Packin’ the PMK
Of the robustness of WPA/WPA2 authentication

Cédric Blancher and Simon Marechal

cedric.blancher@eads.net
Computer Security Research Lab
EADS Innovation Works

simon@banquise.net
Special guest
Undisclosed entity ;)

BA-Con - September 30th - October 1st 2008
http://ba-con.com.ar/
Agenda

1. WPA/WPA2 authentication

2. WPA-PSK assessment
   - How does that work?
   - Theoretical attack cost
   - Implementation comparisons
   - Passphrase strength assessment
   - Limits of practical attacks

3. WPA-EAP thoughts
   - EAP authentication
   - Pwning the Master Key
   - Practical considerations

4. Conclusion
Introduction

Wi-Fi security...

- WEP is crippled and broken
- WPA came up to replace it
- Now, we have WPA2
Introduction

Wi-Fi security...
- WEP is crippled and broken
- WPA came up to replace it
- Now, we have WPA2

Questions
- What are WPA and WPA2 good at?
- How long will they stand?
Agenda

1. WPA/WPA2 authentication

2. WPA-PSK assessment
   - How does that work?
   - Theoretical attack cost
   - Implementation comparisons
   - Passphrase strength assessment
   - Limits of practical attacks

3. WPA-EAP thoughts
   - EAP authentication
   - Pwning the Master Key
   - Practical considerations

4. Conclusion
Authentication modes

- Preshared secret (PSK)
- EAP

Key hierarchy

- Authentication leads to Master Key (MK)
- Pairwise Master Key (PMK) derived from MK
One key to rule them all...

From MK come all further keys

- Pairwise Master Key
- Key exchange keys
- Encryption keys
- Authentication keys if applicable

Conclusion

Owning the Master Key $==$ Owning everything else
The Preshared Key option

- MK is your PSK
- PMK is derived from MK

PSK situation

Owning the PSK $\equiv$ Owning MK
The EAP option

- Authentication between client and RADIUS
- MK derived from authentication

- MK pushed to AP by RADIUS

EAP situation

Owning client + RADIUS == Ownning MK
Agenda

1. WPA/WPA2 authentication

2. WPA-PSK assessment
   - How does that work?
   - Theoretical attack cost
   - Implementation comparisons
   - Passphrase strength assessment
   - Limits of practical attacks

3. WPA-EAP thoughts
   - EAP authentication
   - Pwning the Master Key
   - Practical considerations

4. Conclusion
Calculating the PMK

The master key (MK)
- it is your secret key, password or passphrase
- 8 to 63 printable ASCII characters (between code 32 and 126)

The pairwise master key (PMK)
- derives from the master key and AP data using the PBKDF2 function
- the derivation function is time consuming
The attack

Retrieving the relevant data

- it must be captured during the handshake
- it is possible to force this handshake
- only works for a single SSID
The attack

Retrieving the relevant data

- it must be captured during the handshake
- it is possible to force this handshake
- only works for a single SSID

Testing a master key
The attack

Retrieving the relevant data
- it must be captured during the handshake
- it is possible to force this handshake
- only works for a single SSID

Testing a master key
1. for every potential MK, compute the corresponding PMK
The attack

Retrieving the relevant data
- it must be captured during the handshake
- it is possible to force this handshake
- only works for a single SSID

Testing a master key
1. for every potential MK, compute the corresponding PMK
2. compute the PTK (four HMAC-SHA1 calls using PMK and nonces)
The attack

Retrieving the relevant data

- it must be captured during the handshake
- it is possible to force this handshake
- only works for a single SSID

Testing a master key

1. for every potential MK, compute the corresponding PMK
2. compute the PTK (four HMAC-SHA1 calls using PMK and nonces)
3. finally, get the MIC (one HMAC-SHA1 call) and compare it with the captured handshake
The PBKDF2 function

**Algorithm of PBKDF2**

```plaintext
x1 = HMAC_SHA1(MK, SSID + '\1');
x2 = HMAC_SHA1(MK, SSID + '\2');
for(i=1;i<4096;i++) {
    x1 = HMAC_SHA1(MK, x1);
    x2 = HMAC_SHA1(MK, x2);
}
return x1 + x2;
```

