quebremesepuder.apk

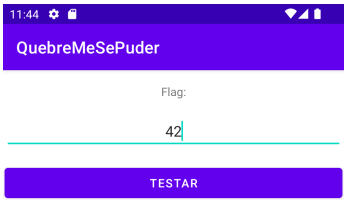
Write-up do desafio `quebremesepuder.apk`.

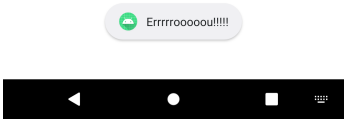
@eduardo.vasconcelos

✅ Reversing (Android)
✅ Reversing (binário)
✅ Crypto

Pré-análise

Ao abrir o app, nota-se que uma interface contendo uma `EditText` espera a flag. Se a flag errada é submetida, uma mensagem de erro é apresentada:



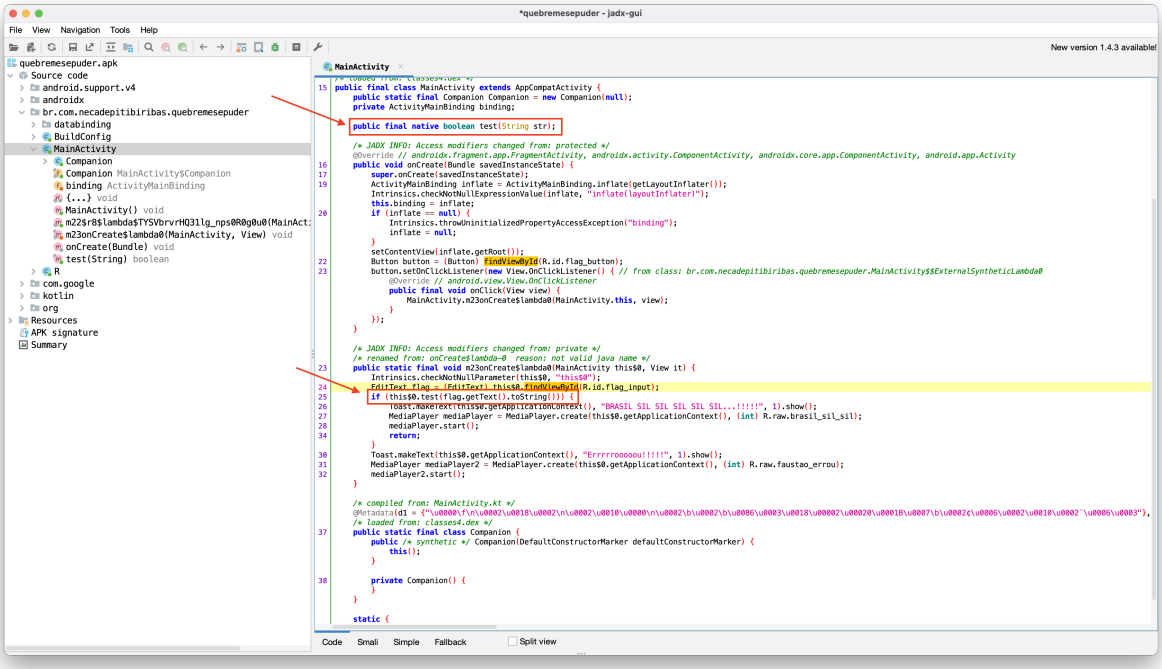


Portanto, *ao que parece,* para resolver o desafio, a flag correta precisa ser submetida na `EditText`.

Análise

Passo 1. Reversão do APK e análise do código Java

Ao reverter o app (e.g. usando o JADX), a `MainActivity` revela a chamada efetuada para testar a flag submetida. Trata-se de um método nativo: `boolean test(String str)`:

