Samsung Pay: Tokenized Numbers, Flaws and Issues

Salvador Mendoza

July 14, 2016

Introduction

Samsung Pay is a new method to make purchases over the latest line of Samsung smartphones devices. The goal is to implement mst technology mainly in every purchase. However, Samsung Pay could use nfc technology as well. Samsung implements a “new” sophisticated alphanumeric algorithm called tokenization. Partner with card providers like Visa, Mastercard and others, Samsung embraced the VTS framework(Visa Token Service) to push its ambitious project. If you are a Samsung Pay user, you do not even think about the new markets goals that this company is planning. Such as online purchases or customizable memberships.

Samsung made a step forward in the electronic market. As result, its purchase process is very interesting. So every time when people add a card at their Samsung Pay, the system generates a new “virtual random” CC implementing the framework which assigns a token to each card. This process is based in another package: Spayfw. That token is saved in a Token Vault somewhere relating the original PAN information. So in each transaction instead of using the original CC’s data, the system sends a tokenized number: a new card number with some “parameters” in the tracks. The main idea behind this is that if someone is able to get a token, he/she will not be able to reuse or extract the original CC’s information.

Analyzing Spay

Before I started digging into the apk, I needed to take a look at the Spay’s databases to have a better understanding of its complexity. I started playing with the terminal and ‘adb’ command, first I backed up the data of Samsung Pay:

```
adb backup com.samsung.android.spay -f sbackup.ab ↵
```

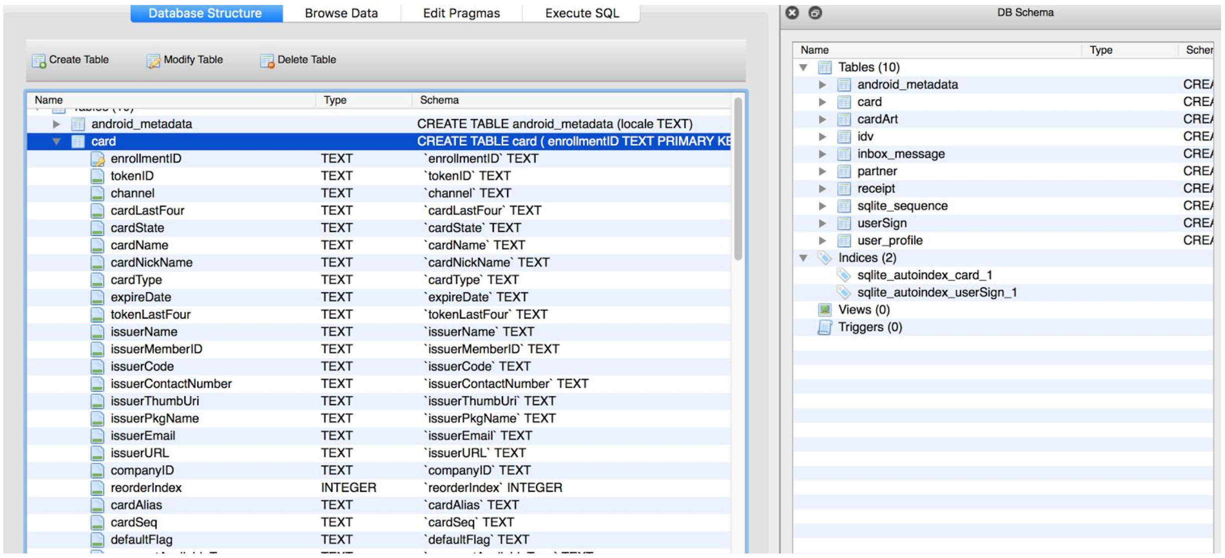
Decompress it using openssl with zlib support or dd with python.

```
dd if=sbackup.ab bs=24 skip=1 | openssl zlib -d > sbackup.tar ↵
```

[Or Python with zlib support.]

```
dd if=sbackup.ab bs=1 skip=24 | python -c "import
zlib,sys;sys.stdout.write(zlib.decompress(sys.stdin.read()))" | tar -xvf - ↵
```

If we open the database with Sqlitebrowser program, we can see how Samsung Pay database is designed. The data is “encrypted” using a private function implementing substitution with static passwords. Some of the fields are CC, last four digits of the token, zip code, card name, token id and many more.



Name	Type	Schema
android_metadata		CREATE TABLE android_metadata (locale TEXT)
card		CREATE TABLE card (enrollmentID TEXT PRIMARY KE
enrollmentID	TEXT	\`enrollmentID\` TEXT
tokenID	TEXT	\`tokenID\` TEXT
channel	TEXT	\`channel\` TEXT
cardLastFour	TEXT	\`cardLastFour\` TEXT
cardState	TEXT	\`cardState\` TEXT
cardName	TEXT	\`cardName\` TEXT
cardNickName	TEXT	\`cardNickName\` TEXT
cardType	TEXT	\`cardType\` TEXT
expireDate	TEXT	\`expireDate\` TEXT
tokenLastFour	TEXT	\`tokenLastFour\` TEXT
issuerName	TEXT	\`issuerName\` TEXT
issuerMemberID	TEXT	\`issuerMemberID\` TEXT
issuerCode	TEXT	\`issuerCode\` TEXT
issuerContactNumber	TEXT	\`issuerContactNumber\` TEXT
issuerThumbUri	TEXT	\`issuerThumbUri\` TEXT
issuerPkgName	TEXT	\`issuerPkgName\` TEXT
issuerEmail	TEXT	\`issuerEmail\` TEXT
issuerURL	TEXT	\`issuerURL\` TEXT
companyID	TEXT	\`companyID\` TEXT
reorderIndex	INTEGER	\`reorderIndex\` INTEGER
cardAlias	TEXT	\`cardAlias\` TEXT
cardSeq	TEXT	\`cardSeq\` TEXT
defaultFlag	TEXT	\`defaultFlag\` TEXT

Name	Type	Scher
Tables (10)		
android_metadata		CRE/
card		CRE/
cardArt		CRE/
idv		CRE/
inbox_message		CRE/
partner		CRE/
receipt		CRE/
sqlite_sequence		CRE/
userSign		CRE/
user_profile		CRE/
Indices (2)		
sqlite_autoindex_card_1		
sqlite_autoindex_userSign_1		
Views (0)		
Triggers (0)		

However this is not the only database/data that Samsung Pay implements to make a purchase. Do you remember that I mentioned the Spayfw package? Well, Spayfw is basically the Visa Token Service framework with a combination of all rules, commands and connection protocols to make possible the tokenization process. Some of its databases and info are very restricted and essential in each transaction. Some of its requested data is stored even in the 'efs' folder which is not easily accessible. Those databases save logs, transaction, memberships and many other things:

PlccCardData_enc.db, spayfw.db, spayfw_enc.db, collector.db, collector_enc.db, mc_enc.db...

Tokenization theory

According to Samsung's explanation, there is no way to anyone could guess any token because the system makes them "randomly." But how random has to be a number to be completely random? Also, they mention that they reduced dramatically the security issues with this technology. However, I found something interesting about its tokenized numbers. Implementing a usb card reader, I started collecting tracks to find patterns or a way for a possible attack. This is a complete string transmitted with MST with Samsung Pay app(I changed the original CC number in the tracks).

%4012300001234567^21041010647020079616?**;**4012300001234567^210410106470200796166?**~**4012300001234567^21041010647020079616?

% = Start sentinel for first track
; = Start sentinel for second track

~ = Start sentinel for third track
^ = Separator
? = End sentinel

If you notice, even when those are three tracks, the tokenized number is the same one in all of them. This is logic right? Each purchase goes with an id, so id = token. Even if the card reader does not detect any data initially, the app has a button to resend the same token again for a second time to be detected. Samsung Pay sends the tokens based in a special configuration to transmit in different baud rate implementing a xml file for this configuration:

Meaning of each term in the file:

t2	Track 2
t1	Track 1
TZ	Trailing zeros
LZ	Leading zeros
r	Baud rate
D	Delay rate

File:

```
<?xml version='1.0'?>
<default transmitTime="15" idleTime="1800" mstSequenceId="v0013.11.T2T2R.T2.T1T2R">
        <!-- L22 -->
        (t2, r200, LZ30, TZ15, D0);
        (t2, r200, LZ15, TZ30, R, D950);
        <!-- L22 -->
        (t2, r200, LZ30, TZ15, D0);
        (t2, r200, LZ15, TZ30, R, D950);
        <!-- L2 -->
        (t2, r800, LZ30, TZ30, D950);
        <!-- L12 -->
        (t1, r300, LZ30, TZ4, D0);
        (t2, r300, LZ6, TZ30, R, D950);
        <!-- L2 -->
        (t2, r800, LZ30, TZ30, D950);
        <!-- L12 -->
        (t1, r300, LZ30, TZ4, D0);
        (t2, r300, LZ6, TZ30, R, D950);
        <!-- L22 -->
        (t2, r200, LZ30, TZ15, D0);
        (t2, r200, LZ15, TZ30, R, D950);
        <!-- L12 -->
        (t1, r300, LZ30, TZ4, D0);
        (t2, r300, LZ6, TZ30, R, D950);
```

```
<!-- L2 -->
(t2, r800, LZ30, TZ30, D950);
<!-- L12 -->
(t1, r300, LZ30, TZ4, D0);
(t2, r300, LZ6, TZ30, R, D950);
<!-- L22 -->
(t2, r200, LZ30, TZ15, D0);
(t2, r200, LZ15, TZ30, R, D0);
[...]
```

Analyzing a token

Why the second track is very important for Samsung Pay? Why the second one and no the first one or third? Well a normal card reader terminal detects the second track as authorization track which has all the data to complete a purchase.

Let's take a look at that token(It does not matter if it is the first or second track, because it is the same token after all).

Splitting the token up:

- The first 16 digits are the new assigned CC number: 4012300001234567

4012300001234567	**401230**	**000**	**01234567**
New CC number	Private BIN #	Never change, from original CC	Still researching

- The last 20 digits are the token's heart: 21041010647020079616

2104-101-0647020079616	**21/04**	**101**	**064702-0079-616**
Token	New expiratio n date.	Service code: **1**: Available for international interchange. **0**: Transactions are authorized following the normal rules. **1**: No restrictions.	**64702**: It handles transaction's range/CVV role. **0079**: Transaction's id, increase +1 in each transaction. **616**: Random numbers, to fill IATA/ABA format, generated from a cryptogram/array method.

Note that when you add a Credit Card with Chip-and-PIN protection, Samsung Pay changes the first service code from 2 to 1 to use it without Chip-and-PIN policies. So the user will be able to make purchases avoiding the necessity to insert the card into the terminal every time.

Token Generation

Researching how the app generates tokens, I found that Samsung Pay implements a cryptogram as base. It uses a combination of arrays and matrices in combination of a “random” number to have a different ids in each token “without” pattern. The main alteration occurs in the last [4-6] digits of the token and in the middle of it. In every transaction, Samsung Pay app sends data to the server; first I thought to match, log or to validate future tokens, but I was wrong. Initially, I assumed that Samsung Pay needs a constant internet connection to have a full control of the tokens creation. But what happens when there is no Internet connection?

When the device was in airplane mode, something interesting occurred. Samsung Pay was able to make purchases in offline mode. In airplane mode, the numbers do not change in the middle section. The app could implement both NFC or MST simultaneously to have a better and wider support in the vendors’ terminals.

I assumed before testing that Samsung Pay had no way to validate the new tokenized number in offline mode. So, the app will have a partial token’s control, and I will be able to create my own random numbers, but I was partially wrong. The tokens have to follow a special structure (a specific random path) in the last 4 digits to be valuable. So are those digits truly random numbers? How the transaction server knows which token is valid even when Spay is offline? Keep these questions in mind.

When the device connects to Internet, Spay increases its tokens +1 in the middle section to make sure that the next tokenized number will be bigger than last one. This means that Samsung Pay is able to make purchases in offline mode, so this counter could be a security range. Some examples:

Without internet; the nfc is active, does not change the middle counter

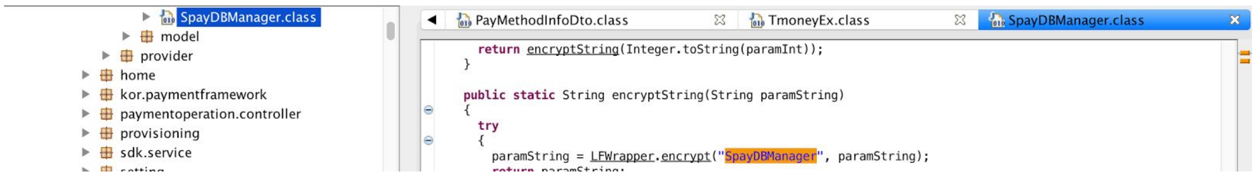
%4012300001234567^21041010**82017(constant)**0216242?
%4012300001234567^21041010**82017**0217826?
%4012300001234567^21041010**82017**0218380?
%4012300001234567^21041010**82017**0219899?
%4012300001234567^21041010**82017**0220006?
[…]

With internet, the middle counter increases +1:

%4012300001234567^21041010**82000**0232646?
%4012300001234567^21041010**82000**0233969?
%4012300001234567^21041010**82000**0234196?
%4012300001234567^21041010**82001**0235585?← **+1**

Flaws and Issues in Samsung Pay

In the application, some flaws are constant: the passwords to encode hashes, comments all over the code, and even weak obfuscation. The credit card data in the database is “encrypted”, but it will be eventually decrypted it with static passwords in the code.



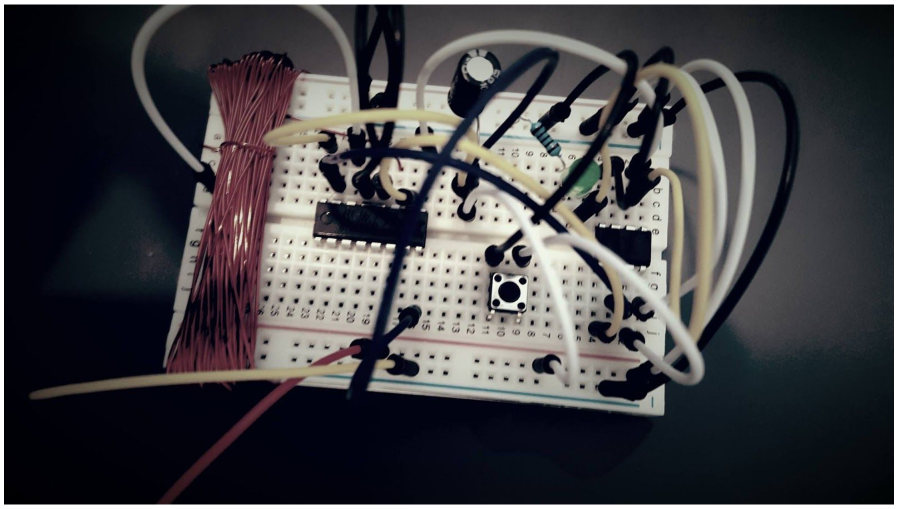
```
    return encryptString(Integer.toString(paramInt));
  }

  public static String encryptString(String paramString)
  {
    try
    {
      paramString = LFWrapper.encrypt("SpayDBManager", paramString);
```

There are many questions that need a deep research like how magically the transaction server can distinguish between faked and proper tokens. I assumed that the server has a reverse of the tokens generator function to validate them.

Attacking Samsung Pay: Scenarios

Scenario 1:

The next step was trying to reuse tokenized numbers, guess them or take an approximation to see how they work separately from Samsung’s technology. If you think about it, the tokens are created implementing visa or mastercard framework at Spayfw package, so they should work without issues in any device. The only thing to be aware is how to transmit the tracks properly. Thanks to Samy Kamkar and his invention(http://samy.pl/magspoof/), I could implement MagSpoof to use tokens from Samsung Pay easily.



With Magspoof, I successfully made purchases with tokens obtained from Samsung Pay. However, I could not reuse them. Every token that goes through, it is burned. So there is no way to reuse it repeatedly. However an attacker could try to guess the last 3 digits of the next token. Analyzing many entries, an attacker can narrow to a small range of possible for future tokens.

```
%4059557240050212^22111014226000128069?~4059557240050212^22111014226000128069?//Monday, june 27
%4059557240050212^22111014226000129860?~4059557240050212^22111014226000129860?
%4059557240050212^22111014226000130648?~4059557240050212^22111014226000130648?
%4059557240050212^22111014226000131953?~4059557240050212^22111014226000131953?
%4059557240050212^22111014301000132466?~4059557240050212^22111014301000132466?//Tuesday junio 28 11:26am
%4059557240050212^22111014301000133576?~4059557240050212&22111014301000133576?
%4059557240050212^22111014301000134764?~4059557240050212^22111014301000134764?
%4059557240050212^22111014301000135708?~4059557240050212^22111014301000135708?
%4059557240050212^22111014315000136483?~4059557240050212^22111014315000136483?//Wed junio 29 9:23am
%4059557240050212^22111014315000137459?~4059557240050212^22111014315000137459?
%4059557240050212^22111014315000138345?~4059557240050212^22111014315000138345?
%4059557240050212^22111014315000139093?~4059557240050212^22111014315000139093?
%4059557240050212^22111014349000140436?~4059557240050212^22111014349000140436?//Friday july 1, 5:34pm
%4059557240050212^22111014349000141752?~4059557240050212^22111014349000141752?
%4059557240050212^22111014349000142461?~4059557240050212^22111014349000142461?
%4059557240050212^22111014349000143073?~4059557240050212^22111014349000143073?
%4059557240050212^22111014393000144157?~4059557240050212^22111014393000144157?//Sat july 2, 5:05pm
%4059557240050212^22111014393000145812?~4059557240050212^22111014393000145812?
%4059557240050212^22111014393000146740?~4059557240050212^22111014393000146740?
%4059557240050212^22111014393000147260?~4059557240050212^22111014393000147260?
%4059557240050212^22111014417000148809?~4059557240050212^22111014417000148809?//Sunday july 3 4:00:05
%4059557240050212^22111014417000149485?~4059557240050212^22111014417000149485?
%4059557240050212^22111014417000150578?~4059557240050212^22111014417000150578?
%4059557240050212^22111014417000150578?~4059557240050212^22111014417000150578?
%4059557240050212^22111014417000151300?~4059557240050212^22111014417000151300?
%4059557240050212^22111014439000152433?~4059557240050212^22111014439000152433?//4:19
%4059557240050212^22111014439000153159?~4059557240050212^22111014439000153159?//4:20
%4059557240050212^22111014439000154227?~4059557240050212^22111014439000154227?
%4059557240050212^22111014440000155547?~4059557240050212^22111014440000155547?//4:38
%4059557240050212^22111014440000156001?~4059557240050212^22111014440000156001?//4:59
%4059557240050212^22111014440000157597?~4059557240050212^22111014440000157597?//5:00
%4059557240050212^22111014440010158248?~4059557240050212^22111014440010158248?//Monday july 4 4:04pm
```

```
%4059557240050212^22111014440010159354?~4059557240050212^22111014440010159354?
%4059557240050212^22111014440010160392?~4059557240050212^22111014440010160392?
%4059557240050212^22111014440010161183?~4059557240050212^22111014440010161183?
%4059557240050212^22111014440010162318?~4059557240050212^22111014440010162318?
%4059557240050212^22111014464010168604?~4059557240050212^22111014464010168604?//Tuesday july 5 3:30pm
%4059557240050212^22111014464010169807?~4059557240050212^22111014464010169807?
%4059557240050212^22111014464010170821?~4059557240050212^22111014464010170821?
%4059557240050212^22111014487000171851?~4059557240050212^22111014487000171851?
%4059557240050212^22111014487000172235?~4059557240050212^22111014487000172235?
%4059557240050212^22111014487000173882?~4059557240050212^22111014487000173882?/Wed July 6 10:51pm
offline/mode
%4059557240050212^22111014487000174133?~4059557240050212^22111014487000174133?
%4059557240050212^22111014487000175454?~4059557240050212^22111014487000175454?
%4059557240050212^22111014487000176404?~4059557240050212^22111014487000176404?
%4059557240050212^22111014487000177103?~4059557240050212^22111014487000177103?
```

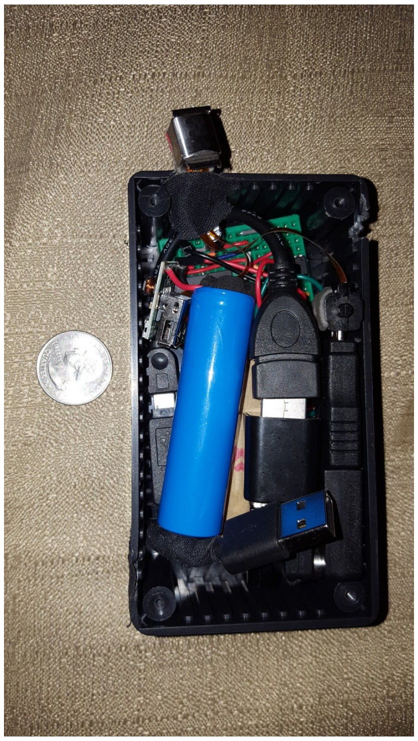
If an attacker analyze the tokens very carefully, he/she could implement a guessing method, a brute force attack or a tokens' dictionary attack.

Scenario 2:

Another possible scenario could be If a Samsung customer tries to use Samsung Pay but something happens in the middle of the transaction, and this does not go through, that token still alive. Meaning that an attacker could jam the transaction process to make Samsung Pay failed and force it to generate the next token. So the attacker will be able to use the previous tokenized number to make a purchase without any restrictions. This attack technique is very similar to Samy Kamkar methodology to attack the rolling code algorithm implementing the RollJam tool. But in this particular case, unlike Samy's attack, the malicious attacker will not need to release the previous token because that token will be used to make the purchase.

Preparing JamPay for Scenario 2

Hardware:
- Raspberry zero captures tokens and send them by email. Also, it will provide the necessary power for MagSpoof and acts like a web server handling requests.
- MagSpoof will act as jammer to confuse the terminal.
- Battery pack
- Jump wires
- Coil
- Wifi usb dongle



Part of the project. MagSpoof is already integrated with Raspberry zero.

Demo: https://www.youtube.com/watch?v=CujkEaemdyE

Solution

Samsung Pay has to work harder in the token's expiration date, to suspend them as quickly as possible after the app generates a new one, or the app may disposed the tokens which were not implemented to make a purchase.
Also Samsung Pay needs to avoid of using static passwords to “encrypt” its files and databases with the same function because eventually someone would be able to reverse it and exploit them. The databases are very sensitive. They contains delicate information to update token status, server connections instructions and validation certificates.

Conclusion

The usage of Samsung Pay is an actual risk for its users’ integrity in different security levels. By providing the correct fixes and updates, Samsung Pay could become one of the most sophisticated purchase app in the digital market.

Samsung Pay is growing exponentially around the planet; adding different services at different countries. As result, Samsung Pay needs to improve its security as soon as possible to have better controls and to avoid any possible attacks against its customers.

Sources

https://samy.pl/magspoof/