**Cost**

- 8192 calls to HMAC-SHA1
The HMAC-SHA1 function

Algorithm for HMAC(secret, value)
put secret in two 64 bytes buffers, Bi and Bo, padding with zeroes

Bi
secret00000000000000000000000000000000
00000000000000000000000000000000

Bo
secret00000000000000000000000000000000
00000000000000000000000000000000
The HMAC-SHA1 function

Algorithm for HMAC(secret, value)

- XOR Bi with 0x36
- XOR Bo with 0x5c

Bi
ESUDSB6666666666666666666666666666666
666666666666666666666666666666666

Bo
/9?.9(__________________________/
_______________________________
_______________________________
_______________________________
_______________________________
_______________________________
_______________________________
_______________________________
_______________________________
The HMAC-SHA1 function

Algorithm for HMAC(secret, value)
append value to Bi

Bi
ESUDSB66666666666666666666666666666
666666666666666666666666666666666666666value
The HMAC-SHA1 function

Algorithm for HMAC(secret, value)

append SHA1(Bi) to Bo

Bo

/e7de45e0b885228e9a48a9add37b504eba7fd3c4/
The HMAC-SHA1 function

Algorithm for HMAC(secret, value)
get SHA1(Bo)

330b72d384df41adf440e1d8aeb543ab73eeeb8a

Summary
\[
HMAC\_SHA1(s, v) = SHA1((s \oplus 0x5c) || SHA1((s \oplus 0x36) || v))
\]
The SHA1 function

Description

- it is a cryptographic hash function
- works on 64 bytes blocks by padding user inputs
- produces a 20 bytes digest
- the main part of this function is called "BODY"
- other parts have an amortized cost of zero
The HMAC trick

Reminder
- we want SHA1(Bo || SHA1(Bi || value))

What will be computed
- BODY(secret ^ 0x5c)
- BODY(value + padding)
  ⇒ hash1
- BODY(secret ^ 0x36)
- BODY(hash1 + padding)
  ⇒ result
The HMAC trick

Reminder
- we want SHA1(Bo || SHA1(Bi || value))

What will be computed
- BODY(secret ^ 0x5c)
- BODY(value + padding)
  ⇒ hash1
- BODY(secret ^ 0x36)
- BODY(hash1 + padding)
  ⇒ result

if the secret is constant ...
### The HMAC trick

**Reminder**
- we want $\text{SHA1}(\text{Bo} \ || \ \text{SHA1(Bi} || \ \text{value}))$

**What will be computed**
- $\text{BODY}(\text{secret} \ ^{0x5c})$
- $\text{BODY}(\text{value} + \text{padding})$
- $\Rightarrow \text{hash1}$
- $\text{BODY}(\text{secret} \ ^{0x36})$
- $\text{BODY}(\text{hash1} + \text{padding})$
- $\Rightarrow \text{result}$

- if the secret is constant . . .
- . . . two BODY calls could be cached
The BODY function – initialization

Algorithm

```c
unsigned int K[80];
memcpy(K, input, 64);
a = ctx[0]; b = ctx[1]; c = ctx[2];
d = ctx[3]; e = ctx[4];
```

Operation count

- 32 bits memory assignments: 22
The BODY function – input expansion

Expand the input

\[
K[i] = K[i - 3] \hat{\lor} K[i - 8] \hat{\lor} K[i - 14] \hat{\lor} K[i - 16];
\]

\[
K[i] = \text{rotate}_\text{left}(K[i], 1);
\]

Operation count

- 32 bits memory assignment : 1
- elementary operations : 4
- done 64 times
The BODY function – rounds

**Algorithm**

STEP(v, w, x, y, z, m, c):

1. \( z \leftarrow F(w, x, y) + c + K[m] \);
2. \( z \leftarrow \text{rotate left}(v, 1) \);
3. \( w = \text{rotate left}(w, 30) \);

**Operation count**

- 32 bits memory assignments: 2
- Elementary operations: \( 5 + \text{cost of } F \)
- 4 rounds of 20 steps
- The average F cost is 3.75 operations
The BODY function – ending

