Dragonblood: Analyzing the Dragonfly Handshake of WPA3 and EAP-pwd

Mathy Vanhoef and Eyal Ronen

Real World Crypto, New York, 10 January 2020.
Background: Wi-Fi Security

› **1999:** Wired Equivalent Privacy (WEP)
  › **Broken** in 2001 [FMS01]

› **2003:** Wi-Fi Protected Access (WPA)

› **2004:** Wi-Fi Protected Access 2 (WPA2)
  › Allows **offline password brute-force**
  › KRACK and Kraken attack [VP][2017-8]
Background: Dragonfly in WPA3 and EAP-pwd

= Password Authenticated Key Exchange (PAKE)

Provide mutual authentication

Negotiate session key

Prevent offline dictionary attacks
Dragonfly

Pick random $r_A$ and $m_A$

$s_A = (r_A + m_A) \mod q$

$E_A = -m_A \cdot P$

Pick random $r_B$ and $m_B$

$s_B = (r_B + m_B) \mod q$

$E_B = -m_B \cdot P$

Convert password to group element $P$
Commit($s_A, E_A$)

Pick random $r_A$ and $m_A$
$s_A = (r_A + m_A) \mod q$
$E_A = -m_A \cdot P$

Commit($s_B, E_B$)

Pick random $r_B$ and $m_B$
$s_B = (r_B + m_B) \mod q$
$E_B = -m_B \cdot P$

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 $E_A$
$K = r_B \cdot (s_A \cdot P + E_A)$
$\kappa = \text{Hash}(K)$
$tr = (s_B, E_B, s_A, E_A)$
$c_B = \text{HMAC}(\kappa, tr)$
Pick random $r_A$ and $m_A$

$s_A = (r_A + m_A) \mod q$

$E_A = -m_A \cdot P$

Pick random $r_B$ and $m_B$

$s_B = (r_B + m_B) \mod q$

$E_B = -m_B \cdot P$

**Commit($s_A, E_A$)**

**Negotiate shared key**

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 $E_A$

$K = r_B \cdot (s_A \cdot P + E_A)$

$\kappa = \text{Hash}(K)$

$tr = (s_B, E_B, s_A, E_A)$

$c_B = \text{HMAC}(\kappa, tr)$
Verify $s_B$ and $E_B$

$K = r_A \cdot (s_B \cdot P + E_B)$

$\kappa = \text{Hash}(K)$

$tr = (s_A, E_A, s_B, E_B)$

$c_A = \text{HMAC}(\kappa, tr)$

Confirm peer negotiated same key

Verify $s_A$ and $E_A$

$K = r_B \cdot (s_A \cdot P + E_A)$

$\kappa = \text{Hash}(K)$

$tr = (s_B, E_B, s_A, E_A)$

$c_B = \text{HMAC}(\kappa, tr)$
How to derive P from a password?

1. MODP groups
2. Elliptic curves
How to derive P from a password?

1. MODP groups
2. Elliptic curves
Hash-to-curve: EAP-pwd

for (counter = 1; counter < 40; counter++)
    x = hash(pw, addr1, addr2, counter)
    if x >= p: continue
    if square_root_exists(x) and not P:
        return (x, \sqrt{x^3 + ax + b})
Hash-to-curve: EAP-pwd

for (counter = 1; counter < 40; counter++)
    x = hash(pw, addr1, addr2, counter)
    if x >= p: continue
    if square_root_exists(x) and not P:
        return (x, \sqrt{x^3 + ax + b})

Half of x values aren't on the curve
Hash-to-curve: EAP-pwd

for (counter = 1; counter < 40; counter++)
    x = hash(pw, addr1, addr2, counter)
    if x >= p: continue
    if square_root_exists(x) and not P:
        return (x, \sqrt{x^3 + ax + b})
Hash-to-curve: EAP-pwd

for (counter = 1; counter < 40; counter++)
    x = hash(pw, addr1, addr2, counter)

if x >= p: continue

if square_root_exists(x) and not P:
    return (x, √x^3 + ax + b)

#iterations depends on password
(and public MAC addresses)
Hash-to-curve: EAP-pwd

for (counter = 1; counter < 40; counter++)
    x = hash(pw, addr1, addr2, counter)

if x >= p: continue

if square_root_exists(x) and not P:
    return (x, 𝑥^3 + ax + b)

