Manually unpacking Anubis APK

Published: 2018-08-30 · Archived: 2026-04-02 12:06:59 UTC

I've been seeing people talk about Anubis lately so I decided to take a look at it, unfortunately these led me to a whole bunch of packed APK files. Obviously there are blog posts describing the unpack files but all the hashes are leading me to the packed versions. So what do you do in this situation? Well you learn how to search basically, just like you have to learn how to use your favorite search engine if you have a virustotal account you end up having to figure out how to search for whatever you're looking for. Take a look at the jadx-gui picture in this phishlabs writeup[1], in this writeup we can see a number of strings but if we search for the twitter address on VirusTotal[2] then we come up with a number of classes.dex files.

Searching for the twitter address from the aforementioned writeup leads us to a number of dex files in VirusTotal

We can then pivot backwards from this file to see where this dex file came from by utilizing the ITW(In The Wild) tab which will show that it came from a file bundle.

This file bundle is just a zipped up classes.dex file, using the ITW tab again we see it was created during the execution of another file.

This parent file is an APK with a similar looking obfuscation as the other files I had looked at from reading reports! So these obfuscated APKs are creating these Anubis DEX files which is actually a common occurrence with packed APK files that keep an encoded DEX file on board as a resource.

Another hint that this is packed is by taking a look at the manifest inside this APK.

We can see lots of referenced code in the manifest which doesn't actually exist in the current decompiled DEX file, this is another very big indicator that we're dealing with an encoded DEX file in this APK. So the idea is that initial execution in this APK will decode the hidden classes.dex file and replace the current one with that one. Since the resources has a file called 'files' which is just binary data I assume all my theories up until now are true, we could then just execute the APK and catch the decoded classes.dex file like how sandboxes do but that's not really any fun.

So a few possible ways to attack finding the relevant code section that will be responsible for decoding the dex file, we can trace execution through the obfuscated code of the current dex file and look for possibilities or interesting functions, you can look for where the resource object gets loaded and then trace that, or you can just blindly look for functions that might appear to be doing something interesting and then backtrack. You'd actually be surprised how often number 3 works after you have a few years experience with reverse engineering malware.

For this one however because of all the garbage code that's been addded I just literally searched the decompiled code for "^ " and ended up finding an interesting little function that was being called with an array of integers over and over again by the int values were changing(obfuscated strings?).

If you've spent much time reversing encoding and encryption algorithms you might recognize the general flow of that code block but I won't ruin the surprise for now for the rest of you.

Backtracking we can see this function and others named the same, this is basically an overloaded function which can make tracing execution a little painful as you have to match up which function does what based on which parameters are passed. Ofcourse since there's garbage code and obfuscation that can be easier said than done sometimes.

So this function takes an array and then it sets the byte array that it ends up XORing so what is this 'OAeAqJYuzXcD'? Searching for it shows that it's built as a byte array.

The value being passed in for the length of the array is 256 so this a byte array of length 256. Searching for this array some more shows that it's also used with the same 256 value and filled with data based on a byte array passed in. So this looks like it's building an SBOX similar to RC4. Looking for how this all gets called shows that it ends up being called near the top of the 'com.lpapxwl.bemtobai.SVBkpSlwf' class.

We can continue to follow that plus the previously identified function that was XORing the SBOX back to a section of code near the top of the same code page.

What stands out there is that another 256 byte array is being built and then passed in to the 'dlPiWCFOIB' function and then passed into the overloaded function that builds the SBOX like we previously found. Looking up the 'dlPiWCFOIB' function shows that it is initializing the SBOX.

