how security isn't universal, but case-specific. Additionally, we thoroughly explain how and why the attack itself works and is used. We also build a CTF with detailed instructions to allow people from various backgrounds to interact with the code. Finally, we believe that our work shows the need for consistent auditing of cyber-security protocols, for the constant public scrutinization of said protocols, and allows for the CTF players to gain some knowledge about the field\specific exploit.

1 INTRODUCTION

We use Wi-Fi in almost every place we go. At work, in our home, at the mall, and in some cases even while outside on a walk. But first, what even is Wi-Fi? Wi-Fi is a facility allowing computers, smartphones, or other devices to connect to the internet or communicate with one another wirelessly within a particular area.[oxford dictionary] In more understandable terms, Wi-Fi is a family of protocols, upon which wireless communication is based. These protocols are the way two or more devices communicate and exchange data using radio waves. This allows us to connect to the internet, it all of its vastness. Thanks to Wi-Fi, we have the ability to "doomscroll" in our favourite social networks, and watch cute cat videos, whilst not having to be tethered to some clunky and cumbersome cable network.

Now, we get to a pressing matter that comes with the use of Wi-Fi: security concerns. Obviously, most people would not mind if it is publicly known that they watch cat videos(unless their partner is an avid dog person of course), so let's give a much more disturbing example that affects all of us.

Let's consider two parties, Alice and Bob. Alice wants to share some extremely secret information with Bob. Bank account details, company secrets worth billions of dollars, nuclear or government secrets, or any explicit information about either party that could cause harm being public. Alice, in the comfort of her own home, would like to send Bob the information over the internet, as they live in different countries. So, Alice connects to her home Wi-Fi network, and using strong encryption, sends the secret message to Bob. However, unbeknownst to the duo, our antagonist Eve is also there. Let's say that Eve has the power to gain access to the Wi-Fi network, and manages to create a Man-In-The-Middle attack(or any other attack) and steals the information. This is a cause for worry, especially since these situations can happen every day.

That is the reason that we have the WPA's - Wi-Fi Protected Access certificates [1]. These certificates, which are given to devices that are up to standard, are meant to ensure that the transfer of the information using the Wi-Fi protocols is done in a safe enough manner.

The WPA3 certificate, which was released by the Wi-Fi alliance as the successor to WPA2, is one such certification, developed to secure wireless computer networks. WPA3 was proposed after some serious concerns and vulnerabilities arose regarding WPA2, the previous certificate proposed by the Wi-Fi alliance. If we simplify things down a bit, the WPA3 certificate mandates support of the Dragonfly handshake protocol, and only adds a transition mode(backwards compatibility with WPA2).

Since the security of WPA3 is mostly reliant on the Dragonfly handshake, we should expect it to be pretty secure(in theory and in practice). Some variant of the Dragonfly has been formally proven to be secure [2], so we shall see to the practicing side of things.

In our project, we used open source code to manipulate formerly discovered(now patched) vulnerabilities in WPA3, and thus managed to gain access to a wireless network using the unpatched code and certificate version. We created a CTF around said vulnerabilities for a few reasons. First, we wanted to highlight the need for certificates that ensure the safe and secure usage of wireless networks, and the need to constantly monitor and fix vulnerabilities in them, for the privacy and security of all users. Second, we wanted to show how sloppy implementations, even of theoretically secure protocols, can sometimes have disastrous consequences. Last but not least, we want to allow people, from interested newbies to more experienced coders to gain hands-on experience in said vulnerability, and through that understand it more thoroughly than before.

2 RELATED WORK

During the process of our project, we used several open resources. The basis for the attack itself comes from "Dragonblood: Analyzing the Dragonfly Handshake of WPA3 and EAP-pwd" [3], which is the primary paper. The paper describes the attack and the code for both simulating a WP3 network, and making repeated fraudulent logins to measure response time. Using the paper as the how-to for the implementation of the attack, we modified some of the code published by the writers, adapting it for our use case.

