Depletion Region

Monday, October 7, 2013

CC3000 Smart Config - transmitting SSID and keyphrase

13 comments 2:54 PM

Initially TI clearly documented how the SSID and password were transmitted to a CC3000 enabled device in their "CC3000 First Time Configuration" document. However with release 1.10 they changed the approach to one called Smart Config and now document the API but no longer explain what is happening at the network level. Here I cover this missing information for the new approach.

So let's start at the start - we have a problem - we want to send two pieces of information, an SSID and the keyphrase, from one party that is already a member of the wifi network to an external party who can monitor all the encrypted wifi traffic but who cannot decrypt it.

Someone who cannot decrypt the wifi traffic can still see quite a lot of information, e.g. they can see the source and receiver MAC addresses of every packet sent.

They can also see the length of the data portion of the packets. The encryption affects that size of the packets sent but in a consistent manner, e.g. if one sends *n* bytes of data in a given packet then the encrypted packet will contain *(n + x)* bytes where *x* is constant across all packets.

So the solution to our problem is to encode the information in the size of the packets sent (the actual content is irrelevant).

The party on the secured network just sends UDP packets with particular lengths to another party on the network. That the other party is not interested in receiving the packets is not important.

The external party cannot tell directly that a packet that it is looking at contains UDP data, however the packets still include basic type information that allows many packets to be excluded from consideration, e.g. any packet that is not of 802.11 subtype "QoS data" can be excluded.

As the external party does not know in advance which wifi channel to look at or which source and receiver address pair to pay attention to one must, in addition to the underlying data, i.e. encoded SSID etc., send regular repeating patterns that allow this data to be spotted.

We convert our SSID and keyphrase into a sequence of tag values, string lengths, nibble values and separators values and then encode and transmit all these values as packet lengths.

Let's look in detail at the values sent.

We use two tags - an SSID tag with value 1399 and a keyphrase tag with value 1459 and one standard separator sequence consisting of two values - 3 followed by 23.

And we use two constants, L with value 28 and C with value 593, that we will see used below.

So for the SSID the following sequence of values are generated in this order:

- The SSID tag 1399.
- L plus the length of the SSID in bytes.
- The two separator values 3 and 23.
- Then we loop over each byte of the SSID and generate a set of four values for each:
  Two values - one for each nibble of the byte, as described in the next section.
  Followed by the two separator values 3 and 23.

Values are generated in an identical fashion for the keyphrase (except that the keyphrase tag 1459 is used in place of the SSID tag).

Note: the TI Android library and Java applet library generate values as described above, oddly the TI

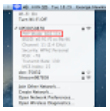
Search the site Search!

Popular Posts

CC3000 Smart Config - transmitting SSID and keyphrase
Initially TI clearly documented how the SSID and password were transmitted to a CC3000 enabled device in their "CC3000 First Time Confi...

The CC3000, 802.11n and MIMO
MIMO creates a significant issue for the CC3000. Most modern wifi access points (APs) support at least the following three 802.11 protoco...

Smart Config clients for iOS, Android, Mac OS X, Windows and Linux
TI supplies a number of Smart Config reference implementation clients for developers here: http://www.ti.com/tool/smartconfig However th...



Determining the 802.11 protocol (b, g or n) being used and forcing the use of b or g
Most wireless access points (APs) operate in what is known as mixed mode and can support clients that use 802.11 protocols b, g or n. Th...

CC3000 Smart Config and AES
One can use AES with the CC3000 and Smart Config - this will prevent arbitrary third parties who do not know the relevant AES key from reco...

CC3000 Smart Config and keyphrase recovery
Having previously described how the SSID and keyphrase are transmitted to a CC3000 enabled device I thought I should put my money where my ...

CC3000 advertises presence on network via DNS-SD
Once you press "Start" in a TI Smart Config setup application the application will repeatedly

iOS library produces a slightly different ordering (which clearly doesn't affect the CC3000's ability to decode the data). This difference can be seen in the example data length dumps shown latter.

Once we have all these values then UDP packets, each with an amount of data corresponding to one of these values, are sent from the machine running the Smart Config application, i.e. the one that has generated the values just described, to another system on the same network (currently always the network's default gateway).