# iterations depends on password
(and public MAC addresses)

No timing leak countermeasures, despite warnings by IETF & CFRG!
Attacking Clients
Attacking Access Points
Leaked information: #iterations needed

<table>
<thead>
<tr>
<th>Client address</th>
<th>addrA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Measured</td>
<td></td>
</tr>
</tbody>
</table>
Leaked information: #iterations needed

<table>
<thead>
<tr>
<th>Client address</th>
<th>addrA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Measured</td>
<td></td>
</tr>
<tr>
<td>Password 1</td>
<td></td>
</tr>
<tr>
<td>Password 2</td>
<td></td>
</tr>
<tr>
<td>Password 3</td>
<td></td>
</tr>
</tbody>
</table>
# Leaked information: #iterations needed

<table>
<thead>
<tr>
<th>Client address</th>
<th>addrA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Measured</td>
<td></td>
</tr>
<tr>
<td>Password 1</td>
<td></td>
</tr>
<tr>
<td>Password 2</td>
<td></td>
</tr>
<tr>
<td>Password 3</td>
<td></td>
</tr>
</tbody>
</table>
What information is leaked?

for (counter = 1; counter < 40; counter++)
    x = hash(pw, addr1, addr2, counter)

    if x >= p: continue

    if square_root_exists(x) and not P:
        return (x, \sqrt{x^3 + ax + b})

Spoof client address to obtain different execution & leak new data
Leaked information: #iterations needed

<table>
<thead>
<tr>
<th>Client address</th>
<th>addrA</th>
<th>addrB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Measured</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Password 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Password 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Password 3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Leaked information: #iterations needed

<table>
<thead>
<tr>
<th>Client address</th>
<th>addrA</th>
<th>addrB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Measured</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Password 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Password 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Password 3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Leaked information: #iterations needed

<table>
<thead>
<tr>
<th>Client address</th>
<th>addrA</th>
<th>addrB</th>
<th>addrC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Measured</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Password 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Password 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Password 3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Leaked information: #iterations needed

<table>
<thead>
<tr>
<th>Client address</th>
<th>addrA</th>
<th>addrB</th>
<th>addrC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Measured</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Forms a signature of the password**

Need **~17 addresses** to determine password in RockYou (~$10^7$) dump
Raspberry Pi 1 B+: differences are measurable
Raspberry Pi 1 B+: differences are measurable

EAP-pwd client:

~30 measurements / address

Using Crosby’s box test
Hash-to-curve: EAP-pwd

for (counter = 1; counter < 40; counter++)
    x = hash(pw, counter, addr1, addr2)
    if x >= p: continue
    if square_root_exists(x) and not P:
        return (x, \sqrt{x^3 + ax + b})
Hash-to-curve: WPA3

for (counter = 1; counter < 40; counter++)
    x = hash(pw, counter, addr1, addr2)
    if x >= p: continue
    if square_root_exists(x) and not P:
        P = (x, $\sqrt{x^3 + ax + b}$)
        pw = rand()
return P

WPA3: always do 40 loops & return first P
Hash-to-curve: WPA3

for (counter = 1; counter < 40; counter++)
    x = hash(pw, counter, addr1, addr2)
    if x >= p: continue
    if square_root_exists(x) and not P:
        \[ P = (x, \sqrt{x^3 + ax + b}) \]

return P
Hash-to-curve: WPA3

for (counter = 1; counter < 40; counter++)
    x = hash(pw, counter, addr1, addr2)
    if x >= p: continue

if square_root_exists(x) and not P:
    P = (x, \sqrt{x^3 + ax + b})
    pw = rand()

return P

Extra iterations based on random password
for (counter = 1; counter < 40; counter++)
    \( x = \text{hash}(pw, \text{counter}, \text{addr1}, \text{addr2}) \)

    if \( x \geq p \): continue

    if \( \text{square_root_exists}(x) \) and not \( P \):
        \( P = (x, \sqrt{x^3 + ax + b}) \)

    pw = \text{rand}()

return \( P \)
Hash-to-curve: WPA3

for (counter = 1; counter < 40; counter++)
    x = hash(pw, counter, addr1, addr2)
    if x >= p: continue
    if square_root_exists(x) and not P:
        P = (x, \sqrt{x^3 + ax + b})

Brainpool: \( p = 0xA9FB57DBA1EEA9BC... \)

