ABSTRACT

Recently, the Dragonblood attacks have attracted new interests on the security of WPA-3 implementation and in particular on the Dragonfly code deployed on many open-source libraries. One attack concerns the protection of users passwords during authentication. In the Password Authentication Key Exchange (PAKE) protocol called Dragonfly, the secret, namely the password, is mapped to an elliptic curve point. This operation is sensitive, as it involves the secret password, and therefore its resistance against side-channel attacks is of utmost importance. Following the initial disclosure of Dragonblood, we notice that this particular attack has been partially patched by only a few implementations.

In this work, we show that the patches implemented after the disclosure of Dragonblood are insufficient. We took advantage of state-of-the-art techniques to extend the original attack, demonstrating that we are able to recover the password with only a third of the measurements needed in Dragonblood attack. We mainly apply our attack on two open-source projects: iwd (iNet Wireless Daemon) and FreeRADIUS, in order underline the practicability of our attack. Indeed, the iwd package, written by Intel, is already deployed in the Arch Linux distribution, which is well-known among security experts, and aims to offer an alternative to wpa_supplicant. As for FreeRADIUS, it is widely deployed and well-maintained upstream open-source project. We publish a full Proof of Concept of our attack, and actively participated in the process of patching the vulnerable code. Here, in a backward compatibility perspective, we advise the use of a branch-free implementation as a mitigation technique, as what was used in hostapd, due to its quite simplicity and its negligible incurred overhead.

CCS CONCEPTS

• Security and privacy → Security protocols; Mobile and wireless security; • Networks → Wireless access points, base stations and infrastructure.

KEYWORDS

Dragonfly, PAKE, WPA3, Wi-Fi, cache attack
overhead. Moreover, they argue that a secure implementation of the countermeasures is an arduous task.

In this paper, we focus on the recommendations related to Cache-Based Elliptic Curve Side-Channels in [5], which address mitigations to the set of Dragonblood vulnerabilities related to cache-based attacks. Two mitigations are underlined: (i) performing extra dummy iterations on random data, and (ii) blinding the calculation of the quadratic residue test. For the first mitigation, the RFC 7764 [16] recommends that 40 iterations are always executed even if the password was successfully encoded requiring fewer iterations. Concerning the second mitigation, a blinding scheme is suggested for the function determining whether or not a value is a quadratic residue modulo a prime.

1.2 Our Contribution

In our paper, we show that such countermeasures are not enough to defend against cache-based side-channel attacks. In fact, these particular measures are designed to prevent only a part of Dragonblood’s attacks, and does not affect one of them. Especially, the cache attack leveraging a password dependent control-flow of loop in the try-and-increment conversion function is neither discussed in this document, nor patched in most implementations (except for hostapd, which was the direct target of the original attack). We aim to raise awareness about this particular attack, and prove that we can extend it to gain additional information, with fewer measurements. To this end, we identify several implementations in which some code is executed only during the iteration where the password was correctly converted (or encoded). We show how an attacker can use cache attacks in order to leak some information on the password. We stress that the original Dragonblood attacks are still applicable on such implementations. However, our work takes a step further by leveraging some state-of-the-art techniques that improve the attack performance without changing the underlying threat model.

Indeed, we extend the original attack in which only the outcome of the first iteration is leaked. Using an unprivileged spyware, we demonstrate that attackers are able to learn the exact iteration where the first successful conversion occurred with high probability. We achieve this result by monitoring well-chosen memory-lines with a Flush+Reload attack [38] to keep track of each iteration, and the success-specific code. We enhanced the reliability of our measurements by combing the attack to a Performance Degradation Attack (PDA) [4]. Since the successful iteration is directly related to key exchange context (defined by both MAC addresses and the password), this leakage allows attackers to significantly reduce the number of measurements needed to recover the password. For instance, only 160 measurements are required in order to discard all the wrong passwords using the Rockyou dictionary [25], while Dragonblood needs 580 measurements. Roughly, we cut down the number of measurements by three, which makes our attack performs better in practice.

We apply our findings on the wireless daemon iwd (iNet Wireless Daemon) that aims to replace wpa_supplicant. Ironically, iwd is written by Intel and our identified vulnerabilities in their implementation are caused by Intel cache design. The version 1.0 was released in October 2019 (after the publication of Dragonblood) and it is already adopted by Arch Linux and Gentoo. We also extend our work to FreeRADIUS, which a widely deployed project used by millions of users 1. We have not only communicated our findings to the maintainers of these two open-source projects, but also helped them to patch the vulnerable code.

The underlying technical details are quite similar concerning the identified vulnerability in iwd and FreeRADIUS. Therefore, for the sake of clarity and brevity, we will only detail the iwd case in the core of this paper. The FreeRADIUS case is discussed in Appendix C) in order to highlight the specificity of their implementation. In summary, we make the following main contributions:

- We extended the original Dragonblood attack to recover not only the outcome of the first round, but the iteration yielding a successful conversion (see Section 3).
- We estimated the theoretical complexity of our attack and compared it to the original one (see Section 3.5).
- We implemented a Proof of Concept of our attack, presenting practical results (see Section 4).
- We implemented mitigations and evaluated the overhead (see Section 5.1).
- We made all our code available 2, from the testing environment setup using Docker, to the password recovery script.

Our attack illustrates the danger of overlooking a widely potential attack during a standardization process. Therefore, we hope that our work would raise awareness concerning the need of constant-time algorithms by design that do not rely on savvy developers to provide secure implementations of ad-hoc mitigations.

1.3 Attack Scenario

We suppose a classical infrastructure where clients communicate with an access point (AP) across a wireless network. The goal of the attacker is to steal the password used to establish a secure communication with the AP. Once the password is compromised, the attacker can enter the network and perform malicious activities.

In order to leverage the vulnerabilities defined in this paper, the attacker requires to perform two tasks. First, they need to install a spy daemon on a client station without any particular privilege. Second, they need to create a rogue AP that behaves as the legitimate AP, but can use different MAC addresses for different connections.

Of course, we suppose that the rogue AP does not know the correct password, and therefore any session establishment between the rogue AP and a valid client will fail. Here, the goal of the rogue AP is to state different MAC addresses and to trick a client device to start a Dragonfly key exchange. Thus, the Wi-Fi daemon, using the correct password, will perform some operations that will be monitored by the attacker spy process. For each of these (failed) connections, the spy will generate a new trace that leaks the number of iterations needed to successfully encode the password. These bits of information are then used offline in order to prune a dictionary by verifying the number of iterations needed for each password. Each trace, with a different MAC address, yields a different iteration number. In our paper, we estimate that attackers require 16 traces to prune, for instance, the entire Rockyou dictionary. It is worth noting that, in our work, a trace generation needs 10 measurements

1https://freeradius.org/about/#usage_statistics
2https://gitlab.inria.fr/ddealmei/poc-iwd-acsac2020/-/tree/master/
with the same MAC address in order to guarantee a high accurate leakage.

### 1.4 Responsible Disclosure

Our attacks were performed on the most updated version of iwd and FreeRADIUS, as published at the time of discovery. We compiled both libraries using their default compilation flags, leaving all side-channel countermeasures in place. We reported our findings to the maintainers of iwd and FreeRADIUS following the practice of responsible disclosure. We further actively participated in coding as well as the empirical verification of the proposed countermeasures. Correspondingly, three patches were committed on the vulnerable projects: on iwd\(^4\), ell\(^5\) (the underlying cryptographic library of iwd, also maintained by Intel), and FreeRadius\(^5\). On a side note, iwd maintainers prefered not to scrupulously respect the recommendations of the RFC 7664 [16] by fixing the number of iterations to 30 (instead of 40). Moreover, we received special thanks from Alan Dekok, the project leader of FreeRADIUS, for our disclosure of the issue, and for helping with creating and verifying the fix.

We did not issue any communication to the Wi-Fi Alliance, since the identified vulnerability is mainly caused by implementation flaws, and not the standard itself.

## 2 BACKGROUND

In this section, we introduce the Dragonfly protocol, and describe the variant currently used in WPA3 and EAP-pwd.

### 2.1 The Dragonfly Key Exchange

Dragonfly is part of the Password Authenticated Key Exchange (PAKE) family. Its purpose is to use a low entropy password as an authentication medium, and to derive some high entropy cryptographic material from it. An important security requirement of PAKE protocols is to avoid offline dictionary attack: the only way an attacker should be able to get information about the password is to run the protocol with a guess and observe the outcome. Since Dragonfly is a symmetric PAKE, each party knows the password before initiating the protocol.

Dragonfly has been designed by Dan Harkins in 2008. In 2012, it has been submitted to the CFRG as a candidate standard for general internet use. This standardization ended up in 2015 by the release of RFC 7664 [16]. Along with the protocol described in this standard, some other variants have been included in other protocols, such as TLS-pwd [19], WPA3 [1] or EAP-pwd [40]. These variants mainly differ by instantiation details, such as some constant values.

The protocol follows the same workflow for both sides, meaning it can be performed simultaneously by both sides, without attributing a role. It can be broken down into three main parts: (i) password derivation; (ii) password commitment; and (iii) confirmation.

Following the disclosure of Dragonblood attack [35] in 2019, both the Wi-Fi standard [17] and EAP-pwd [18] are updating the password derivation function of Dragonfly. Due to the fact that updates are long to be approved, and even longer to be deployed, current implementations of WPA3 still use the original derivation function, as described in [16]. In this section, we will focus on currently deployed implementations, hence the original design.

#### 2.1.1 Password derivation

First, both the sender and the receiver need to convert the shared password into a group element. To do so, the standard describes a try-and-increment method called **Hunting and Pecking**. This approach consists in hashing the password along with the identity of both parties and a counter until the resulting value corresponds to a group element. For MODP groups, this method, called hash-to-group, converts the password into an integer modulo \( p \). For ECP groups, the method, called hash-to-curve, converts the password into the \( x \)-coordinates of an elliptic curve point. The \( y \)-coordinate is chosen at the end from the parity of the digest. The pseudocode describing this process on ECP groups is summed-up in Listing 1.

#### Listing 1: Hunting and Pecking on ECP group as used in WPA3. The value of \( label_1 \) and \( k \) may vary along with the implementation.

```python
1 def hash2curve(pwd, id1, id2):
2     found, counter = False, 0
3     A, B = max(id1, id2), min(id1, id2)
4     while counter < k or not found:
5         counter += 1
6         base = Hash(A || B || pwd || counter)
7         seed = KDF(base, label_1, p)
8         if is_quadratic_residue((seed/3 + a*seed + b, p):
9             if found == False:
10                 x, save, found = seed, base, True
11             # Not described in the RFC, but implemented in SAF
12             pwpd = random(32)
13             y = sqrt(x^3 + ax + b)
14             P = (x, y) if lsb(y) == lsb(save) else (x, p-y)
15         else:
16             return P
```

The security of Dragonfly is based on the discrete logarithm problem. Implementations can therefore rely on either Finite Field Cryptography (FFC) over multiplicative groups modulo a prime (MODP groups) or Elliptic Curve Cryptography (ECC) over prime field (using ECP groups). The exact workflow of the Dragonfly hand-shake varies slightly depending on the underlying group (ECP/ MODP). In order to avoid confusion, we adopt a classic elliptic curve notation: \( G \) is the generator of a group, with order \( q \). Lowercase denotes scalars and uppercase denotes group element. For elliptic curve, we assume the equation to be in the short Weirestrass form \( y^2 = x^3 + ax + b \mod p \) where \( a, b \) and \( p \) are curve-dependent and \( p \) is prime.

The protocol follows the same workflow for both sides, meaning it can be performed simultaneously by both sides, without attributing a role. It can be broken down into three main parts: (i) password derivation; (ii) password commitment; and (iii) confirmation.

Along the standardization process, various design flaws have been identified regarding the password-dependent nature of this function. Therefore, some mitigations were introduced to avoid password-dependent time variation in the execution of the function. Indeed, the number of rounds needed to find a value \( x \) that corresponds to a point on the curve is directly related to the password and the parties identities. First, the standard mandates a fixed number of iterations in the derivation loop, noted \( k \), regardless of the correct iteration. Setting this limit at \( k = 40 \) is recommended to minimize the risk of a password needing more iterations. All extra operations are performed on a random string, with no impact on the resulting element. Generating a dummy string for the extra

\(^3\)https://git.kernel.org/pub/scm/network/wireless/iwd.git/commit/?id=211f7dde6e87b4ab5234ac9883c8f9735b5977de498f1
\(^4\)https://git.kernel.org/pub/scm/libc/ell/ell.git/commit/?id=47c2a6fecc967b3ac538fd1e8f2dc7375725b76
\(^5\)https://github.com/FreeRADIUS/free-radius-server/commit/609e10ac47f4e66e4eece18c225a7d75e4ab66b
operations is not described in RFC 7664, but has been discussed by the CFRG during the standardization process, and has been included in deployed variants of Dragonfly (such as TLS-pwd [19] and SAE [1]). In our paper, we show that such an operation is not enough to defend against our cache attacks.

Sensitive information may also leak when checking for the validity of the potential x-coordinate (Listing 1, line 7). Indeed, WPA3 mandates to compute the Legendre before computing $y$. However, textbook Legendre may not be constant time and leak information about the value of $x$ [20]. To overcome this issue, the protocol has been updated [12, 15] to blind the computations by generating a random number for each test, squaring it, and multiplying it to the number being tested. The result is then multiplied by a per-session random quadratic (non-)residue before computing the Legendre symbol. The square root is then computed once and for all at the end of the function.

2.1.2 Commitment and Confirmation phase. Once the shared group element has been computed, both parties exchange a commit frame followed by a confirmation frame to conclude the handshake, as illustrated in Figure 1.

The commit frame is built with two values: a commit scalar $s_i = r_i + m_i \mod q$, computed by adding two random numbers $r_i$, $m_i \in [2, q)$, and a commit element $Q_i = \pm m_i P$. When receiving this frame, a party needs to check if the value $s_i$ is in the bounds (i.e. $s_i \in [2, q)$) and if the commit element $Q_i$ belongs to the group. A failure in any check results in aborting the handshake.

In the confirmation phase, both parties compute the master key $K$. For MODP groups, the key can be used as is, but the x-coordinate is extracted in case of ECP group. This value is then derived into two sub keys using a KDF. kck is a confirmation key and mk is used as a master key for the subsequent exchanges. Using the confirmation key, HMAC is computed over the transcript of the session. The resulting tag is included then in a confirm frame, to be verified by the other party. The handshake succeeds only if both verification ends successfully.

2.2 Integration of Dragonfly in WPA3

WPA3 uses a slight variant of Dragonfly, called Simultaneous Authentication of Equals (SAE) [1]. In this particular variant, the label values are fixed and each party is identified by its MAC address (id1 and id2 in Listing 1).

The SAE handshake is executed between the client and the access point (AP) in order to compute the Pairwise Master Key (PMK) called mk in Figure 1. Afterward, a classic WPA2 4-way handshake is performed with this PMK in order to derive fresh cryptographic material. Since the entropy of the initial master key is significantly higher than in WPA2, the dictionary attack on the 4-way handshake is no longer relevant.

2.3 Micro-architectural Preliminaries

2.3.1 Cache architecture. To mitigate the gap between slow memory access and fast processing, CPU benefits from fast access caches that are located close to the processor cores. The storage capacity is kept small, so only currently or recently used data are stored. On modern processors, the CPU cache is usually divided into several levels following an access hierarchy. Higher-level caches are closer to the core and typically smaller and faster than lower-level caches. In classical Intel architecture, which we will consider from now on, CPU cache is divided into three levels. Each core has two dedicated caches, L1 and L2, shared by all processes executing on the same core. The third cache, called Last-Level-Cache (LLC) is shared between all cores, hence all the processes.

When the CPU tries to access a memory address, it will first check the higher level cache. If the memory line has been cached, the processor will find it (cache hit). Otherwise, in a cache miss, the processor will keep looking in lower memory, down to the DRAM if needed. Once the appropriate memory line is found, the processor saves its content in cache for a faster access in the near future.

Finally, in modern Intel CPUs, the LLC has a significant property of being inclusive, meaning that it behaves as a superset of all higher caches. An important consequence of this feature, exploited in some attacks, evicting a memory line from the LLC will also have impacts on L1 and L2 caches.

2.3.2 Cache optimizations. In some cases, memory lines can be brought to cache even though they are not accessed. This is due to some cache optimization, that makes the exact cache behavior difficult to predict. For instance, Intel’s prefetcher ([23], Chapter 7) will pair consecutive memory lines and attempt to fetch the pair of a missed line to avoid looking for it in the near future. It may also detect memory access patterns and prefetch the lines to be loaded next.

2.3.3 Micro-architectural leaks. The time taken to access some data will significantly change whether the data is already in a CPU cache (cache hit), or if the CPU needs to look for it in the RAM (cache miss). This cache interaction can be triggered by two behaviors: (i) the CPU needs to access some data; (ii) the CPU needs to access some instruction.
In both cases, this can lead to a vulnerability if the element to access is related to some secret information (e.g., the index of the array or the instruction to access depends on a secret value). Given this information, an attacker can use a snoop process interacting in a particular way with the cache to trigger different timing of memory access. The nature of the interaction defines various types of attacks, each having benefits and drawbacks. Most instruction-driven attacks consist in probing the victim code, and inferring some data from the instructions performed.

Depending on the threat model and the targeted architecture, an attacker may or may not be able to access low-level caches shared between two threads. However, the LLC is shared between all cores. From now on, cache will refer to the LLC unless specified otherwise.

2.4 Related Work

Micro-architectural attacks have long been used to gain information about sensitive data. In 2014, Yarom and Falkner [38] presented a ground-breaking approach called Flush+Reload. Unlike previous approaches, which infer victim memory access based on the cache set activity, the novel approach directly monitors memory access in the inclusive L3 cache, yielding more interesting results. Since then, this method has been exploited to recover sensitive information in various contexts [3, 4, 6, 8, 9, 11, 14, 30, 32, 37, 39].