The values are sent repeatedly until the external party, i.e. the CC3000 enabled device, successfully sifts them out from all the other network traffic and uses them to connect to the network, at which point it advertises its presence on the network in a manner that the transmitting application can detect and which causes it to stop transmitting.

Note that the range of packet lengths that need to be supported places a lower bound on the maximum transmission unit (MTU) for the network. Currently the Smart Config client application expects the MTU to be 1500 or greater (this is a reasonable expectation on any normal network).

The TI Smart Config reference implementation resends the full set of UDP packets corresponding to the SSID and keyphrase repeatedly. The TI Java applet library pauses 100ms after each complete transmission of the full set of values, the Android and iOS libraries do not bother pausing.

Encoding the characters of the SSID or keyphrase.

If an SSID consists of *n* characters *0* to *n - 1* then we generate *2n* corresponding values.

Note: according to IEEE standard 802.11i-2004, Annex H.4.1, users may enter keys as a string of 64 hexadecimal digits (or alternatively as a passphrase of printable ASCII characters). Presumably WEP and WPA specify similar restrictions. The SSID must be a sequence of between 1 and 32 bytes, there is no mandated character set (more details in this StackOverflow answer) and how the SSID is displayed is left up to the end user application (however many routers apparently only accept printable ASCII characters for the SSID).

So if we assume ASCII characters encoded as 8 bit values then each value consists of a high and low nibble.

E.g. 'M' in ASCII is hex 0x4D, the high nibble is 0x4 and the lower nibble is 0xD.

If we maintain a sequence number starting from 0 and increment it each time we generate a value then for character *i* of the SSID, consisting of a high and low nibble H_i and L_i, we generate two values with sequence number *2i* and *(2i + 1)* respectively. Each of these values has a high and low nibble calculated as follows:

Seq.	High	Low
2i	L_{i-1} ^ (2i % 16)	H_i
2i+1	H_i ^ ((2i + 1) % 16)	L_i

Note that the value containing the high nibble of *i* is generated before the one containing the low nibble of i. And note that caret, i.e. '^', is used here to mean *XOR*, rather than *power of*.

The following shows how the SSID "MyPlace" would be split up into high and low nibbles:

	'M'		'y'		'P'		'l'		'a'		'c'		'e'	
Hex:	0x4D		0x79		0x50		0x6C		0x61		0x63		0x65	
Nibbles:	0x4	0xD	0x7	0x9	0x5	0x0	0x6	0xC	0x6	0x1	0x6	0x3	0x6	0x5
	H_0	L_0	H_1	L_1	H_2	L_2	H_3	L_3	H_4	L_4	H_5	L_5	H_6	L_6

For each 4 bit nibble we generate a value whose lower 4 bits consist of the nibble itself and whose higher 4 bits consist of the current sequence number XORed with the value of the previously used nibble value. We then add the constant C mentioned above, i.e. 593, to each value generated in this way and this becomes the length of the packet that encodes such a value.

broadcast the SSID etc. until the re...

Java Smart Config desktop client

I've written a desktop Smart Config user interface in Java for setting up CC3000 enabled devices. It's available under the Apac...

CC3000 and security

People talk about a coming Internet of Things (IoT), pervasive small cheap headless devices that connect to the internet and do things li...

Smart Config for consumer products? Alternatives?

Smart Config looks like magic the first time you see it - what it's doing seems impossible if you know a bit about encrypted wifi networ...

About Me

GEORGE HAWKINS

Follow 34

VIEW MY COMPLETE PROFILE

Powered by Blogger.

Blog Archive

Note that the 4 bit constraint means that we only use the lower 4 bits of the current sequence number, i.e. if the sequence number *S* is above 15 then we use *S % 16*.

This results in the generation of 14 values for the 7 character of the SSID name "MyPlace" like so:

Char	Seq	→	Hi	Lo	→	Byte	Hi	Lo	→	Hi	Lo	→	Sum	→	Len
'M'	0		0x0	H_0		0x4D	0x0	0x4		0x0	0x4		0x04 + 593		597
	1		H_0 ^ 0x1	L_0			0x4 ^ 0x1	0xD		0x5	0xD		0x5D + 593		686
'y'	2		L_0 ^ 0x2	H_1		0x79	0xD ^ 0x2	0x7		0xF	0x7		0xF7 + 593		840
	3		H_1 ^ 0x3	L_1			0x7 ^ 0x3	0x9		0x4	0x9		0x49 + 593		666
'P'	4		L_1 ^ 0x4	H_2		0x50	0x9 ^ 0x4	0x5		0xD	0x5		0xD5 + 593		806
	5		H_2 ^ 0x5	L_2			0x5 ^ 0x5	0x0		0x0	0x0		0x00 + 593		593
'l'	6		L_2 ^ 0x6	H_3		0x6C	0x0 ^ 0x6	0x6		0x6	0x6		0x66 + 593		695
	7		H_3 ^ 0x7	L_3			0x6 ^ 0x7	0xC		0x1	0xC		0x1C + 593		621
'a'	8		L_3 ^ 0x8	H_4		0x61	0xC ^ 0x8	0x6		0x4	0x6		0x46 + 593		663
	9		H_4 ^ 0x9	L_4			0x6 ^ 0x9	0x1		0xF	0x1		0xF1 + 593		834
'c'	10		L_4 ^ 0xA	H_5		0x63	0x1 ^ 0xA	0x6		0xB	0x6		0xB6 + 593		775
	11		H_5 ^ 0xB	L_5			0x6 ^ 0xB	0x3		0xD	0x3		0xD3 + 593		804
'e'	12		L_5 ^ 0xC	H_6		0x65	0x3 ^ 0xC	0x6		0xF	0x6		0xF6 + 593		839
	13		H_6 ^ 0xD	L_6			0x6 ^ 0xD	0x5		0xB	0x5		0xB5 + 593		774

The keyphrase is encoded in the same way, note that the sequence number starts again from 0 when encoding the keyphrase, i.e. the value is not carried over from encoding the SSID.

Currently Smart Config enforces an upper limit of 32 characters on the keyphrase length, i.e. shorter than the maximum length allowed by the relevant WPA2 standard.

I find the approach used to actively leak information from a secure wireless network to an external party (that does not have the relevant network keyphrase) interesting and would like to hear if anyone has come across it before or whether it is novel? I asked about this on Stack Overflow but have since moved the question to the sister site crypto.stackexchange.com after people suggested it was more appropriate there.

Update Oct 21, 2013: I've now got at least one good answer. In a paper from 2007 by P. Martin called "Covert channels in secure wireless networks" you can find section 4.4.2 "UDP Packet Size vs MAC Frame Size Experiment" that essentially describes exactly the process used by Smart Config. The question of patents has come up once or twice in relation to Smart Config (though no one has ever provided pointers to any actual patent applications or granted patent numbers). The answers and other comments on my question would seem to suggest that there's definitely prior art for the fundamental idea behind Smart Config.

Choosing the destination for the UDP packets

The current logic always sends the UDP messages, that encode the SSID etc., to the default gateway address. However it doesn't actually matter what address they're sent to as long as it's the address of another machine on the network that actually exists and is capable of receiving packets. However it makes sense to have decided on a definite address.

The CC3000 doesn't support ad hoc networks so you are never going to get a situation where it is

used on a network that doesn't have an access point (AP), and in most normal setups the AP is also the default gateway. However I don't think a default gateway is mandatory (someone can correct me on this?), one can imagine a self contained wifi network where the AP isn't a gateway to anywhere else.

I wonder why TI didn't choose on the address of the AP rather than the default gateway. Even if the two are almost always the same I think an end user is probably likely to have much more luck, if needed, asking Google or their ISP first level technical support how they find the address of their wifi AP than asking about default gateways.

Note: in a network using an AP all traffic goes via the AP, so even if one chose the address of a machine other than the AP the traffic would still be received by the AP and retransmitted by the AP to the destination machine. This doesn't cause an issue if you try it with the CC3000 but it does mean the traffic in pointlessly duplicated.

Further details of the Smart Config library

The sections above cover the heart of how Smart Config works, the following covers details of TI's Smart Config Java applet library that don't immediately seem relevant - they may be left over from the development process and have just never been cleaned out or they may relate to non-default functionality that it's possible to use if a particular CC3000 device is configured in a particular way.

- One can set the DatagramSocket used to send the UDP packets to something other than one simply setup to bind to any available port on the local machine.
- One can set the port to which the UDP packets are sent to something other than 15000.
- One can set the port on which mDNS UDP messages are expected to something other than 5353.
- One can set can the timeout, i.e. the time the application waits, for the CC3000 enabled device to connect to something other than 5 minutes.
- One can set the number of times all the details are retransmitted before taking a 100ms pause. By default this is 1, i.e. the logic takes a 100ms pause between every full transmission of the details.
- One can set the two separators values, i.e. 3 and 23 mentioned above, to different values and more bizarrely set the characters used to make up the packets with these lengths.

These features are available in the library but are never made use of by the TI Smart Config application built on top of this library.

One can also set the network interface of the socket used to listen for mDNS message. However this seems completely pointless as this just affects *outgoing* multicast datagrams and the relevant logic is only looking for *incoming* datagrams.

With one exception all of this configurability looks pointless.

- It's hard to think of why one might want to specify the bind address of the DatagramSocket used to send the UDP packets.
- The machine receiving the packets will ignore them and so choosing a particular port seems irrelevant and with encrypted network traffic one cannot see port information so it shouldn't be relevant to the CC3000 device's ability to detect the relevant traffic.
- mDNS always uses port 5353 and given how mDNS is used it's hard to imagine that port number being changed on a particular network.
- The timeout is fairly arbitrary, 5 minutes seems extremely long so it does seem reasonable to adjust it down if one really cares about this.
- Being able to set the number of full retransmits before a pause has no obvious benefits.

That leaves us with being able to change the separator lengths. Perhaps this is configurable on CC3000 devices. I can't see a reason why you'd want to change the default values. And if one did I suspect one would have to be careful what values one chose. The tag values for SSID and keyphrase, and the constants C and L mentioned above, along with the two separator length values, all seem to have been chosen to ensure no overlap in values between one kind of thing and another, the current setup means no packet length encoding a nibble of an SSID character or keyphrase will end up with a length equal to the tags or separator values.

Note: the port number 15000 mentioned above has not been registered by TI and actually belongs to a legitimate service called "hydap" which has been registered by HYPACK Inc. with IANA. This shouldn't be an issue for anyone.

Multicast mDNS

Above you saw some mention of mDNS. This is involved in detecting that a CC3000 device has successfully connected to the network and is discussed in more detail in this post.

What character set does the CC3000 use?

The Smart Config Java applet library stores the SSID etc. as standard Java strings, i.e. sequences of Unicode characters. It converts back and forward between these strings and arrays of bytes at various points, but takes no care of character sets, e.g. when it calls String.getBytes() it always uses the form that uses the platform's default character set. This will work on most systems, including pretty much everything in the US and Western Europe, but will obviously be an issue elsewhere. The CC3000 presumably has a fixed character set and this should be used explicitly in the library when converting between characters and bytes.

Update: experimentation shows that while the TI Java applet library just uses the default character set of the machine it's running on, when converting to and from bytes, the TI iOS and Android apps both seem to consistently use UTF-8.

Packet length dumps from the TI Smart Config applications

If any of the above wasn't too clear, hopefully this section will help with a practical example of the packet lengths generated by the TI Smart Config applications when sending the SSID "MyPlace" and the password "LetMeIn".