\( \rightarrow \) High chance that \( x \geq p \)
for (counter = 1; counter < 40; counter++)
    x = hash(pw, counter, addr1, addr2)
    if x >= p: continue
    if square_root_exists(x) and not P:
        P = (x, \sqrt{x^3 + ax + b})
        pw = rand()
return P
Hash-to-curve: WPA3

for (counter = 1; counter < 40; counter++)
    x = hash(pw, counter, addr1, addr2)
    if x >= p: continue
    if square_root_exists(x) and not P:
        P = (x, \sqrt{x^3 + ax + b})
    pw = rand()
return P

Code may be skipped
Hash-to-curve: WPA3

for (counter = 1; counter < 40; counter++)
    x = hash(pw, counter, addr1, addr2)
    if x >= p: continue
    if square_root_exists(x) and not P:
        P = (x, \sqrt{x^3 + ax + b})
        pw = rand()
    return P

#Times skipped depends on password
Hash-to-curve: WPA3

```
for (counter = 1; counter < 40; counter++)
  x = hash(pw, counter, addr1, addr2)
  if x >= p: continue
  if square_root_exists(x) and not P:
    P = (x, \sqrt{x^3 + ax + b})
  pw = rand()
return P
```

#Times skipped depends on password & random password in extra iterations
for (counter = 1; counter < 40; counter++)
    x = hash(pw, counter, addr1, addr2)
    if x >= p: continue
    if square_root_exists(x) and not P:
        P = (x, \sqrt{x^3 + ax + b})
pw = rand()
return P

Variance ~ when password element was found
Hash-to-curve: WPA3

for (counter = 1; counter < 40; counter++)
    x = hash(pw, counter, addr1, addr2)
    if x >= p: continue
    if square_root_exists(x) and not P:
        P = (x, \sqrt{x^3 + ax + b})
pw = rand()
return P

Variance ~ when password element was found
Average ~ when found & #iterations code skipped
Raspberry Pi 1 B+
Raspberry Pi 1 B+

WPA3 AP (Hostap):

~300 measurements / address

Using Crosby’s box test
Cache Attacks
Threat Model
Threat Model
Cache attack on NIST curves

for (counter = 1; counter < 40; counter++)
    x = hash(pw, counter, addr1, addr2)
    if x >= p: continue

if square_root_exists(x) and not P:
    P = (x, x^3 + a*x + b)

NIST: p = 0x0FFFFFFF00000001000000000000

→ Negligible chance that x >= p

return P
Cache attack on NIST curves

```python
for (counter = 1; counter < 40; counter++)
    x = hash(pw, counter, addr1, addr2)
    if x >= p: continue
    if square_root_exists(x) and not P:
        P = (x, \sqrt{x^3 + ax + b})
        pw = rand()
return P
```

NIST curves: use Flush+Reload to detect when code is executed
for (counter = 1; counter < 40; counter++)
    x = hash(pw, counter, addr1, addr2)
    if x >= p: continue
    if square_root_exists(x) and not P:
        P = (x, \sqrt{x^3 + ax + b})
    pw = rand()
return P

NIST curves: use Flush+Reload to detect when code is executed
Attacking client: Intel Core i7-7500

![Graph showing probability of value for nQR and QR. The x-axis represents the value ranges from 5 to 14, and the y-axis represents the probability of value ranging from 0.0 to 0.6. The graph compares the distributions for nQR (red bars) and QR (blue bars).]
Attacking client: Intel Core i7-7500

WPA3 client (Hostap):