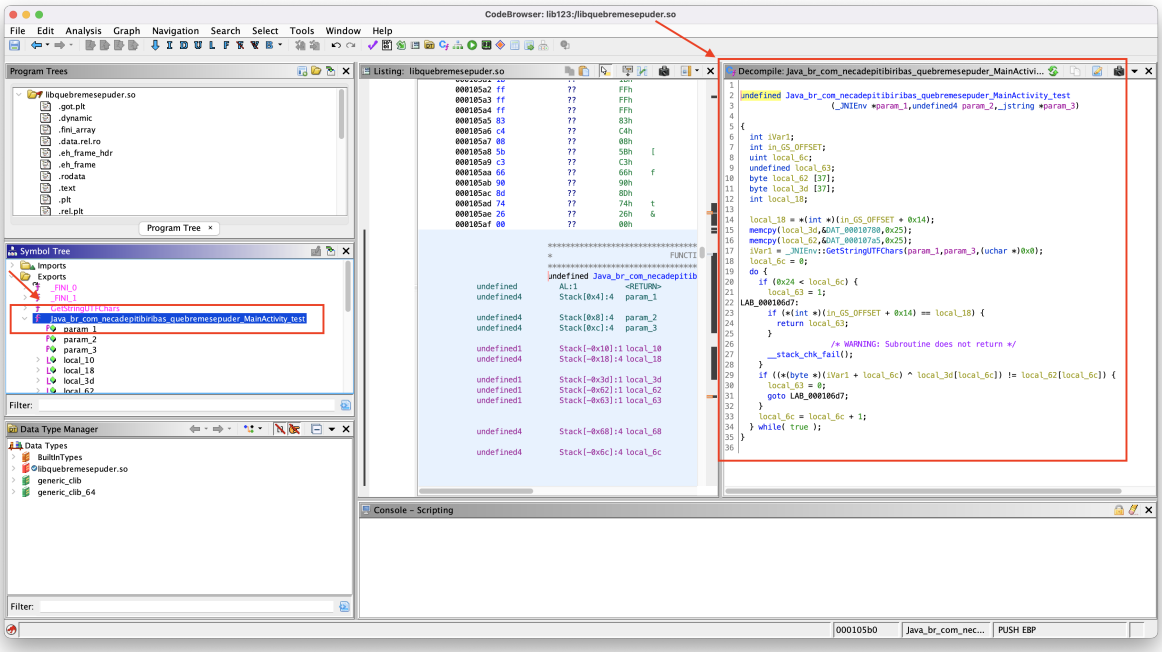


Nota-se que tal método é definido em uma biblioteca chamada `quebremesepuder`:

É necessário, então, reverter o binário da biblioteca para continuar a análise.

Passo 2. Reversão do binário da biblioteca e análise do código nativo

Ao descompactar o app (e.g. `unzip quebremesepuder.apk`) e reverter uma das versões da biblioteca em questão (e.g. `lib/x86/libquebremesepuder.so`), é possível visualizar a implementação do método `test()` (e.g. usando o Ghidra):



O método recebe a string correspondente à flag candidata como `param_3`. Internamente, a variável local `iVar1` aponta para a cadeia de caracteres em questão:

```
// ...
/* ... */ _jstring *param_3)

{
    int iVar1;

    // ...

    iVar1 = _JNIEnv::GetStringUTFChars(param_1,param_3,(uchar *)0x0);

    // ...
```

A análise do código revertido leva à conclusão de que o método carrega duas constantes para as variáveis locais `local_3d` e `local_62`, ambas de comprimento 37 bytes:

```
// ...

byte local_62 [37];
byte local_3d [37];

// ...

memcpy(local_3d,&DAT_00010780,0x25);
memcpy(local_62,&DAT_000107a5,0x25);

// ...
```

Ao examinar a listagem do programa revertido em linuguagem de montagem (no próprio Ghidra, na coluna "Listing"), verifica-se que tais constantes têm os seguintes valores:

- `DAT_00010780`:

```
# ...

                            //
                            // .rodata
                            // SHT_PROGBITS  [0x780 - 0x7c9]
                            // ram:00010780-ram:000107c9
                            //
                            DAT_00010780
XREF[4]:     Java_br_com_necadepitibiribas_qu

Java_br_com_necadepitibiribas_qu

Java_br_com_necadepitibiribas_qu
```

```
_elfSectionHeaders::0000023c(*)
        00010780 61              ??         61h    a
        00010781 72              ??         72h    r
        00010782 61              ??         61h    a
        00010783 72              ??         72h    r
        00010784 61              ??         61h    a
        00010785 71              ??         71h    q
        00010786 75              ??         75h    u
        00010787 61              ??         61h    a
        00010788 72              ??         72h    r
        00010789 61              ??         61h    a
        0001078a 61              ??         61h    a
        0001078b 72              ??         72h    r
        0001078c 61              ??         61h    a
        0001078d 72              ??         72h    r
        0001078e 61              ??         61h    a
        0001078f 71              ??         71h    q
        00010790 75              ??         75h    u
        00010791 61              ??         61h    a
        00010792 72              ??         72h    r
        00010793 61              ??         61h    a
        00010794 61              ??         61h    a
        00010795 72              ??         72h    r
        00010796 61              ??         61h    a
        00010797 72              ??         72h    r
        00010798 61              ??         61h    a
        00010799 71              ??         71h    q
        0001079a 75              ??         75h    u
        0001079b 61              ??         61h    a
        0001079c 72              ??         72h    r
        0001079d 61              ??         61h    a
        0001079e 61              ??         61h    a
        0001079f 72              ??         72h    r
        000107a0 61              ??         61h    a
        000107a1 72              ??         72h    r
        000107a2 61              ??         61h    a
        000107a3 71              ??         71h    q
        000107a4 75              ??         75h    u

# ...
```