The first dump shows the packet lengths generated by the Android and Java applet applications. The first colum just shows the UNIX time the packet was sent at, the second column shows the length of the packet and this is then followed by an indication of what the length of packet is encoding.

```
1381084544.032552000    1399 <----- SSID tag
1381084544.033572000    35   <----- SSID length + 28
1381084544.033589000    3    <--+-- separator
1381084544.033594000    23   <--'
1381084544.033667000    597  <----- 'M' hi-nibble
1381084544.033675000    3    <--+-- separator
1381084544.033723000    23   <--'
1381084544.034369000    686  <----- 'M' lo-nibble
1381084544.035385000    3    <--+-- separator
1381084544.036271000    23   <--'
1381084544.036448000    840  <----- 'y' hi-nibble
1381084544.036467000    3    <--+-- separator
1381084544.036481000    23   <--'
1381084544.036541000    666  <----- 'y' lo-nibble
1381084544.037262000    3    <--+-- separator
1381084544.037271000    23   <--'
1381084544.037496000    806  <----- 'P' hi-nibble
1381084544.038019000    3    <--+-- separator
1381084544.038032000    23   <--'
1381084544.038097000    593  <----- 'P' lo-nibble
1381084544.043096000    3    <--+-- separator
1381084544.044209000    23   <--'
1381084544.044785000    695  <----- 'l' hi-nibble
1381084544.045422000    3    <--+-- separator
1381084544.045855000    23   <--'
1381084544.048359000    621  <----- 'l' lo-nibble
1381084544.049327000    3    <--+-- separator
1381084544.049347000    23   <--'
1381084544.049406000    663  <----- 'a' hi-nibble
1381084544.049412000    3    <--+-- separator
1381084544.049416000    23   <--'
1381084544.049568000    834  <----- 'a' lo-nibble
1381084544.050052000    3    <--+-- separator
1381084544.050067000    23   <--'
1381084544.050808000    775  <----- 'c' hi-nibble
1381084544.051463000    3    <--+-- separator
1381084544.052082000    23   <--'
1381084544.055415000    804  <----- 'c' lo-nibble
1381084544.056319000    3    <--+-- separator
1381084544.056334000    23   <--'
1381084544.056398000    839  <----- 'e' hi-nibble
1381084544.056404000    3    <--+-- separator
1381084544.056407000    23   <--'
1381084544.056644000    774  <----- 'e' lo-nibble
1381084544.058021000    3    <--+-- separator
```

```
1381084544.058034000    23   <--'
1381084544.059236000    1459 <----- passphrase tag
1381084544.059252000    35   <----- passphrase length + 28
1381084544.059255000    3    <--+-- separator
1381084544.059258000    23   <--'
1381084544.059261000    597  <----- 'L' hi-nibble
1381084544.059937000    3    <--+-- separator
1381084544.059949000    23   <--'
1381084544.060043000    685  <----- 'L' lo-nibble
1381084544.060723000    3    <--+-- separator
1381084544.060729000    23   <--'
1381084544.060884000    823  <----- 'e' hi-nibble
1381084544.061407000    3    <--+-- separator
1381084544.061411000    23   <--'
1381084544.061954000    678  <----- 'e' lo-nibble
1381084544.062651000    3    <--+-- separator
1381084544.062709000    23   <--'
1381084544.063217000    616  <----- 't' hi-nibble
1381084544.063696000    3    <--+-- separator
1381084544.063699000    23   <--'
1381084544.064344000    629  <----- 't' lo-nibble
1381084544.064893000    3    <--+-- separator
1381084544.064897000    23   <--'
1381084544.065561000    629  <----- 'M' hi-nibble
1381084544.066131000    3    <--+-- separator
1381084544.066221000    23   <--'
1381084544.066947000    654  <----- 'M' lo-nibble
1381084544.066955000    3    <--+-- separator
1381084544.067371000    23   <--'
1381084544.067491000    679  <----- 'e' hi-nibble
1381084544.067871000    3    <--+-- separator
1381084544.068325000    23   <--'
1381084544.069089000    838  <----- 'e' lo-nibble
1381084544.069097000    3    <--+-- separator
1381084544.069593000    23   <--'
1381084544.069711000    837  <----- 'I' hi-nibble
1381084544.070191000    3    <--+-- separator
1381084544.070656000    23   <--'
1381084544.074244000    842  <----- 'I' lo-nibble
1381084544.074259000    3    <--+-- separator
1381084544.075225000    23   <--'
1381084544.075286000    679  <----- 'n' hi-nibble
1381084544.075291000    3    <--+-- separator
1381084544.075293000    23   <--'
1381084544.075521000    783  <----- 'n' lo-nibble
1381084544.075533000    3    <--+-- separator
1381084544.076058000    23   <--'
-------------------------   No delay on Android, 100ms delay with Java applet library, then repeat from start again.
1381084544.076246000    1399 <----- SSID tag
1381084544.076850000    35   <----- SSID length + 28
...
```

The output generated by the TI iOS and Java applet Smart Config applications is identical (except for the noted 100ms delay). However oddly the iOS Smart Config application does not interleave the separator values 3 and 23 between the characters of the SSID and keyphrase, instead it always sends out a sequence of 10 separator value pairs as shown here and then sends out the SSID and keyphrase. So here one can hardly call 3 and 23 separators but I've stuck with this name here:

```
1381085051.154799000    3    <--+-- separator 1
1381085051.159414000    23   <--'
1381085051.164143000    3    <--+-- separator 2
1381085051.170050000    23   <--'
1381085051.174861000    3    <--+-- separator 3
1381085051.179503000    23   <--'
1381085051.185282000    3    <--+-- separator 4
1381085051.190274000    23   <--'
1381085051.195296000    3    <--+-- separator 5
1381085051.200047000    23   <--'
1381085051.206394000    3    <--+-- separator 6
1381085051.211076000    23   <--'
1381085051.215383000    3    <--+-- separator 7
1381085051.225363500    23   <--'
1381085051.235344000    3    <--+-- separator 8
1381085051.235459000    23   <--'
1381085051.236902000    3    <--+-- separator 9
```

```
1381085051.241718000    23   <--'
1381085051.249366000    3    <--+-- separator 10
1381085051.253099000    23   <--'
1381085051.257767000    1399 <----- SSID tag
1381085051.262315500    35   <----- SSID length + 28
1381085051.266864000    597  <----- 'M' hi-nibble
1381085051.273117000    686  <----- 'M' lo-nibble
1381085051.278023500    840  <----- 'y' hi-nibble
1381085051.282930000    666  <----- 'y' lo-nibble
1381085051.291178000    806  <----- 'P' hi-nibble
1381085051.294688000    593  <----- 'P' lo-nibble
1381085051.299266000    695  <----- 'l' hi-nibble
1381085051.308603000    621  <----- 'l' lo-nibble
1381085051.311723000    663  <----- 'a' hi-nibble
1381085051.315706000    834  <----- 'a' lo-nibble
1381085051.321567000    775  <----- 'c' hi-nibble
1381085051.326156000    804  <----- 'c' lo-nibble
1381085051.332654000    839  <----- 'e' hi-nibble
1381085051.337025000    774  <----- 'e' lo-nibble
1381085051.342818000    1459 <----- passphrase tag
1381085051.346519000    35   <----- passphrase length + 28
1381085051.353083000    597  <----- 'L' hi-nibble
1381085051.359196000    685  <----- 'L' lo-nibble
1381085051.362984000    823  <----- 'e' hi-nibble
1381085051.366772000    678  <----- 'e' lo-nibble
1381085051.373192000    616  <----- 't' hi-nibble
1381085051.382117000    629  <----- 't' lo-nibble
1381085051.386131000    629  <----- 'M' hi-nibble
1381085051.390145000    654  <----- 'M' lo-nibble
1381085051.393997000    679  <----- 'e' hi-nibble
1381085051.400047000    838  <----- 'e' lo-nibble
1381085051.404880000    837  <----- 'I' hi-nibble
1381085051.412003000    842  <----- 'I' lo-nibble
1381085051.414365000    679  <----- 'n' hi-nibble
1381085051.420336000    783  <----- 'n' lo-nibble
--------------------------  No delay then repeat from start again.
1381085051.432048500    3    <--+-- separator 1
1381085051.443761000    23   <--'
...
```

Dumping and decrypting wifi packets using tshark

The above dumps were produced with the command line version of the well known packet analyser tool Wireshark that's called tshark. Wireshark is available for Mac OS X, Windows and Linux. The following shows how I used it to produce these dumps.

First I started it off recording packets before doing anything with the particular TI Smart Config application I was working with like so:

```
$ tshark -i en0 -I -w output.pcap
```

Note that the `-I` flag puts your machine's wifi network interface into monitor mode, this disables standard network access for everything else on the machine while tshark is running but allows tshark to see all the traffic on the network, not just traffic intended for the machine that it's running on. On my Mac putting the network interface into monitor mode works fine but on other platforms this can be far more complicated - I'll discuss this further in a later post.

I then ran the particular Smart Config application on a *different* machine, i.e. an iPhone, an Android phone or a different desktop machine. I hit the start button that you find in each Smart Config application that tells it to start transmitting the SSID etc., and let it run for a while. There's no requirement to have an actual CC3000 enabled device listening for the data. After a few seconds I killed the tshark process with control-C and hit stop in the Smart Config application.

The resulting network traffic ends up in the file output.pcap, note the size of this file can grow very quickly as we're recording all traffic on a given wifi channel, not just the Smart Config related traffic.

Then we can filter out the Smart Config traffic and dump it out as above like so:

```
$ tshark -r output.pcap -o 'wlan.enable_decryption:TRUE' \
    -Y 'wlan.fc.retry == 0 && !icmp && udp && ip.src == 192.168.1.177 && ip.dst == 192.168.1.1 && udp.dstport == 15000'
    -T fields -e frame.time_epoch -e data.len | head -n 512
```

ip.src *must* to be changed to the IP address of the device running the Smart Config application, e.g.

an iPhone, and ip.dst must be the IP address used as the gateway by the application.

Note that I exclude icmp packets, these are control packets that can contain embedded UDP packets that would be matched by the udp filter. In our case icmp packets are generated to tell us that the port we are sending the packets to is unreachable. With the wlan.fc.retry filter I also exclude retransmitted packets - I'll discuss this more in a later post.

As shown here tshark can actually decrypt the wifi packets (something which a CC3000 enabled device that's waiting for the network password obviously cannot do). Having done so it can then filter the traffic using higher level network terms like the IP protocol, in this case UDP, and IP addresses.

If the option `wlan.enable_decryption` is set to `TRUE`, as shown above, tshark will search for the keyphrase necessary to do the decryption in the file `~/.wireshark/80211_keys` where it will expect to find something like:

```
"wpa-pwd","LetMeIn:MyPlace"
```

Note that the keyphrase comes before the SSID and that storing keys (in plain text, yes) in the file `80211_keys` is relatively new to wireshark/tshark. For earlier versions of tshark you may have to specify the key information as a command line option - Google for what works for your version.

However even if it has the keyphrase it can only decrypt the traffic if your pcap file contains the EAPOL packets that are generated when the device designated by ip.src goes through the initial negotiation of a secure connection with the AP. To force my devices to generate such packets I flipped the phones in and out of airplane mode and on my desktops I temporarily disabled and then reenabled wifi. This didn't *always* seem to work - I don't know if even on being disabled and reenabled a device can sometimes avoid having to renegotiate its connection. One can check if the pcap file contains EAPOL packets like so:

```
$ tshark -r sc-ios.pcap -Y eapol
```

You should see about four packets which correspond to your device (check that the displayed MAC addresses match the MAC address of the device).

Obviously the CC3000 cannot decrypt the packets like this - I'll discuss in a later post how it does much the same as we've done here but without having the relevant decryption information.

Google . Like 7

Related Posts

The CC3000, 802.11n and MIMO


CC3000 Smart Config and AES

Smart Config for consumer products?...

Ideas for improving Smart Config


CC3000 and security

NEWER POST OLDER POST

COMMENT!

13 COMMENTS:

DBM123 October 12, 2013 at 7:22 PM

+1

Reply

NABIL KHALIL December 9, 2013 at 2:21 PM

Wow! What an amazing piece of work. Thank you so much for going so deep into this topic. Your blog is invaluable!

Reply

FRANCOIS GERVAIS April 7, 2014 at 4:57 PM

Wow thanks a lot!

Reply

AKASH NIGAM July 7, 2014 at 7:16 PM

That helped a lot in my understanding.
Thanks.

Reply

CHETHAN September 19, 2014 at 1:46 PM

This comment has been removed by the author.

Reply

CHETHAN September 19, 2014 at 1:48 PM

Hi really very usefull information .. thanks for the effort.
****
However it doesn't actually matter what address they're sent to as long as it's the address of another machine on the network that actually exists and is capable of receiving packets.
The CC3000 doesn't support ad hoc networks so you are never going to get a situation where it is used on a network that doesn't have an access point (AP), and in most normal setups the AP is also the default gateway.
****

After reading the above two paragraph from your bolg , I got the below doubts.
Take a case during very fresh set up, the CC3x00 may not connected to any of the AP and its not in the network .
Then Where to send the data(Destination ID) from smartConfig app?
How would CC3x00 devices receives it ? .. Pls help me in understanding this ..

Reply

Replies

GEORGE HAWKINS September 19, 2014 at 2:41 PM

The smart config setup app does not run on the CC3000 - it runs on e.g. your Android phone. This device must already be connected to your wifi network and therefore knows the address of the AP - the destination for the UDP packets it will send. The CC3000 device is not connected but it can see the encrypted traffic of *all* nearby APs (it's just radio traffic - everyone can see it but only people with the right keys can decrypt any of it, assuming encryption is being used). The trick the CC3000 performs is to zero in on a particular subset of all the wifi traffic around it - by looking for repeating patterns of packets of particular lengths - and then extracts the key information encoded in the lengths of the packets interleaved with these

"advertising" packets (as described above). One can see the length of any packet even if one can't yet decrypt it.

YERVANT December 7, 2014 at 8:00 PM

Very helpful post! Thanks.

What about the security breach that smart config brings? Anyone who monitors this setup process would decode the wifi key.

Regarding 'Choosing the destination for the UDP packets', you wonder why smart config didn't choose the address of the AP. Well, I think the point is:

1) Smart Config may not be aware of the address of the AP if the AP is not the gateway. Imagine an enterprise wired network which requires a wireless 'extension', an AP is simply plugged in as a repeater/bridge for wireless devices. In this case, the AP MAY (I'm not sure if it's a MUST) have an IP address in the same subnet, but it's not necessary for the wireless stations to know it;
2) I think TI chooses the gateway as the target may be just for the sake of programming Smart Config based on the assumption that the network needs a way out to another network (e.g.: the Internet). In most cases, the gateway is always a reachable target. Thus it's OK to send out UDP packets by basic socket programming. To allow a different target, the end user must manually enter it, which means the end user should be familiar with their networks. If by chances the target is not reachable, UDP packets will not be present on the fly as ARP would fail. To resolve this case, programmers have to use raw socket which would just make things more complicated.

Reply

Replies

GEORGE HAWKINS December 8, 2014 at 2:46 PM

Thanks Yervant for your comments on not using the AP address. As to the security breach - yes I think it's terrible and in a later post "keyphrase recovery" I show how easy this is :(

MARK December 8, 2014 at 10:26 AM

Awesome work here. It looks like SmartConfig has changed for current generation CC3100 and CC3200. Have you attempted any similar analysis for those platforms?

Reply

Replies

GEORGE HAWKINS December 8, 2014 at 2:59 PM

Hi Mark - to be honest I've kind of given up on the CC3000 - I think Smart Config is a neat trick but not something for consumer products (see my later post on this), in addition to this the CC3000 had no end of firmware issues. I have a CC3200 board and think it's a vast improvement but nothing is going to make Smart Config more than a clever trick. I hadn't done any checks for any changes in the Smart Config "protocol" but did a quick check following your comment. It looks much the same - except now I see:

* len 3 packet
* hi nibble packet
* len 23 packet
* lo nibble packet

Whereas, as shown above, one used to see:

* len 3 packet
* len 23 packet
* hi nibble packet
* lo nibble packet

I.e. the two separator packets instead of coming directly one after the other are now separated by the nibble packets.

Also I see that if you use the AES option they are now using OFB mode rather than ECB. I note also that they seem to be using constant initialization vectors for OFB which *seems* to be a bad idea.

MARK December 10, 2014 at 3:38 AM

George, this confirms what I was seeing, so thanks for taking the time to verifying and the reply. I agree with you. SmartConfig is kind of neat, but also kind of hackish (my word).

Reply

RADAMÉS AJNA January 31, 2015 at 6:26 AM

Great post! you've solved the mystery! thanks

Reply

Enter your comment...

Comment as: Google Account

Publish Preview

Copyright @ 2013 **Depletion Region**.

Designed by Templateify & Sponsored By Twigplay