Algorithm

\[
a \leftarrow \text{ctx}[0]; \quad b \leftarrow \text{ctx}[1]; \quad c \leftarrow \text{ctx}[2]; \\
d \leftarrow \text{ctx}[3]; \quad e \leftarrow \text{ctx}[4];
\]

Operation count

- 32 bits memory assignments : 5
- elementary operations : 5
The BODY function – summary

**Elementary operations count**

- initialization : 0
- input expansion : 4 times 64
- rounds : 8.75 times 80
- ending : 5
- total : 961

**Comparison with MD5**

- MD5 BODY function : 496
- if cracking a single MD5 : 317
The PBKDF2 function cost

**Elementary operations count**
- it requires 8192 HMAC-SHA1 calls using the same secrets
- that is, $2 + 8192 \times 2$ calls to SHA1
- that means 15.7M elementary operations
The PBKDF2 function theoretical speed

**Hypothesis :** perfect processors
- memory fetch/stores are free
- no penalties

**Speeds**
- for a perfect SSE2 implementation running at 3GHz on a single x86 core, about 500 checks/s
- for a perfect native CELL (PS3, 7 SPUs) implementation, about 2,840 checks/s
- for a perfect Linux CELL implementation, about 2,440 checks/s
## Real world implementations

### Aircrack
- 650 checks/s on Xeon E5405 (4x2Ghz)
- 650 checks/s on Opteron 2216 (4x2.4Ghz)
- "pipe multithreading", fails on AMD

### Pico Computing products
- on a LX25 FPGA, 430 checks/s
- on a FX60 FPGA, 1,000 checks/s

### Pyrit (GPU Project)
- around 6,000 checks/s on Tesla C870
Other cracking methods

WPA-PSK "rainbow tables"
- really PMK lookup tables
- precomputation of 1,000,000 passwords for 1000 SSIDs

Jason Crawford CELL implementation
- "Lockheed Breaks WPA-Encrypted Wireless Network With 8 Clustered Sony PlayStations"
- why did I bother, it is already broken :/
- unknown performance
Implementation on the cell architecture

**CELL benchmark**
- not a real cracker, just a bench
- under Linux, so only 6 SPUs are available
- pipeline filled by cracking 16 passwords at the same time

Result: 2,300 checks/s close to theoretical 2,400 checks/s expected on CELL
Implementation on the cell architecture

CELL benchmark
- not a real cracker, just a bench
- under Linux, so only 6 SPUs are available
- pipeline filled by cracking 16 passwords at the same time

Result
- 2,300 checks/s
- close to theoretical 2,400 checks/s
- expected on CELL
My implementation

NVidia CUDA cracker

- (almost) full fledged cracker, needs input from a modified aircrack-ng
- CUDA is easy: from no knowledge to this in 4 days
My implementation

**NVidia CUDA cracker**
- (almost) full fledged cracker, needs input from a modified aircrack-ng
- CUDA is easy: from no knowledge to this in 4 days

**Result**
- 4,400 checks/s on a 8800 gts
- 12,000 checks/s on a gtx280
- might not be too hard to do better
- roughly equivalent to Pyrit
The best bang for the buck

### Raw cost comparisons

<table>
<thead>
<tr>
<th>Type</th>
<th>checks/s</th>
<th>cost</th>
<th>checks/s/$</th>
</tr>
</thead>
<tbody>
<tr>
<td>LX25</td>
<td>430</td>
<td>385$</td>
<td>1.1</td>
</tr>
<tr>
<td>Q6600</td>
<td>800*</td>
<td>190$</td>
<td>4.2</td>
</tr>
<tr>
<td>Q9550</td>
<td>900*</td>
<td>325$</td>
<td>2.77</td>
</tr>
<tr>
<td>CELL</td>
<td>2300</td>
<td>400$</td>
<td>5.75</td>
</tr>
<tr>
<td>gtx280</td>
<td>12,000</td>
<td>440$</td>
<td>27.3</td>
</tr>
<tr>
<td>gtx260</td>
<td>9200*</td>
<td>300$</td>
<td>30.6</td>
</tr>
</tbody>
</table>

But . . .

- speeds marked with a * are not actual benchmarks, but interpolated results
- the CELL costs of 400$ is for a whole *PlayStation*
Password strength assessment function

A function $F$ gives the strength $s$ of password $p$: $F(p) = s$. 
Password strength assessment function

A function $F$ gives the strength $s$ of password $p$:
$$F(p) = s.$$  

Desirable properties

1. compute $F(p)$ effectively for any given $p$
2. for a given $s_{\text{max}}$, enumerate and generate all passwords \(\{p_0, p_1, \ldots p_n\}\) where \(F(p_i) < s_{\text{max}}, 1 \leq i \leq n\)
3. generate the set \(\{p_a, p_{a+1}, \ldots p_b\}\) where \(F(p_i) < s_{\text{max}}, a \leq i \leq b\) without generating \(\{p_0, \ldots p_{a-1}\}\)
4. assess the strength on a detailed scale
Well known methods

Dictionary checks

*it is weak if it is in a dictionary*

⇒ limited to ”known” passwords
Well known methods

**Dictionary checks**

*it is weak if it is in a dictionary*

⇒ limited to ”known” passwords

**Charset complexity**

*a strong password contains letters, numbers and at least three special characters*

⇒ Weak passwords could still be created
Well known methods

Dictionary checks

*it is weak if it is in a dictionary*
⇒ limited to ”known” passwords

Charset complexity

*a strong password contains letters, numbers and at least three special characters*
⇒ Weak passwords could still be created

Cracking tests

*it is weak if it is cracked in less than 4 hours with john on my computer*
⇒ requires computing ressources compatible with the risk analysis
A better method

## Markov chains

- The conditional probability distribution of letter $L_n$ in a password is a function of the previous letter, $L_{n-1}$, written $P(L_n|L_{n-1})$
- For example, $P(sun) = P(s) \cdot P(u|s) \cdot P(n|u)$
- To keep friendly numbers,
  $P'(x) = -10 \cdot \log(P(x))$
- $P'(sun) = P'(s) + P'(u|s) + P'(n|u)$
In practice

It works well

- has all the desired properties
- cracks more effectively than `john -inc` (in my tests!)
- a patch exists for `john`
In practice

It works well

- has all the desired properties
- cracks more effectively than `john -inc` (in my tests!)
- a patch exists for `john`

Sample strength

- "chall", strength 100
- "chando33", strength 200
- "chaneoH0", strength 300
- "chanlLr%", strength 400
- "chanereaAiO4", strength 500
- "% !", strength 1097
Hypothesis

### Attacker strength

<table>
<thead>
<tr>
<th>Attacker</th>
<th>Available time</th>
<th>Ressources (GPUs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wardriver</td>
<td>15 minutes</td>
<td>1</td>
</tr>
<tr>
<td>Individual</td>
<td>7 days</td>
<td>2</td>
</tr>
<tr>
<td>Large organisation</td>
<td>1 year</td>
<td>1024</td>
</tr>
</tbody>
</table>

### Defender strength

- worst case scenario: mac user :)
- password is 12 characters or less
Not so good passwords

Statistics source
- an Apple themed forum that got owned
- clear text passwords published on 4chan

Passwords strength
- 628,753 passwords
- mean strength: 245
- median strength: 197
- most common passwords “base”: password, qwerty, apple, letmein
Strength of crackable passwords

Now
- wardriver: 7.2M, markov strength of 169
- individual: 14.5G, markov strength of 239
- large organisation: 387.8T, markov strength of 344

In 10 years, 32 times faster (Moore)
- wardriver: 345.6M, markov strength of 202
- individual: 464.5G, markov strength of 273
- large organisation: 12409T, markov strength of 388
Discovery ratio vs. computing power

![Graph showing the relationship between discovery ratio and cracking capacity.](image-url)
Discovery ratio vs. computing power

- Wardriver
- Individual
- Organization
Discovery ratio vs. computing power

- **Wardriver**
- **Individual**
- **Organization**
- 10 years projection
Agenda

1. WPA/WPA2 authentication
2. WPA-PSK assessment
   - How does that work?
   - Theoretical attack cost
   - Implementation comparisons
   - Passphrase strength assessment
   - Limits of practical attacks
3. WPA-EAP thoughts
   - EAP authentication
   - Pwning the Master Key
   - Practical considerations
4. Conclusion
Usual issues

EAP strength directly linked to good configuration
- Good choice in EAP method
- Proper RADIUS authentication

In particular...
Strictly verify RADIUS certificate to avoid MiM
Looking more carefully

AP acts as a relay between client and RADIUS server
Looking more carefully

AP acts as a relay between client and RADIUS server

Direct EAP communication between client and RADIUS
What if...