- DAT_000107a5:

```
# ...

                            DAT_000107a5
XREF[3]:     Java_br_com_necadepitibiribas_qu

Java_br_com_necadepitibiribas_qu

Java_br_com_necadepitibiribas_qu
```

```
          000107a5 08              ??         08h
          000107a6 34              ??         34h    4
          000107a7 0d              ??         0Dh
          000107a8 13              ??         13h
          000107a9 06              ??         06h
          000107aa 0a              ??         0Ah
          000107ab 14              ??         14h
          000107ac 2d              ??         2Dh    -
          000107ad 3e              ??         3Eh    >
          000107ae 3e              ??         3Eh    >
          000107af 18              ??         18h
          000107b0 42              ??         42h    B
          000107b1 14              ??         14h
          000107b2 20              ??         20h
          000107b3 3e              ??         3Eh    >
          000107b4 13              ??         13h
          000107b5 41              ??         41h    A
          000107b6 12              ??         12h
          000107b7 41              ??         41h    A
          000107b8 3e              ??         3Eh    >
          000107b9 55              ??         55h    U
          000107ba 00              ??         00h
          000107bb 52              ??         52h    R
          000107bc 2d              ??         2Dh    -
          000107bd 23              ??         23h    #
          000107be 42              ??         42h    B
          000107bf 19              ??         19h
          000107c0 51              ??         51h    Q
          000107c1 3c              ??         3Ch    <
          000107c2 06              ??         06h
          000107c3 3e              ??         3Eh    >
          000107c4 06              ??         06h
          000107c5 51              ??         51h    Q
          000107c6 2d              ??         2Dh    -
          000107c7 14              ??         14h
          000107c8 44              ??         44h    D
          000107c9 08              ??         08h
  # ...
```

Logo, `local_3d` contém a string "araraquara" repetida diversas vezes e truncada aos 37 bytes, e `local_62` contém uma sequência de 37 bytes aparentemente aleatórios.

Continuando a análise do código revertido, verifica-se que, após a cópia de tais constantes, o método executa o seguinte laço:

```
local_6c = 0;
do {
    if (0x24 < local_6c) {
        local_63 = 1;
LAB_000106d7:
        if (*(int *)(in_GS_OFFSET + 0x14) == local_18) {
```

```
            return local_63;
            }
            /* WARNING: Subroutine does not return */
            __stack_chk_fail();
        }
        if ((*(byte *)(iVar1 + local_6c) ^ local_3d[local_6c]) !=
    local_62[local_6c]) {
            local_63 = 0;
            goto LAB_000106d7;
        }
        local_6c = local_6c + 1;
    } while( true );
```

Do código acima: o método inicializa o contador `local_6c` com 0, e o incrementa ao final do laço. Durante as iterações, este contador é utilizado como offset para acessar valores nas sequências de bytes para as quais apontam `iVar1`, `local_3d` e `local_62`. A cada iteração, o método compara (i) o resultado da operação de XOR (`^`) entre o byte no mesmo endereço relativo em `iVar1` e `local_3d` com (ii) o byte no mesmo endereço relativo em `local_62`:

```
// ...

if ((*(byte *)(iVar1 + local_6c) ^ local_3d[local_6c]) !=
local_62[local_6c]) {

// ...
```

Caso essa comparação resulte no valor lógico "falso", a variável local `local_63` recebe o valor 0 e o fluxo de execução é desviado para o label `LAB_000106d7:`, o que resulta no método retornar o valor lógico "falso".

Entretanto, caso a comparação resulte sempre no valor lógico "verdadeiro" até a 36ª iteração ($(36)_{10}$ = 0x24), ao adentrar a 37ª iteração, a condicional `if (0x24 < local_6c) {` resulta em "verdadeiro", o que faz o método retornar o valor lógico "verdadeiro".

Portanto, é possível concluir que:

> O método aparenta *descriptografar* um segredo, contido no segmento `DAT_000107a5` (aquela "sequência de 37 bytes aparentemente aleatórios"), utilizando a chave "araraquara" e o algoritmo Repeating Key XOR, para comparar o texto em claro de tal segredo com a flag submetida.

Logo:

> Descriptografar os bytes do segmento `DAT_000107a5`, utilizando os mesmos chave criptográfica e algoritmo, deve resultar na flag necessária para resolver o desafio.

Passo 3. Descriptografar o segredo para obter a flag

Extraindo a chave e o segredo a partir da listagem do programa revertido em linuguagem de montagem, pode-se chegar ao seguinte script Python que implementa a operação de descriptografar o segredo para

obter a flag:

```
#!/usr/bin/python3

from operator import xor

e = [0x08, 0x34, 0x0D, 0x13, 0x06, 0x0A, 0x14, 0x2D, 0x3E, 0x3E, 0x18,
0x42, 0x14, 0x20, 0x3E, 0x13, 0x41, 0x12, 0x41, 0x3E, 0x55, 0x00, 0x52,
0x2D, 0x23, 0x42, 0x19, 0x51, 0x3C, 0x06, 0x3E, 0x06, 0x51, 0x2D, 0x14,
0x44, 0x08]
k = "araraquaraararaquaraararaquaraararaqu"

p = ""

for i in range(0, len(k)):
   p = p + chr(xor(ord(k[i]), e[i]))

print(p)
```

De fato, executar o script acima resulta na flag desejada: