**MSR TRACK DECRYPTION ALGORITHM**

**BDK:** This is an acronym for Base Derivation Key. This key is known only to the manufacturer and the software developer interfacing with the magstripe scanner.

**IPEK:** This is an acronym for Initial Pin Encryption Key. This key is derived from the **BDK**. This key is injected onto the device by the manufacturer and is used to derive future keys. Compromising the **IPEK** does not compromise the **BDK**.

**KSN**: This is an acronym for Key Serial Number. The KSN is a combo of the serial number of the magstripe scanner and a counter representing the number of swipes that have taken place on the device.

**How It Works**

The BDK is used by the manufacturer to generate the IPEK which is injected onto the device during the manufacturing process. The device uses the IPEK and the KSN to generate a session key that is used to encrypt the data coming off the card.

The BDK is required by the software developer so they can also generate the IPEK. Once they have the IPEK they can query the device for the KSN. The IPEK and KSN are then used to derive the key for that specific transaction/swipe. Once the developer has this key they can easily decrypt the card data.

**Generating the IPEK**

In order to generate the Initial Pin Encryption Key (IPEK) we need the Base Derivation Key (BDK) and the Key Serial Number (KSN). The IPEK is derived using TripleDES encryption. TripleDES is just DES encryption daisy chained together three times.

The algorithm takes a 24 byte key. It runs a DES algorithm three times using bytes 1-8, then again using bytes 9-16 and then a final time with bytes 17-24 as the key for each round respectively.

If the TripleDES algorithm is given a 16 byte key it should use what is known as the EDE3 method. What this means is it will use bytes 1-8, then bytes 9-16, then bytes 1-8 again to fake a 24 byte key.

Some implementations of TripleDES will not do this automatically. I learned this the hard way and spent many hours trying to solve the problem. I was assuming that the particular implementation of TripleDES I was using would fake a 24 bytes key from a 16 byte key.

After a lot of frustration it occurred to me to try taking the first 8 bytes and append it to the end of my key before I fed it into the TripleDES algorithm. This fixed my issue.

**The Details**

The IPEK is going to consist of two 8 byte registers that are produced by two separate TripleDES algorithms. Both will encrypt the 8 most significant bytes of the KSN with the counter zeroed out. The difference between the left and right register is that the right register is going to encrypt the KSN with a slightly modified version of the BDK. I will describe the process below.

Let's assume we have a 16 byte BDK represented as the hexadecimal string "0123456789ABCDEFFEDCBA9876543210".  We also have a 10 byte KSN with an 8 count represented as the hexadecimal string "FFFF9876543210E00008".

We are going to form the BDK used to encrypt the left register of our IPEK by appending the first 8 bytes to the end of the BDK to give us the following 24 byte key.

0123456789ABCDEFFEDCBA98765432100123456789ABCDEF

Now if the KSN is not already 10 bytes in length then pad it to 10 bytes with hexadecimal "F" (1111). It is important to note that the IPEK represents the very first key on the device. This means that we want to generate it with the counter portion of the KSN set to 0. In order to get our KSN with a 0 counter we want to bitmask it with the hexadecimal represented by the string "FFFFFFFFFFFFFFE00000".

FFFF9876543210E00008 and FFFFFFFFFFFFFFE00000

= FFFF9876543210E00000

Great. Now we have our 0 counter KSN. But we only want the most significant 8 bytes of this KSN. We get this by bit shifting this KSN to the right 16 bits.

FFFF9876543210E00000 >> 16 = FFFF9876543210E0

Perfect. The next step is to TripleDES encrypt "FFFF9876543210E0" with the 24 byte "0123456789ABCDEFFEDCBA98765432100123456789ABCDEF" BDK. The result of this encryption should give us the left register of our IPEK.

LR=6AC292FAA1315B4D

If you remember, I mentioned that the right register would use a slightly modified version of the BDK to encrypt our KSN. In order to do this we want to start with our original 16 byte BDK "0123456789ABCDEFFEDCBA9876543210" and XOR it with the following mask "C0C0C0C000000000C0C0C0C000000000". This seems to be a totally arbitrary mask, but alas, it's required to get the right IPEK.

0123456789ABCDEFFEDCBA9876543210 xor C0C0C0C000000000C0C0C0C000000000

= C1E385A789ABCDEF3E1C7A5876543210

We are going to do the same thing we did with the key for the left register and take the most significant 8 bytes and append it to the end to get the following 24 byte key.

C1E385A789ABCDEF3E1C7A5876543210C1E385A789ABCDEF

Go ahead and take the most significant 8 bytes of the KSN with the counter zeroed (as we computed earlier) and TripleDES encrypt it with this new key we just generated. This will give you the right register of your IPEK, producing the following 16 byte IPEK (I've separated the left and right registers for clarity).

RR=858AB3A3D7D5933A

IPEK=6AC292FAA1315B4D 858AB3A3D7D5933A

**Generating Future Keys**

We now have the IPEK. We need to get from the IPEK to the unique key for a specific card swipe (**the session key**). In order to get to this point I am going to define the existence of a black box subroutine that has the sole purpose of returning a single future key. What happens in this blackbox we will not concern ourselves with at the moment. Right now we are only concerned with the preparation of this subroutines inputs.

This blackbox subroutine takes two inputs. One is going to be a key and the other is a message to encrypt. This message is a modification of the KSN.