So could the array of integers at the top of the decoding overview screenshot be the RC4 key then? Let's test it on the binary data blob we found in the resources.

```
 >>> a = "(byte) 75, (byte) 41, (byte) -22, (byte) 1, (byte) -99, (byte) -118, (byte) 73, (byte) 34, (byte) 71,
>>> a.split('(byte) ')
['', '75, ', '41, ', '-22, ', '1, ', '-99, ', '-118, ', '73, ', '34, ', '71, ', '-89, ', '-26, ', '11, ', '-21,
>>> b = a.split('(byte) ')
>>> b
['', '75, ', '41, ', '-22, ', '1, ', '-99, ', '-118, ', '73, ', '34, ', '71, ', '-89, ', '-26, ', '11, ', '-21,
>>> b = b[1:]
>>> b
['75, ', '41, ', '-22, ', '1, ', '-99, ', '-118, ', '73, ', '34, ', '71, ', '-89, ', '-26, ', '11, ', '-21, ',
>>> b[-1].split(',')
['-95']
>>> map(lambda x: x.split(', '),b)
[['75', ''], ['41', ''], ['-22', ''], ['1', ''], ['-99', ''], ['-118', ''], ['73', ''], ['34', ''], ['71', ''],
>>> map(lambda x: x.split(', ')[0],b)
['75', '41', '-22', '1', '-99', '-118', '73', '34', '71', '-89', '-26', '11', '-21', '24', '-108', '-24', '24',
>>> c = map(lambda x: x.split(', ')[0],b)
>>> c
['75', '41', '-22', '1', '-99', '-118', '73', '34', '71', '-89', '-26', '11', '-21', '24', '-108', '-24', '24',
>>> map(int,c)
[75, 41, -22, 1, -99, -118, 73, 34, 71, -89, -26, 11, -21, 24, -108, -24, 24, 89, 20, 91, -49, 104, -99, -16, 2
```

```
>>> d = map(int,c)
>>> map(lambda x: x & 0xff, d)
[75, 41, 234, 1, 157, 138, 73, 34, 71, 167, 230, 11, 235, 24, 148, 232, 24, 89, 20, 91, 207, 104, 157, 240, 27,
>>> e = map(lambda x: x & 0xff, d)
>>> map(chr,e)
['K', ')', '\xea', '\x01', '\x9d', '\x8a', 'I', '"', 'G', '\xa7', '\xe6', '\x0b', '\xeb', '\x18', '\x94', '\xe8
>>> ''.join(map(chr,e))
'K)\xea\x01\x9d\x8aI"G\xa7\xe6\x0b\xeb\x18\x94\xe8\x18Y\x14[\xcfh\x9d\xf0\x1bI&\x85\xc4\x0e\xb9\xfcf\xa0%.\x9b\x
>>> f = ''.join(map(chr,e))
>>> rc4 = ARC4.new(f)
>>> rc4.decrypt(data)[:500]
'\x88P\xe3"\x8d\xfa{A\x9d\xe2\xf3\xd67\x80\x0f(\xfc\xf8\'\xff\xe7\xf9Ul\xff\x9b\x9eQ{\xa1\xde\xad6\xd1\xb0Y9\xf9
```

Well that didn't work, so let's take a look at the binary data a little closer.

```
 >>> data[:100]
'\x9a\xb8\x01\x00B\xa3\xe1&\xdbY\x9agN\xbb\xc44vv\x8ch?\x12\x89/\xd9\xeb(N"p\xbd\x1fY\xd1\x00\xde\x0es\xc3\xe2D'
```

```
 >>> import struct
>>> struct.unpack_from('<I', data)
(112794,)
>>> len(data)
225596
```

It's definately possible, so let's try decrypting past that.

```
 >>> rc4 = ARC4.new(f)
>>> rc4.decrypt(data[4:])[:500]
'PK\x03\x04\x14\x00\x00\x00\x08\x00\xad\x85\x0fMGt\xb5\x9c"\xb8\x01\x00\xf4\r\x04\x00\x0b\x00\x00\x00classes.dex
```

A quick look at the decoded dex file shows lots of interesting data including our twitter string from earlier.

Looking around at some of the other code shows a few interesting routines.

So going off prior research into Anubis we know that the twitter data is then base64 decode to a hexlified string, so let's find where that twitter string gets used.

Here we can see the twitter string being used along with it referencing the tags for pulling out the data, following one of the later function calls if we're assuming it's first going to base64 decode and unhexlify leads us to the following function.

So could this be the RC4 key then? Let's test with the hexlified string from the phishlabs report.

```
 >>> m = "3090c08a8f3c3950d98c612399622d02057bce22a5b8b01e4dc3960fa03648c822f3"
>>> a = binascii.unhexlify(m)
```

```
>>> a
'0\x90\xc0\x8a\x8f<9P\xd9\x8ca#\x99b-\x02\x05{\xce"\xa5\xb8\xb0\x1eM\xc3\x96\x0f\xa06H\xc8"\xf3'
>>> l = ARC4.new('flash1')
>>> w = l.decrypt(a)
>>> w
'hxxps://lukasstefankotiywlepok.com'
```

It works! That’s it, hope it helps! For further reading I’ve included a number of references to android unpacking articles.