In 2016, Allan et al. enhanced the leakage by introducing the Performance Degradation Attack [4]. The goal is to systematically evict some well-chosen memory line in order to make the leakage easier and more reliable to exploit.

The Dragonfly handshake has already been reviewed in the past. A first version was found vulnerable to offline dictionary attack [13]. In 2014, Clarke and Hao outlined a small subgroup attack due to a lack of verification by the parties [16]. In 2019, Vahoef and Ronen identified several flaws in different implementations of Dragonfly, namely in WPA3 and EAP-pwd [35]. They outlined various vulnerabilities at the protocol level as long as at the implementation level. They demonstrated that some implementations of the hash-to-curve method leak sensitive information through micro-architectural attacks. Exploiting these leaks with a classic Flush+Reload attack, they were able to learn the outcome of the first quadratic residue computation, and therefore they could learn if the password was successfully derived at the first iteration or not. From there, they could guess the target password by tremendously cutting down the dictionary size.

In our attack, we greatly reduce the required traces by exactly estimating the number of iterations for a particular password with high probability. Then, we show how our attack can be used to guess the target password by tremendously cutting down the dictionary size.

We demonstrated our attack on iNet Wireless Daemon6 (iwd) version 1.8 (current version as of the time of writing), but we believe that our work is applicable to any unpatched implementation that is still vulnerable to the initial cache-attack (see Appendix C for the case of the current version of FreeRADIUS).

3.1 Threat Model

Our attack targets Wi-Fi networks, either a client or an Access Point (AP). Thus, we assume that the attacker to be within range of the physical target. To efficiently reduce the set of potential passwords, attackers need to monitor multiple handshakes, involving the same password and different MAC addresses. When the target is an AP, this can easily be done either by waiting for a client to connect, or by playing the role of a client. If attackers target clients, they can setup multiple rogue clones of the legitimate AP, advertising stronger signal strength (thereby making the client automatically choosing it) and different MAC addresses. If clients are already connected to the legitimate AP, attackers can force a de-authentication beforehand [7, 33]. Blocklist mechanisms are usually limited, since implementations tend to apply them based on the MAC address of the AP (that can easily be forged). We note that iwd might automatically generate a new random MAC address every time the daemon starts (or if an interface is detected, due to a hot-plug for instance). However, the default configuration uses one permanent address. We note that using different MAC addresses is not relevant to EAP-pwd, that is the Dragonfly variant in FreeRADIUS (see Appendix C for further details).

Due to the micro-architectural nature of the leak, attackers need to be able to monitor the CPU cache, using a classical Flush+Reload attack for instance. Since cache access and eviction do not rely on particular permissions, the most common assumption is that attackers can deploy an unprivileged user-mode program in the targeted device. This spy process runs as a background task and records the CPU cache access to some specific functions. Papers in the literature also suggest that such memory access can be granted remotely, performing the attack through JavaScript code injection in a web browser [28]. However, we did not investigate the effectiveness of our attack in such a context.

3.2 IWD Implementation

The Dragonfly exchange implemented in iwd follows the standard SAE [1]. Only the ECP-groups variant is supported with the NIST’s curves P256 and P384. The corresponding Hunting and Pecking is implemented in the function sae_compute_pwe, as illustrated in Listing 2.

---

6https://git.kernel.org/pub/scm/network/wireless/iwd.git/
bool sae_compute_pwd(struct l_ecc_curve *curve, char *pwd,
    const uint8_t *saepwdata, uint8_t *sae_pwd_value);

for (int counter = 1; counter < 20, counter++) {
    pwdseed = Hmac((x, b) \{ mina, b \}, base \{ | counter \})
    pwd_value = Kdf(seed, "SAE Hunting and Pecking", p);
    sae_is_quadratic_residue(curve, pwd_value, qr, qnr);
}

Listing 2: Hunting and Pecking on ECP group as implemented in iwd. Variable names have been adapted for a better fit.

Each type or function starting by L_* refers to a function in the Embedded Linux Library\(^7\) (ell), a minimalist cryptographic library developed by Intel. By default, this library is statically linked to the binary at compilation time. Users can decide to use a dynamic linking by specifying the correct option before compiling. We stress that the linking strategy does not impact the result of our attack; only some details in the addresses to monitor are concerned (see Section 3.3).

It is easy to notice that explicit branching at lines 15 and 16 makes the control flow input-dependent. An attacker who is able to tell at what iteration the code between line 17 and 22 is executed can guess how many rounds are needed before successfully returning from this function.

3.3 Cache-Attack Details

In order to efficiently determine at what iteration a password is successfully converted, the attackers' needs are twofold. First, they need to be able to distinguish each iteration. Second, they shall guess when the success-specific code (lines 17-22) is executed.

To achieve the first goal, we create a synchronization clock by monitoring some memory line accessed at the beginning of each loop. The call to kdf_sha256, a function of libell called inside sae_pwd_value, is a good candidate. More specifically, we monitor a memory address corresponding to the loop calling this hash function. Thanks to the complex nature of this operation, we were able to detect access to this call every time. Moreover, this particular memory address is not accessed during the rest of the protocol, thereby avoiding any potential noise in our traces.

Monitoring access to the code executed on success is less straightforward: the address range to be accessed inside sae_compute_pwd is too small and too close to the rest of the loop to be reliably monitored. The best choice is to monitor instruction in one of the functions called at lines 17 to 19. Tests have shown that monitoring inside l_getrandom yields the best results: other functions are called too often, at various places, bringing noise to our traces. However, random number generation is also part of the quadratic residue verification (sae_is_quadratic_residue, line 15) in order to blind the computation. Fortunately, these accesses can be distinguished given the number of cycles elapsed since the beginning of the iteration.

Due to complex CPU optimization techniques (see Section 2.3.2) and some system noise, the measurements are noisy and some traces may yield incorrect results. Moreover, a call to l_getrandom is usually performed in a few cycles, implying that we can miss this call due to the temporal resolution of Flush+Reload.

In order to significantly improve the reliability of our results, we combined the Flush+Reload attack with the Performance Degradation attack (PDA), as presented in [4]. Since the first call to l_getrandom occurs before the proper quadratic residue check, we evict a memory line inside the code in charge of the Legendre symbol computation. Hence, we significantly increase the delay between our synchronization clock and the success-specific code, while keeping a low delay to reach the first call to l_getrandom.

To sum up, by simply monitoring two addresses with a classic Flush+Reload technique, and repeatedly evicting a single memory address, we were able to collect traces that yield more relevant results with only a few samples.

3.4 Miscellaneous Leak

As specified in the Dragonfly RFC [16] and in the SAE standard [1], the number of iterations to perform during the password conversion is not fixed. It can be defined as any non-negative integer, providing it effectively guarantees a successful conversion with high probability. RFC 7664 advises to set $k$ to at least $40$ in order to get roughly one password over one trillion that needs more iterations.

As for iwd, the implementation sets $k = 20$, making this probability significantly lower, with about one over $2 \cdot 10^5$ passwords requiring more than $k$ iterations. In practice, using only password drawn from existing dictionaries [25, 31], we were able to find a consistent list of password needing more than 20 iterations (see Appendix A for a sample). Using these password related dictionaries, with random MAC addresses, we found an average of 33.6 passwords (9.5 $\cdot 10^{-5}$% of the dictionaries).

In this scenario, a client would be unable to authenticate to the AP until the password or the MAC address of one party is changed. From an attacker perspective, finding such a tuple provides a lot of information on the password, without the aforementioned cache-attack. Indeed, they can assume that the password needs at least 20 iterations, and perform an offline dictionary attack as described in Section 3.5. However, due to the low probability of finding these tuples, we did not take it into account in the rest of the paper.

3.5 Dictionary Partitioning

By exploiting the leakage presented above, attackers can significantly reduce the set of potential passwords with an offline brute-force program. Given a dictionary and some $m$ collected traces, it iterates over the passwords and eliminates those that do not

\(^7\)https://git.kernel.org/pub/scm/libs/ell/ell.git/
yield the same result when derived with the corresponding MAC addresses. The remaining passwords, giving the same results, are potential candidates that now constitute the new dictionary.

3.5.1 Theoretical success rate. Let each leak be represented by a tuple \((A, B, k)\) with \(A, B\) the MAC addresses and \(k \in [1, 20]\) the number of iterations. When converting a password into a group element, the success of each iteration is bounded to the success of the quadratic residue test. Let be \(p\) the order of the underlying field and \(q\) the order of the generator? Since Dragonfly only support elliptic curves of cofactor \(h = 1\), \(q\) also denotes the total number of points on the curve. Then, a random integer \(x \in [0, p)\) is a quadratic residue with probability:

\[
P_x = \frac{q}{2p} \approx 0.5 \approx 1 - p_x.
\]

The input of the quadratic residue is considered random (being the output of a KDF). Hence, each iteration is independent of the others if we model the KDF as a random oracle. Let \(X\) denote the random variable representing the number of iterations of a trace, and \(k \in [1, 20]\):

\[
\Pr[X = k] = p_x^k.
\]

The probability for a trace to eliminate any tested password depends on the number of iterations \(k\). Let \(Y_i\) be the random variable representing the success (1) or the failure (0) of a password to pass each test in a single trace. We got \(Y_i = 1\) only if the password succeeds all tests, i.e. with probability \(\Pr[X = k]\), hence:

\[
\Pr[Y_1 = 0 \mid X = k] = 1 - \Pr[X = k] = 1 - p_x^k.
\]

More generally, the probability for a password to be eliminated by a random trace is:

\[
\Pr[Y_1 = 0] = \sum_{i=1}^{20} \Pr[X = i] \cdot \Pr[Y_1 = 0 \mid X = i].
\]

Hence, the probability for a password to be pruned by at most \(n\) traces is the sum of probabilities for it be pruned either at the first trace or to pass the first and be pruned at the second, and so forth:

\[
p_y = \Pr[Y_n = 0] = \sum_{i=0}^{n-1} \Pr[Y_1 = 0] \cdot (1 - \Pr[Y_1 = 0])^i.
\]

Let \(L\) be the size of our dictionary, and \(d\) be the number of passwords we want to eliminate. Let \(Z_n\) be the number of passwords we remove using \(n\) traces. Since tests behave as independent trials, \(Z_n\) follows a binomial law, hence:

\[
\Pr[Z_n \geq d] = \sum_{i=d}^{L} \binom{L}{i} \cdot p_y^i \cdot (1 - p_y)^{L-i}.
\]

Table 1 gives an overview of the number of traces required to eliminate all wrong passwords from different dictionaries, with a probability greater than 0.95. We outline the benefit of our attack compared to the original Dragonblood’s, reducing the average number of required traces by roughly 43%. In practice, we do not need to remove all passwords from the dictionary, we only need to reduce it enough, so that remaining passwords can be tested in an active attack. Keeping more passwords in the dictionary would reduce the number of required traces.

3.5.2 Complexity of the offline search. Each test we perform is bounded by the complexity of a quadratic residue test (which is basically a modular exponentiation). The theoretical cost of such an operation has already been discussed in [35], and can be applied the same way in our context. Authors estimated, given their benchmark of the PowMod function [27] on an NVIDIA V100 GPU, that approximately \(7.87 \cdot 10^8\) passwords per second can be tested. Since each test is independent, the amount of parallelization is up to the attacker capacity, and can be higher. Namely, one can choose to split the dictionary into \(k\) pieces and run \(k\) instances of the dictionary reducer.

4 EXPERIMENTAL RESULTS

In this section, we describe our setup and give details about the experimental results we obtained during our evaluation. All the scripts and programs we used are made open-source\(^5\).

4.1 Experimental Setup

Our experiments were performed on a Dell XPS13 7390 running on Fedora 31, kernel 5.6.15, with an Intel® Core(TM) i7-10510U and 16 GB of RAM. Binaries were compiled with gcc version 9.3.1 build 20200408 using the default configuration (optimization included). Namely, the Embedded Linux Library version 0.31 was statically linked to iwd during compilation.

During our experiment, we deployed hotspotd (version 2.9) as an Access Point, and iwd (version 1.7) as a client. Both were installed and launched on the same physical device, using emulated network interfaces, as described in [26].

We kept the default configuration on both ends, meaning the key exchange is always performed using IKE group 19, corresponding to P256. Similar results would have been observed using group 20 (curve P384) by tweaking the threshold of our spy process.

Our spy process has been implemented by following classical Flush+Reload methods. Moreover, we used Mastik v0.02 implementation of the FDA [36].

4.2 Trace Collection

Once both client and AP were setup to use a password that was randomly drawn from a dictionary, we launched the spy process to monitor well-chosen memory lines (see Section 3.3). After each connection, we disconnected the client and reconnected it a few times to acquire multiple samples. This step emulates a de-authentication attack aiming at collecting multiple samples with the same MAC.

<table>
<thead>
<tr>
<th>Dict. size</th>
<th>Avg traces</th>
<th>Avg traces in [35]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rockyou</td>
<td>1.4 \cdot 10^7</td>
<td>16</td>
</tr>
<tr>
<td>CrackStation</td>
<td>3.5 \cdot 10^7</td>
<td>17</td>
</tr>
<tr>
<td>HaveIBeenPwned</td>
<td>5.5 \cdot 10^8</td>
<td>20</td>
</tr>
<tr>
<td>8 characters</td>
<td>4.6 \cdot 10^{14}</td>
<td>32</td>
</tr>
</tbody>
</table>

Table 1: A Comparison of the Number of the Required Traces to Prune all Wrong Passwords Between Our attack and Dragonblood.

\(^5\)https://gitlab.inria.fr/ddealmei/poc-iwd-acsac2020/-/tree/master/
addresses. For each password we went through this process using 10 different MAC addresses, allowing us to acquire up to 10 independent traces for the same password. For each MAC address, we collect 15 samples. Our observations were consistently obtained through testing 80 passwords in order to evaluate the effectiveness and the reliability of our trace collection techniques.

We call sample the result of monitoring one Dragonfly key exchange, with a fixed password and MAC addresses. It is represented by succession of lines, corresponding to either a call to the synchronization clock (kdf_sha256) or to l_getrandom. The value following each label is an indicator of the delay since the last call to the synchronization clock. An example can be found in Appendix B, corresponding to a trace yielding four iterations. A trace is a collection of samples, all corresponding to the same password and the same MAC address.

4.3 Trace Interpretation

We also designed a script that automatically interprets our traces and outputs the most probable iteration in which the process of password conversion first succeeds.

The trace parser process is described in Listing 3. The core idea is to first reduce the noise by eliminating all poorly formed samples (which could not be interpreted anyway, often because of system noise). Then, each sample is processed independently, contributing to the creation of a global trace score. To do so, each line of a sample is read, and depending on the corresponding label, it is processed as follow: (i) if the label is the synchronization clock, we increase the iteration counter by one; (ii) otherwise, the score of the current iteration is increased by the delay associated to that line. In the latter case, if the delay is long enough (the threshold may be architecture specific), we can stop the parsing of that sample and process the next one. Once every sample of a trace has been processed, the score of each iteration comes at as indicator of the most probable successful iteration.

Since false positives have severe consequences, we chose to eliminate any trace that does not yield a clear result. In such a case, the script raises a warning to the attacker for future manual interpretation.

1 def parse_measures(measures):
2    score = [0 for i in range(k+1)]
3    for m in measures:
4        if is_malformed(m):
5            continue
6        # Increments score with the observed delay at each iteration
7        parse_measure(m, score)
8
9    # Convert the score of each iteration into frequency
10    freq = []
11    total_score = sum(score)
12    while sum(score) > 0:
13        m = max(score)
14        freq.append((round(m*100.0/total_score), 2))
15        res[res.index(m)] = 0
16
17    # Raise a warning if we are not sure of the result
18    if freq[0][1] - freq[1][1] <= 15:
19        warning = True
20    print("WARNING! Not quite sure of the result ...")
21    for x in freq:
22        print("{} ({})".format(x[0], x[1]))

Listing 3: Trace parsing script.

4.4 Results

We summed-up the results of our experimentations, with different number of samples for each MAC address, in Figure 2. With only one measurement per address, approximately 70.5% of the traces can be automatically interpreted (others have a high risk of miss-prediction). However, the accuracy of our prediction is only 66%. We need to collect 5 samples to achieve an accuracy greater than 90% (with 77% of usable traces). We achieve 99% accuracy with only 10 measurements, with a trace usability of 88%.

We stress that trace usability only represents the ability for the parser to automatically interpret the trace. For most warnings, a manual reading of the samples (about 1-2 minutes) allows attackers to successfully predict the round (some measurements do not yield a clear result, and should be ignored). We also note that even if our script was unable to decide between two adjacent values, e.g. five and six, we can assume that more than four iterations are required for password conversion.

These results outline the improvement of our attack compared to Dragonblood. In [35], at least 20 samples were needed for each MAC address to achieve a success rate of 99% (only 10 in our attack). Moreover, with our attack, each successfully interpreted trace gives at least as much information, and roughly twice more on average (see Section 3.5). Consequently, our work greatly reduces the number of the required measurements (or samples) in order to prune all wrong passwords in a given dictionary. For instance, our work needs 160 measurements for the Rockyou dictionary, while Dragonblood needs 580 measurements. Roughly speaking, the measurements are cut down by at least 3. Moreover, our attack requires to vary the MAC addresses less often (almost twice as fewer). Thus, our work performs better in practice, particularly in a context where cache-based measurements are limited. Of course, we argue that our results can be generalized for other implementations suffering from the same type of vulnerability.
5 DISCUSSION AND CONCLUSION

5.1 Recommendations for Mitigations

Following the disclosure of Dragonblood, several mitigations have been proposed [17, 18] to replace the iterative hash-to-group function by a deterministic function. This countermeasure suits our requirements. However, backward compatibility might be a requirement in industry. Hence, we suggest to use a branch-free implementation of the loop in order to avoid any residual leakage.

Listing 4: Python-like pseudocode of a constant time version of Hunting and Pecking on P256.

We implemented such mitigations into iw2d (see Listing 4), inspiring ourselves from hostapd patch\(^9\). We estimated the overhead induced by such countermeasure using the rdtscl assembly instruction, which offers very high precision. We made 10,000 measurements for both the mitigated derivation and the original one, while varying the password. We observed a negligible overhead (1.4 \times 10^{-2} \% on average). The code complexity is barley affected by our changes. Considering the attack impact and the negligible downside of the patch, we strongly recommend developers to include it in their products. Following our discoveries, both iw2d and FreeRADIUS has smoothly integrated our patch in their code.

5.2 Discussion

After the original Dragonblood publication, implementations received various patches, and dropped the support of some curves (mainly Brainpool curves). However, the main source of vulnerabilities, the hash-to-group function, is still unchanged, despite the standards update.

In spite of proper branch-free implementations being publicly available, with a negligible overhead, most implementations did not patch the secret-dependent control-flow of the password derivation. We believe the lack of patch is strongly related to the lack of Proof of Concept dedicated to specific implementations. Dragonblood only describes the attack for hostapd which has been fixed.

We demonstrated that this vulnerability has more potential than the original one, allowing to recover more bits of information with fewer measurements. We provide a full Proof of Concept of our vulnerability on Intel’s implementation, but we believe it can extend to others (see Appendix C). Our approach illustrates the risk to users when cryptographic software developers dismiss a widely offered side-channel countermeasure using the application of such countermeasure.

ACKNOWLEDGMENTS

Daniel De Almeida Braga is funded by the Direction Générale de l’Armement (Pôle de Recherche CYBER). We would like to thank the anonymous paper and artifact reviewers for their time and constructive feedbacks.

REFERENCES


\[5\] Wi-Fi Alliance. 2019. WPA3 Security Considerations.


---

9https://w1.fi/security/2019-1/
Here is a sample of passwords requiring more than 20 iterations to be successfully derived into a point on P-256. MAC addresses are noted at the beginning of each list; the needed number of iterations is at the end of the line.

An extended list can be found in our gitlab repository.10

## CF116C758375 553ED5460AA7 ##

### th-commando-regiment 24

thecircleofbanks 21

## 992606B4AD9F FFF23027CB34 ##

### RAJARATNAM 21

RA–KLEINENBERG 22

ellochika 21

VILIFYINGLY 24

believeingod1 24

BELLABOOBAE 25

PRERRAELISTA 21

DOGYLEO1 21

macarthurreviews 23

AMERICANHOOSPICE 21

CHALLENGE 22

HAUNTEDEP 21

Nibbler112 21

0800581064 22

SAKHLIKIS 21

UPDMDFDR48 26

kanakaman 30

OXNWRABBS 25

0874739218 23

DEPEFCQ56 22

taxidermically 21

38concert 21

NONPARISHIONER 22

NOOMMY7685 21

gramocelj 21

YUNKALLAH 22

MILE–MICHEL–HYACINTHE 23

STICKHANDLING 22

faras–071196 21

FARNHAM69 22

1023197616R 21

1102001625160 23

winjsbyk46 24

veroleg351 27

cutelildevilj87 21

cutelildevilj87 21

# 992606B4AD9F FFF23027CB34 ##

### th-commando-regiment 24

thecircleofbanks 21

## 992606B4AD9F FFF23027CB34 ##

### RAJARATNAM 21

RA–KLEINENBERG 22

ellochika 21

VILIFYINGLY 24

believeingod1 24

BELLABOOBAE 25

PRERRAELISTA 21

DOGYLEO1 21

macarthurreviews 23

AMERICANHOOSPICE 21

CHALLENGE 22

HAUNTEDEP 21

Nibbler112 21

0800581064 22

SAKHLIKIS 21

UPDMDFDR48 26

kanakaman 30

OXNWRABBS 25

0874739218 23

DEPEFCQ56 22

taxidermically 21

38concert 21

NONPARISHIONER 22

NOOMMY7685 21

gramocelj 21

YUNKALLAH 22

MILE–MICHEL–HYACINTHE 23

STICKHANDLING 22

faras–071196 21

FARNHAM69 22

1023197616R 21

1102001625160 23

winjsbyk46 24

veroleg351 27

elasticized 21

cutelildevilj87 21

JLNRUJY98 25

FENWICK–1994 21

DEFENDERS 21

2C400000 21

## 992606B4AD9F FFF23027CB34 ##

### th-commando-regiment 24

thecircleofbanks 21

## 992606B4AD9F FFF23027CB34 ##

### th-commando-regiment 24

thecircleofbanks 21


### A PASSWORD REQUIREING MORE THAN 20 ITERATIONS ON IWD

Here is a sample of passwords requiring more than 20 iterations to be successfully derived into a point on P-256. MAC addresses are

10https://gitlab.inria.fr/ddealmei/poc-iwd-ac sac2020/-/blob/master/data/results/buggy_passwords.txt
Sample of a trace yielding four iterations. This has been acquired using the password `superpassword`, with MAC addresses E2F754FE22D1 and 9203835A576B. Annotations have been added and are not part of the original trace.

# First five lines correspond to the qr and qnr generation
# They are ignored during parsing
l_getrandom 5435937 (90)
l_getrandom 5439791 (88)
l_getrandom 5443732 (96)
l_getrandom 5447611 (88)
l_getrandom 5455232 (88)
# Here the loop begins
kdf_sha256 5459308 (82)
kdf_sha256 3324 (86)
kdf_sha256 4891 (82)
kdf_sha256 3972 (84)
l_getrandom 108 (90)
# At the fourth iteration, we notice long-delayed call
# to l_getrandom. It means we can stop there.
l_getrandom 3889 (88)

C ATTACK ON FREERADIUS

FreeRADIUS supports EAP-pwd, a variant of Dragonfly, as a non-default authentication method, encapsulated in the RADIUS protocol. Beside the patches to Dragonblood attacks, we show that EAP-pwd is still vulnerable to timing attacks (due to a variable number of iterations), and to the same cache attack we described in Section 3. In this section, we studied the last version of FreeRADIUS (v3.0.21 at the time of writing).