~20 measurements / address

Using Linear Classifier
# Password Brute-force Cost

<table>
<thead>
<tr>
<th>Group</th>
<th>Dictionary Size</th>
<th>$ for MDP 22</th>
<th>$ for MDP 28</th>
<th>$ for P-256</th>
</tr>
</thead>
<tbody>
<tr>
<td>RockYou [20]</td>
<td>$1.4 \cdot 10^7$</td>
<td>$2.1 \cdot 10^{-6}$</td>
<td>$4.4 \cdot 10^{-4}$</td>
<td></td>
</tr>
<tr>
<td>HaveIBeenPwned [45]</td>
<td>$5.5 \cdot 10^8$</td>
<td>$8.0 \cdot 10^{-5}$</td>
<td>$1.7 \cdot 10^{-2}$</td>
<td></td>
</tr>
<tr>
<td>Probable Wordlists [12]</td>
<td>$8.0 \cdot 10^9$</td>
<td>$1.2 \cdot 10^{-3}$</td>
<td>$2.5 \cdot 10^{-1}$</td>
<td></td>
</tr>
<tr>
<td>8 Low Case</td>
<td>$2.1 \cdot 10^{11}$</td>
<td>$3.0 \cdot 10^{-2}$</td>
<td>6.5</td>
<td></td>
</tr>
<tr>
<td>8 Letters</td>
<td>$5.3 \cdot 10^{13}$</td>
<td>7.8</td>
<td>$1.7 \cdot 10^{3}$</td>
<td></td>
</tr>
<tr>
<td>8 Alphanumeric</td>
<td>$2.2 \cdot 10^{14}$</td>
<td>$3.2 \cdot 10^{1}$</td>
<td>$6.7 \cdot 10^{3}$</td>
<td></td>
</tr>
<tr>
<td>8 Symbols</td>
<td>$4.6 \cdot 10^{14}$</td>
<td>$6.7 \cdot 10^{1}$</td>
<td>$1.4 \cdot 10^{4}$</td>
<td></td>
</tr>
</tbody>
</table>
Implementation
Inspection
Other Implementation Vulnerabilities

**Bad randomness:**
- Can recover password element $P$
- With WPA2 bad randomness has lower impact!

**Invalid curve attack:**
- Attacker sends point not on curve
- Recover session key & bypass authentication
Wi-Fi Specific Attacks
Denial-of-Service Attack

AP converts password to EC point when client connects

› Conversion is computationally expensive (40 iterations)
› Forging 8 connections/sec saturates AP’s CPU
Downgrade Attacks

Transition mode: **WPA2/3 use the same password**

› WPA2’s handshake detects downgrades
› Performing partial WPA2 handshake → **dictionary attacks**

Handshake can be performed with multiple curves

› Initiator proposes curve & responder accepts/rejects
› **Spoof reject messages to downgrade** used curve
Disclosure
Disclosure process

Notified parties early with hope to influence WPA3

Reaction of the Wi-Fi Alliance

› Privately created backwards-compatible security guidelines
› 2nd disclosure round to address Brainpool side-channels
› Nov 2019: Updated guidelines now prohibit Brainpool curves
Latest Wi-Fi Alliance guidelines (Nov 2019)

- SAE implementations must **avoid differences in code execution that allow side channel information collection through the cache** (see Cache-Based Elliptic Curve Side-Channels).

- If WPA3-Personal Transition Mode does not meet the security requirements for a deployment, WPA3-Personal and WPA2™-Personal should be deployed on individual service set identifiers (SSIDs) using unique passwords and logically separated/isolated network segments (see WPA3-Personal Transition Mode).

**Failure to implement these recommendations correctly may expose the vendor implementation to attack and/or compromise the network.**

- “implementations must avoid [..] side-channels”
- If WPA3-Transition “doesn’t meet security requirements”, then separate passwords
- “Failure to implement...” → how can it be checked?
Fundamental issue still unsolved

- Hard to implement in constant time
- On lightweight devices, doing 40 iterations is too costly

Draft IEEE 802.11 standard has been updated

- Exclude MAC addresses from hash2curve
  - Allows offline computation of password element
- Now uses constant-time hash2curve
- Explicitly prohibit use of weak EC & MODP groups
- Prevent crypto group downgrade attack
Remaining issues

Message *transcript is not included in key derivation*
  › Prevents formal proof of protocol
  › High risk of implementation issues
    › E.g. prevention of crypto group downgrade attack

Downgrade to WPA2
  › **Not addressed in the standard**
  › Up to vendor whether to implement trust-on-first-use
    › Done by Android & NetworkManager of Linux
Issue 2: not backwards-compatible

Might lead to WPA3.1?

› Not yet clear how Wi-Fi Alliance will handle this

› **Risk of downgrade attacks** to original WPA3

Should you switch to WPA3?

› WPA2 is trivial to attack... so yes.
Conclusion

- WPA3 vulnerable to side-channels
- Countermeasures are costly
- Draft 802.11 standard updated
- Issues could have been avoided!

https://wpa3.mathyvanhoef.com
Thank you! Questions?

› WPA3 vulnerable to side-channels
› Countermeasures are costly
› Draft 802.11 standard updated
› Issues could have been avoided!

https://wpa3.mathyvanhoef.com