Additionally, during the implementation of our attack, we used "Opportunities and Limits of Remote Timing Attacks" [4], a paper that describes a statistical "Box Test". Using that test, we are able to distinguish between different distributions based on the timing samples collected by running the code published by the first article's writers.

3 ATTACK DESCRIPTION

In our CTF, we implement a side-channel attack on the Dragonfly handshake protocol.

The Dragonfly handshake protocol is a Password Authenticated Key Exchange. Used in WPA3, this protocol transforms(hashes) the password into a pseudo-random element of some finite group. The groups are either a finite field, or an elliptic curve over a finite field.

Particularly, after party A first sends its element, party B must transform its own password to check if party A has sent the correct password. Afterwards, it will almost always reply, either accepting the input from A if it is identical to its own hash, or rejecting otherwise.

In the case of finite fields, Dragonfly transforms the password into an element of a prime group. To do this, it uses the sha256 hash function, which is very slow. The function outputs a pseudorandom uniformly distributed whole number in the interval $[0, 2^n]$, where 2^n is the smallest integer larger than the prime $n = \lceil \log_2 P \rceil$. Since the number must be smaller than P, if we did not succeed in getting such a number in the first try, the protocol will repeat the hash function until it finds a suitable value.

It is important to note that the hash function is deterministic. This means that its output depends entirely on the password, the MAC addresses of the access point and the client's, and the number of attempts at finding a valid number(add hash to curve code from paper here).

The chance of the number being smaller than P is $\frac{\mathrm{P}}{2^n}$. Since most of the safe primes used for finite field cryptography are slightly smaller than a power of 2, we get that $\frac{\mathrm{P}}{2^n} \approx 1$. However, for some of the primes, particularly group 24 [5], $\frac{\mathrm{P}}{2^n} \approx 0.47 \sim \frac{1}{2}$, so multiple iterations are a regular occurrence.

This process leaks information about the password, since the long runtime reveals the number of iterations, and given the set of MAC addresses we spoof, only about 2^{-k} of passwords perform k iterations.

By repeatedly trying to start the protocol, it's possible to get a set of response times for each spoofed MAC address. We do this to a number of addresses, sending all of them to the device simultaneously, which makes the influence of the noise on them equal. It is then possible to compare the response times for the addresses, and get a map of address-iterations that is unique to the correct password. When comparing it to a dictionary of common passwords and their address-iteration maps(can be calculated in a deterministic manner), it becomes much easier to derive the password.

To derive iteration counts from timing measurements, the original paper recommended using Crosby's Box Test. In the test, 2 random variables are compared by picking a range of quantiles $[low, high]$, and checking if the intervals of both variables intersect. So, given 2 random variables X, Y such that F_X, F_Y are their empirical distribution functions, X, Y are assumed to have the same distribution if:

$$F_X^{-1}(low) \in \left[F_Y^{-1}(low), F_Y^{-1}(high)\right]$$

or

$$F_X^{-1}(high) \in \left[F_Y^{-1}(low), F_Y^{-1}(high)\right]$$

We have attempted several methods of implementing this test. The difference between the methods is in the algorithm that decides which addresses give an equal iteration count.

First algorithm.

(1) Assign i = 1
(2) Find the address with the minimal low quantile that hasn't been assigned yet, i.e smallest value which has not yet been assigned to some number of iterations j.
(3) Assign all addresses with intersecting intervals as having i=1 iteration.
(4) Assign i⟵i+1.
(5) Repeat from 2 until all addresses have been assigned.

This algorithm works the best when low and high are chosen in the least noisy part of the data, and fails when there is too much noise.

Second algorithm. Similar to the first algorithm. But here, instead of finding intervals that intersect only the minimal address, we find all addresses that intersect the set i.
This algorithm works the best when [low, high] are in an interval with large distances between different iteration counts, but is otherwise often worse than algorithm 1.

Third algorithm. While telling different quantile graphs F_X^{-1}, F_Y^{-1} apart is easy by eye, Crosby's Box Test only looks at 2 points which can be inconsistent. Therefore, the final algorithm is a clustering algorithm that looks at most of the quantile graph of each address.

The algorithm is a version of agglomerative clustering with ward linkage and a distance threshold that receives quantile graphs as vectors(excluding the highest and lowest quantiles because their variance is too high).

This algorithm fails when addresses with equal iterations are farther apart than the threshold, or different iteration clusters are closer than it, but generally it outperforms the previous algorithms, although picking the right threshold can be harder from just looking at the quantile graph.

After sorting the addresses into different iteration sets, we estimate the iteration time by assuming we measured the sets of addresses with 1,2,3 iterations were measured correctly(meaning the assignment of iterations is correct). In the first algorithm, we calculate the expected value of an iteration given our sample space. In the third algorithm, we average the distance between them, and estimate the number of iterations of the other clusters relative to it.

Final desicion. We chose to work with the first algorithm, since it worked best in most use cases, and since we didn't mind giving low and high values in less noisy parts of the data(as done in the paper).

For testing the iteration approximating process, we have a program for plotting the quantile graphs. This makes it easy to tell different iteration counts by eye, and checking why an algorithm isn't working.

After calculating the length of one iteration, recall we assume our address set for one iteration is correct. This means that we can calculate the leftover for all other addresses not in the set by subtracting the time for one iteration with noise from the time of the other addresses, and then dividing by the time it takes for one iteration. This gives us a better approximation of the number of iterations. From here, it is quite easy to go over the passwords file and check which fingerprint is closest to the measured iterations vector and return it.

4 DESIGN

Our CTF challenge consists of filling in missing functions according to text instructions. To implement the attack, we modified the code for simulating WPA3 networks to make it vulnerable, and the original attack code used in the Dragonblood paper [3].

The programs include:

- fingerprint.c - program for turning a list of passwords into a csv file with their address-iteration count fingerprints.
- dragontime.c - program for performing fraudulent logins and recording response time.
- sae.c - extracted from hostapd sources, we modified the sae_derive_ffc function so it returns the iterations counter.
- datareader.py - program for analyzing timing measurements.
- find_matches.py - program for finding the closest matches of a passwords fingerprint in a csv file of prints.

We have three levels for the CTF:

(1) The easiest level, the players need to complete missing functions in datareader.
(2) The medium level, they need also to implement the dictionary creation, as well as searching the dictionary for the right fit, and more missing code in datareader.py.
(3) The hardest level, players will need to complete more of datareader.py, as well as parts of dragontime.c and more work for fingerprint.c, and they will need to find themselves how to get the iteration count from sae_derive_ffc.

We've used several Python libraries for analysing the data:

- numpy and pandas, for manipulating the timing information.
- matplotlib.pyplot for visualizing the timing information.
- sklearn.cluster for its Agglomerative Clustering implementation, for analyzing the timing information.
- csv for reading the password fingerprints.
- scipy.spatial for its k-d Tree implementation, for analysing the password fingerprints.

5 CTF INSTRUCTIONS

First, make sure that both computers are running linux(we ran the CTF on kali).
You can mock everything on a single computer by running:

```
$ sh setup_mocked.sh
```

Admins

Setup: open the terminal, cd into the admin directory, and run:

```
$ sh setup_admin.sh
```

After the initial setup, there is usually no need to rerun the setup.
To start the CTFd, run:

```
$ sh run_ctfd.sh 1 wlan0
```

Replace “1” with the desired level(1, 2, or 3, from hardest to easiest) and “wlan0” with the Wi-Fi adapter of your choice.
After running run_ctfd.sh, you can start the AP by running:

```
$ sh run_hostapd.sh
```

Players

For the players, tell them to enter the website that runs the CTFd (it's http://[your IP address]:4000, unless you specified a different port). There, they need to register, and they'll see two challenges:

The first challenge requires them to donwload a zipped directory, unzip it, and enter the flag that's stored there. The directory is created and uploaded automatically when running `run_ctfd.sh`. It contains everything that the players need in order to hack the WPA3 network.

The second challenge is the main challenge - here, the players will need to actually modify the code, complete it, compile it and finally run the tools in order to infer the password.

There are three user guides, relevant to each level.

Possible hints to give to players:

datareader.py (level 1):

- Try to think how to mark that an address has an assigned iteration number, and how to choose the smallest low quantile value filtering upon the unassigned addresses.
- Remember how to calculate interval intersection. Maybe draw some intervals, and see what necessitates their intersection.
- Recall how to calculate sum\average on a specific subset of a dataframe.
- Remember we can update all addresses with the same minimal bound all at one.
- The average of the times taken has an added Y for every address average. We need to remove that somehow using time_1, to get the estimated actual time for the rest of the iterations.

datareader.py (level 2):

- Estimate time for iterations using expected value on the sample space of minimal assigned iterations 1, 2 and 3.
- What is needed to initialize the iteration estimation?

find_matches.py (level 2):

- Think how to create the k-d tree, and how to get the password with the closest fingerprint to the fingerprint we have.

dragontime.c(level 3):

- Think how to calculate the time for the measurement using the last time(how much time has passed since).
- How do we deauth? Look for functions that can help us, and remember the condition to actually deauth.
- Look for functions to open the card, set rate and get the MAC address.
- How do we create a timer? How do we initialize its periodic and initial expiration?
- Recall how to commit new commit and inject it(look for useful functions)

6 CONCLUSION

In the beginning of our project, we sought to create a versatile CTF to highlight the need for a better understanding of cybersecurity and the need for it among the public. We believe that our work has done that, and culminated in a tool for learning that can be used by many.

Seeing as even such a basic attack takes advantage of simple statistical tests of the protocol's implementation, it is our belief that a deeper understanding of maths and cryptography among the public is needed. The attack itself exploits a stage in the dragonfly algorithm that is very slow. Additionally, its runtime is relative to both the password and the clients MAC address, making the password easy to decode.

As this attack is a brute-force dictionary attack, we see it as an opportunity to help the public understand some aspects of security, one of which is secure password generation. While the mathematical background to implement the attack from scratch is non-trivial, our work shows that the attack can be converted into more digestible knowledge for those without said background.

Finally, in light of the above, we predict that the relative ease of the implementation and the importance of the subject will make it much more popular in years to come. Although not without problems, as seen here, it seems as though cybersecurity and the public interest in it will flourish in the future, helping us all in keeping our privacy and in finding our way in this new interconnected world.

CONTRIBUTIONS

Ariel. Helped Isolating the dragontime attack from the dragondrain-and-time repository. Wrote most of the Technical Report. Tested the running of the CTF and the attack on multiple computers. Wrote find_matches.py.

Asaf. Helped setting up the attack framework.

Tal. Helped setting up the attack framework. Worked on datareader.py and fingerprint.c, and integrated them as part of the CTF. Wrote the User Guide.

Shmuel. Helped setting up the attack framework. Worked on datareader.py and fingerprint.c. Contributed to the Technical Report, User Guide, and the Presentation.

Ishay. Wrote scripts to automate the CTF deployment. Isolated the dragontime attack from the dragondrain-and-time repository. Tested the running of the CTF and the attack on multiple computers. Contributed to the Presentation. Reformatted the Technical Report into Latex. Configured the CTFd framework.

REFERENCES

[1] Wikipedia, https://en.wikipedia.org/wiki/Wi-Fi_Protected_Access
[2] Jean Lancrenon and Marjan Škrobot. 2015. On the Provable Security of the Dragonfly Protocol. In Information Security. Springer International Publishing.
[3] Mathy Vanhoef and Eyal Ronen. "Dragonblood: Analyzing the Dragonfly Handshake of WPA3 and EAP-pwd." 2020 IEEE Symposium on Security and Privacy (SP). IEEE, 2020.
[4] Scott A. Crosby, Dan S. Wallach, and Rudolf H. Riedi. "Opportunities and limits of remote timing attacks." ACM Transactions on Information and System Security (TISSEC) 12.3 (2009): 1-29.
[5] Matt Lepinski and Stephen Kent, RFC 5114 section 2.3, (2008) datatracker.ietf.org/doc/html/rfc5114#section-2.3.