C.1 EAP-pwd vs SAE

SAE and EAP-pwd being two variants of Dragonfly, they differ in a few points. Some of them are only instantiation details (values of some labels), while others have more impactful consequences on the workflow and the security of the protocol.

First, EAP-pwd standard does not mandate a constant number of iterations. Indeed, it exits the conversion loop as soon as the password is successfully converted. Since a constant number of iterations would not change the outcome of the conversion, some implementations (not FreeRADIUS) include this side-channel mitigation anyway.

Next, EAP-pwd does not benefit from the same symmetry as SAE: client and server are clearly defined. This distinction is highlighted by the fact that the server generates a random token for each new session. This token will be part of the information hashed at each iteration during the password conversion. Hence, while a password is always derived into the same element in SAE (as long as the identities do not change), each EAP-pwd session ends up with a new group element, due to the randomness brought by the token.

C.2 FreeRadius implementation

The Dragonfly exchange implemented by FreeRadius follows EAP-pwd’s specification [40]. All related functions are defined in the according module\textsuperscript{11}. Namely, the Hunting and Pecking is implemented in the function compute_password_element, as illustrated in Listing 5. We cut some parts of the code, and renamed variables for the sake of clarity.

\textsuperscript{11}https://github.com/FreeRADIUS/freeradius-server/tree/v3.0.x/src/modules/rlm_eap/types/rlm_eap_pwd
With a single measurement, we successfully guessed the exact number of iterations needed to convert the password, even if the total number of iterations is fixed.

C.3 Cache-Attack Against FreeRADIUS

Using some minor adaptations, we applied our cache attack (described on iwd in Section 3) to guess the exact iteration in which the password is successfully derived. We stress that switching to a constant number of iterations, with a constant time (or masked) Legendre symbol computation, would mitigate the timing attack, but our cache attack would still be practical.

We perform this attack by only monitoring two memory lines, both in the OpenSSL cryptographic library. To do so, we use the calls to H_Update (called line 13 to 17) as a synchronization clock. Since multiple calls to this function follow each other, we catch them with high probability. Next, we use the call to EC_POINT_is_on_curve (line 46) as a success-specific code. More specifically, this function calls set_affine_coordinates from OpenSSL internals, which is also called if the original check (line 42) is successful success. Thus, some piece of code is called twice on success, and is never called on failure.

C.4 Experimental results

We implemented a full Proof of Concept of our attack, and made it publicly available after the vulnerability has been patched. The experimental setup is the same as described in Section 4.1.

Due to the server-generated token, we only have a single measurement to guess how many iterations are needed to convert the password. We tested our attack on 80 different passwords, spying on 15 iterations for each password, yielding a total of 1200 traces. With a single measurement, we successfully guessed the exact number of iterations for 93% of the traces. We outline some consistency in the errors: most errors occurred because the spy process misses on call to the synchronization clock. Hence, we can achieve a better reliability by losing some information: assuming that if we guess that the password needs $x$ iterations to be converted, then it may need $x$ or $x + 1$ iterations, allowing us to reach 99% accuracy.

Considering we achieve this accuracy with a single measurement, we are able to recover a password with fewer measurements than in previous attacks, even by softening our guess.

---

Listing 5: FreeRADIUS code sample, extracted from eap_pwd.c.

```c
int compute_password_element(pwd_session_t *session, uint16_t grp_num, char const *pwd, int pwd_len, char const *id_server, char const *id_peer, uint32_t *token)
{
    // Instantiation of some variables and contexts ...
    /∗ Clean allocated memory and handle errors ... ∗/

    ctr = 0;
    while (1) {
        if (ctr > 100)
            goto fail;
        ctr++;
        if (pwd seed = H(token | peer_id | server_id | pwd | ctr))
            H_Init(ctx);

        H_Update(ctx, (uint8_t*)token, sizeof(token));
        H_Update(ctx, (uint8_t*)id_peer, id_peer_len);
        H_Update(ctx, (uint8_t*)id_server, id_server_len);
        H_Update(ctx, (uint8_t*)password, password_len);
        H_Update(ctx, (uint8_t*)ctr, sizeof(ctr));
        H_Final(ctx, pwe_digest);[

        prbuf = KDHpwe_digest, "EAP−pwd Hunting And Pecking", prbuf)
        BN_bin2bn_digest, SHA256_DIGEST_LENGTH, pwd);

        if (EAP pwd_kdfpwe_digest, SHA256_DIGEST_LENGTH, "EAP−pwd Hunting And Pecking", prbuf, primebitlen) > 0)
            goto fail;
        BN_bin2bn(prbuf, primebytelen, x_candidate);

        // Handle BN conversion issue ...
        if (primebytelen % 8)
            BN_bin2bn(x_candidate, x candidate, ( 8 − (primebytelen % 8)));
        if (BN_ucmp(x candidate, session−prime) == 0)
            continue;

        /∗
        • need to unambiguously identify the solution, if there is
        • one ...
        /∗
        • solve the quadratic equation, if it’s not solvable then we
        • don’t have a point
        /∗
        if (EC_POINT_set_compressed_coordinates_G1(session−group, session−pwe, x candidate, is_odd, NULL))
            continue;

        // Check if the point is on the curve
        if (EC_POINT_is_on_curve(session−group, session−pwe, NULL))
            continue;

        if (BN_cmp(cofactor, BN_value_one()))
            /∗ make sure the point is not in a small subgroup ∗
        if (EC POINT_mul(session−group, session−pwe, NULL, session−pwe, cofactor, NULL))
            continue;

        if (EC POINT_is_at_infinity(session−group, session−pwe))
            continue;

        /∗ if we got here then we have a new generator ∗
        break;
    }

    /∗ Clean allocated memory and handle errors ... ∗/[
```

This implementation heavily relies on OpenSSL to perform cryptographic operations, such as hashing, manipulating big integers and elliptic curve points. By default, the library is dynamically linked from the system-wide installation when building the project.

---

12https://www.openssl.org/

13https://gitlab.imria.fr/msabt/attack-poc-freeradius