If you remember, the least significant 21 bits of the KSN holds a counter representing how many card swipes have occurred on the device.  We are going to pass a modified KSN into this subroutine **as many times as there are 1's in the binary representation of that counter**. The key that is passed in along with this modified KSN will be the IPEK on the first iteration and on subsequent iterations it will be the last key produced by the black box subroutine.

Let's begin to modify the KSN. To start we are only concerned with the least significant 8 bytes of the KSN. We also want to zero out the counter portion of the KSN. This can be done by bitmasking the KSN using the mask below.

FFFF9876543210E00008 & 0000FFFFFFFFFFE00000

= 00009876543210E00000

This resulting number is what we are going to use to generate each message we pass into the blackbox. In our example with the 8 counter we only have to pass our inputs through the blackbox once due to the nature of the binary representation of the number 8 (1000). So for demonstration let's assume our counter is actually something more complex like 10 (1010).

**We are going to derive a set of binary numbers from the counter**. In the case of a number like 10 (1010), there exists two binary numbers in this set: 1000, and 0010. Do you see the pattern? We construct a binary number to represent each 1 in the binary form of 10 so that if you added this set together it would equal 11.

We take the first of these numbers and OR it with our 8 LSB zeroed counter KSN we prepared previously as follows (note that this is in hex so our first number is represented as 0008 in hex).

9876543210E00000 | 0000000000000008

= 9876543210E00008

We now pass the IPEK (6AC292FAA1315B4D858AB3A3D7D5933A) as the key and this newly generated KSN variation into the blackbox. The blackbox is going to return a new key. This key is the first future key (represented here in hex): "27f66d5244ff62e1aa6f6120edeb4280".

Now we are going to repeat the process for the next number in the previously constructed binary set, 2 (0010). This time, however, we are going to use the future key we just produced as the key and we are going to generate a new variation of the KSN.

To generate this new variation of the KSN we are going to perform the OR operation again using the last variation we generated: 9876543210E00008.

9876543210E00008 | 0000000000000002

= 9876543210E0000A

Now we pass in our new key, "27f66d5244ff62e1aa6f6120edeb4280", and our new KSN variation "9876543210E0000A" into our blackbox and get out another future key, "6cf2500a22507c7cc776ceadc1e33014".

This is our session key for this device with a counter of 10.

However, our actual counter in this blog post was originally 8, so our real session key is actually the "27F66D5244FF62E1AA6F6120EDEB4280" we computed on the first round.

There is one last operation we have to perform on this value before we've generated our final permutation of the key that is going to allow us to decrypt our data. We must XOR it with "00000000000000FF00000000000000FF".

27F66D5244FF62E1AA6F6120EDEB4280 ^ 00000000000000FF00000000000000FF

= 27F66D5244FF621EAA6F6120EDEB427F

This is the final key we need to decrypt our data.

**The Black Box**

This black box I've been referring to is the algorithm that generates our future keys.  This black box takes the current session key, which I will refer to as *current\_sk,* and a modification of the KSN, which I will refer to as *ksn\_mod*.

If we begin with the assumption that the IPEK we generated above was passed in as the *current\_sk* and that our *ksn\_mod* is "9876543210E00008" that we also generated above.

To begin with we want to take the *current\_sk* and get the most significant 8 bytes and bit shift it 64 bits to the right to get "6AC292FAA1315B4".  This can be done by performing the following bitmask.

6AC292FAA1315B4D858AB3A3D7D5933A & FFFFFFFFFFFFFFFF0000000000000000

= 6AC292FAA1315B4D0000000000000000

At this point we just need to bit shift it 64 bits to the right to get "6AC292FAA1315B4D". We will call this value *left\_key* (seeing as how it's the left side of the *current\_sk*).

Next we want to get the 8 least significant bytes of the *current\_sk* which can be done by the following bitmask.

6AC292FAA1315B4D858AB3A3D7D5933A & 0000000000000000FFFFFFFFFFFFFFFF

= 0000000000000000858AB3A3D7D5933A

Let's call this value (you guessed it) *right\_key*. Now we are going to take the *right\_key* and XOR it with the *ksn\_mod* value "9876543210E00008".

858AB3A3D7D5933A & 9876543210E00008

= 1DFCE791C7359332

This value we will call the *message*. Next we are going to take this *message* value DES encrypt it (that's single DES). We are going to pass the *message* into the DES algo as the content to be encrypted and we are going to pass in *left\_key* as the key to encrypt it with. This should give us "2FE5D2833A3ED1BA". We now need to XOR this value with *right\_key*.

2FE5D2833A3ED1BA ^ 858AB3A3D7D5933A

= AA6F6120EDEB4280

This value is the least significant 8 bytes of our session key! We now just need to repeat the above operation once more with different inputs.  This time we are going to take *current\_sk* and XOR it with "C0C0C0C000000000C0C0C0C000000000". As far as I can tell this value is arbitrary but is part of the ANSI standard so you will just have to take my word for it.

6AC292FAA1315B4D858AB3A3D7D5933A ^ C0C0C0C000000000C0C0C0C000000000

= AA02523AA1315B4D454A7363D7D5933A

If we take this new value "AA02523AA1315B4D454A7363D7D5933A" and use it in place of the *current\_sk* in the operation I described above we should get "27F66D5244FF62E1". This is the most significant 8 bytes of our session key. Combined it should be "27F66D5244FF62E1AA6F6120EDEB4280".