Challenge description:

"You're entering a world of pain" [cit.]

Solution

Uploading the file in MOBSF, we can see it has 1 activity and 1 provider, and an unknown permission, which is com.mobiotsec.gnirts.DYNAMIC\_RECEIVER\_NOT\_EXPORTED\_PERMISSION.

Instead, we can see interesting APIs already called.

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

We can see also vulnerabilities over the certiifcates in debug and prone to collision, while also being open to debug/not open to anti-VM code.

Also, let’s launch: > jadx -d out gnirts.apk

We already know where to go at this point (folders: out/sources/com/mobiotsec/gnirts). Inspecting the code, we can understand many things:

* the flag has the usual format FLAG{}
* there is a substring in the middle of 26 chars (from 5 to 31)
* at indexes 8/15/21 is split with hyphens (because we know it’s 135 divided in 3 indices)
* this is split in base64
* there is some purposefully placed mess like (bim/bum/bam) over chars, where each one respects a regex
  + bim controls lowercase chars
  + bum controls uppercase ones
  + bam controls alphanumeric ones
* again, hashing at the end as a function

We know the string is 135 chars. Inside */out/resources/res/values* in the jax decompiled app, we can find:

*<string name="ct1">xwe</string>*

*<string name="ct2">asd</string>*

*<string name="ct3">uyt</string>*

*<string name="ct4">42s</string>*

*<string name="ct5">70 IJTR</string>*

*<string name="flag" />*

*<string name="k1">53P</string>*

*<string name="k2">,7Q</string>*

*<string name="k3">8=A</string>*

*<string name="k4">yvF</string>*

*<string name="k5">dxa</string>*

*<string name="m1">slauqe</string>*

*<string name="search\_menu\_title">Search</string>*

*<string name="status\_bar\_notification\_info\_overflow">999+</string>*

*<string name="t1">82f5c1c9be89c68344d5c6bcf404c804</string>*

*<string name="t2">e86d706732c0578713b5a2eed1e6fb81</string>*

*<string name="t3">7ff1301675eb857f345614f9d9e47c89</string>*

*<string name="t4">b446830c23bf4d49d64a5c753b35df9a</string>*

*<string name="t5">1b8f972f3aace5cf0107cca2cd4bdb3160293c97a9f1284e5dbc440c2aa7e5a2</string>*

The *me* function checks if the two hashes are equal and then reverses the string:

*me -> check if two hash are equals*

*me = stringCompare(context, string, string)*

*private static boolean me(Context ctx, String s1, String s2) {*

*Log.e(TAG, "s1: " + s1 + " s2: " + s2);*

*try { // "slauqe"*

*return ((Boolean) s1.getClass().getMethod(r(ctx.getString(R.string.m1)), Object.class).invoke(s1, s2)).booleanValue();*

*// r\_reverse\_the\_string(slauqe) --> equals*

The *dh* function simply creates an hash based on the string and converts it into hex. The *gs* function takes the string and returns it literally (from *ct5 = 70 IJTR* up to *k5 = dxa*.

The right track to follow would be decrypting t1 up to t4, as seen from here:  
 if (me(ctx, dh(gs(ctx.getString(R.string.ct1), ctx.getString(R.string.k1)), ps[0]), ctx.getString(R.string.t1)) && me(ctx, dh(gs(ctx.getString(R.string.ct2), ctx.getString(R.string.k2)), ps[1]), ctx.getString(R.string.t2)) && me(ctx, dh(gs(ctx.getString(R.string.ct3), ctx.getString(R.string.k3)), ps[2]), ctx.getString(R.string.t3)) && me(ctx, dh(gs(ctx.getString(R.string.ct4), ctx.getString(R.string.k4)), ps[3]), ctx.getString(R.string.t4))) {

The *t1-t4* part seems like an MD5 hash. Using infact <https://www.md5online.it/index.lm>:

* t1 = sic
* t2 = parvis
* t3 = magna

For the latest one, it’s also MD5, but this site finds it: <https://www.dcode.fr/md5-hash>

* t4 = 28jAn1596

We also know sic-parvis-MAGNA-28jAn1596 is the combination

The latest part it’s also an hash, which is created selected literally over the algorithm SHA-256 algorithms. So, the flag would be:

*FLAG{sic-parvis-MAGNA-28jAn1596}*

If you do the SHA-256 of that, you infact find:

1b8f972f3aace5cf0107cca2cd4bdb3160293c97a9f1284e5dbc440c2aa7e5a2