There was an exploitable flaw within EAP?

- Ability to execute arbitrary code
- Access to RADIUS database
- Access to backend
- Etc.

More importantly

Ability to generate RADIUS traffic!
Of MK transmission

AP notification
When authentication done, RADIUS notifies AP
- EAP Success (3) or Failure (4)
- MK sent using MS-MPPE-Recv-Key (attribute 17)
- HMAC-MD5 message (attribute 80)
Injecting arbitrary MK

- Have your shellcode executed
- Craft a EAP Success
- Put your own MK in MS-MPPE-Recv-Key
- Have it sent to AP

Small issue...
You need to compute HMAC-MD5 message
Bypassing HMAC-MD5

- You don’t know RADIUS secret
- But you own the server...

Ideas
- Read secret from conf/memory
- Ask RADIUS to craft packet for you

Product dependant methods
In practice

Some stuff done or to do

- EAP fuzzing (flaws)
- EAP fingerprinting (id)
- Exploits

Then...
Attacker can have his own MK sent back to AP
Not quite the end of it...

Still need to perform 4-Way Handshake
- Hack a WPA/WPA2 supplicant!
- Specific module for wpa_supplicant

**Step by step**
- Answer EAP Request from AP
- Start EAP dialog to RADIUS
- Trigger the vulnerability
- Deliver exploit
- Grab EAP Success from AP
When you’re done...

In the end...

- Rogue client starts authenticating
- Exploits RADIUS server
- Gets authenticated with arbitrary MK
- Finish WPA/WPA2 dialog with AP
When you’re done...

**In the end...**
- Rogue client starts authenticating
- Exploits RADIUS server
- Gets authenticated with arbitrary MK
- Finish WPA/WPA2 dialog with AP

**Most importantly...**
He can now access the network through Wi-Fi
Agenda

1. WPA/WPA2 authentication
2. WPA-PSK assessment
   - How does that work?
   - Theoritical attack cost
   - Implementation comparisons
   - Passphrase strength assessment
   - Limits of practical attacks
3. WPA-EAP thoughts
   - EAP authentication
   - Pwning the Master Key
   - Practical considerations
4. Conclusion
PSK selection

Recommandations

- if possible, just use a random 64 bytes value, or one of the safer authentication schemes
- passwords not derived from a known word and with a strength of 400 or more on the Markov scale should be safe for the next years
- just use "chanereaAiO4", it is safe!
PSK selection

Recommandations

- if possible, just use a random 64 bytes value, or one of the safer authentication schemes
- passwords not derived from a known word and with a strength of 400 or more on the Markov scale should be safe for the next years
- just use "chanereaAiO4", it is safe!

Beware

- the cracker might have a better model for his attacks
- "real" sentences might seem safe because they are long, but are likely to be weak
- crypto flaws might be discovered and exploited
## The future of PSK

### Automatic key setup
- Several proprietary solutions, and a standard
- Automagically sets the network and security settings
- Removes user input, no more bad keys (hopefully)

### Wi-Fi Protected Setup
- Standard from the Wi-Fi Alliance
- Authenticates the device by
  - In-band: entering a PIN code, pushing a button
  - Out-of-band: connecting an USB stick, reading RFIDs
- Might be attacked during the first association
EAP considerations

Recommandations

- Carefully choose your EAP method
- Ensure clients can authenticate RADIUS
- Harden your RADIUS box
- Proxy authentication to another AAA server
EAP considerations

Recommendations
- Carefully choose your EAP method
- Ensure clients can authenticate RADIUS
- Harden your RADIUS box
- Proxy authentication to another AAA server

Beware
- RADIUS certificate must checked, always
- Against your very own CA, only
The end...

Thank you all for your attention

Questions?