xhs的shield字段分析

[軍]

极客

举报

10小时前

131

因为某些事情，对与此款app的分析断断续续的进行了一个月之久。最近，终于这段时间把它搞完了。以此发帖来记录一下，并给出一些经验之谈，以便交流。本贴仅用于交流，望谅解。下面进入正题。

一、apk静态分析

此前先来谈一下它的防护手段。我想，它是利用的okhttp的拦截器这一点，大家应该都知道。若有朋友对okhttp的工作原理不太清楚，可以参看此篇帖子 定位到小红书sign算法。读完此篇帖子，想必你对okhttp的工作原理有了一个大致的了解，重点在于okhttp的拦截器机制。

了解完之后直接抓包，我选的突破点还是在手机号登录模块。

```
POST https://www.xiaohongshu.com/api/sns/v4/user/login/password HTTP/1.1
X-B3-TraceId: 656566ea2e180e0f
xy-common-params:
deviceId=5194ac73-e712-3fb7-ac22-c5ca8d7cb04b&identifier_flag=2&fid=159706423810f73c043456d3994aeb92756507fcdc53&device_fingerprint1=20200425125549 95da036c942a7eadc27f985061ae0b270159ea13070f506f&uis=light&device_fingerprint=20200425125549 95da036c942a7eadc27f985061ae0b270159ea13070f506f&versionName=6.56.0&platform=android&sid=&t=1597235126&x_trace_page_current=&lang=zh-Hans&channel=HuaweiCloud
User-Agent: Dalvik/2.1.0 (Linux; U; Android 5.1.1; hm note 1s Build/LMY48Z) Resolution/900*1600 Version/6.56.0 Build/6560162 Device/(xiaomi;hm note 1s)
discover/6.56.0 NetType/WiFi
shield: a91dce8a8bdfcaadb2dc5172eb9129f8    //32
xy-platform-info: platform=android&build=6560162&deviceId=5194ac73-e712-3fb7-ac22-c5ca8d7cb04b
Content-Type: application/x-www-form-urlencoded
Content-Length: 684
Host: www.xiaohongshu.com
Connection: Keep-Alive
Accept-Encoding: gzip

password=722809cbe154b543f79011f5b8b14fdc&imei=&zone=86&android_version=22&mac=00%3A81%3A04%3A05%3A3c%3A2e&imei_encrypted=QbxAkre1Id8N0shw2EafBg%3D%3D&traceId=1597235083428_fddf4a1d-199c-4525-8eeb-efef502a32d0&phone=15822601106&type=phone&imsi=460008850750168&android_id=11f9cf539347b6ed&deviceId=5194ac73-e712-3fb7-ac22-c5ca8d7cb04b&identifier_flag=2&fid=159706423810f73c043456d3994aeb92756507fcdc53&device_fingerprint1=20200425125549 95da036c942a7eadc27f985061ae0b270159ea13070f506f&uis=light&device_fingerprint=20200425125549 95da036c942a7eadc27f985061ae0b270159ea13070f506f&versionName=6.56.0&platform=android&sid=&t=1597235126&x_trace_page_current=&lang=zh-Hans&channel=HuaweiCloud
```

关注点在拦截器类 com.xingin.shield.http.XhsHttpInterceptor。如何定位到这里的就不多说了，懂得okhttp工作原理的很容易就能定位到此处。值得一提的是，xhs的包并不在java层发送，是在so中发送的，即so反调java层的方法。这也就意味着我们在java层并不能得到shield字段的数据信息，想要揭秘它，那么我们就要进入native层。

下面再进入intercept函数。如下图，

```
public Response intercept(Interceptor.Chain chain) throws IOException {
    Response response;
    long currentTimeMillis = ContextHolder.javaLogAble ? System.currentTimeMillis() : 0;
    a<Request> aVar = this.predicate;
    if (aVar == null || aVar.a(chain.request())) {
        response = intercept(chain, this.cPtr);
    } else {
        response = chain.proceed(chain.request());
    }
    if (ContextHolder.javaLogAble) {
        Log.i("XhsHttpInterceptor", "cost:" + (System.currentTimeMillis() - currentTimeMillis));
    }
    return response;
}
```

此处重写了Interceptor的Interceptor函数。接下来，把焦点放在intercept函数上。

```
public native Response intercept(Interceptor.Chain chain, long j) throws IOException;
```

此函数是在libshield.so中实现的，ida打开此so，找到data段，然后往下翻，找到此处。

```
DCD aInitializenati     ; DATA XREF: sub_73604+44↑o
                        ; .ppp.ttl:off_736A8↑o
                        ; "initializeNative"
DCD aV                  ; "()V"
DCD sub_740E4+1
DCD aIntercept          ; "intercept"
DCD aLokhttp3Interc     ; "(Lokhttp3/Interceptor$Chain;J)Lokhttp3/"...
DCD sub_73B78+1
DCD aInitialize         ; "initialize"
DCD aLjavaLangStrin_12  ; "(Ljava/lang/String;)J"
DCD sub_73960+1
DCD aDestroy            ; "destroy"
DCD aJV                 ; "(J)V"
DCD sub_73B24+1
```

一共有四个注册函数，不着急，我们一个一个的来说一下。

1. initializeNative：讲真，此函数做的事虽然挺多的，但我们的关注点并不在这里。它查找了so中需要调用的java方法。

2.intercept：重点关注函数。

3.initialize：也是做了一些初始化工作，与shield字段有着亲密的联系。

4.destroy：顾名思义，从函数名就可以看出来。

下面，进入intercept函数的分析。

```
nt __fastcall sub_73B78(JNIEnv *env, jobject a2, jobject a3_chain, jlong a4)
```

首先，准备工作做好，参数类型修改好，前两个是动态注册的函数自带的，后两个是传过来的两个参数。

chain暂且放到一边，其实它也没有什么好说的，那来说一下a4。它其实是函数initialize函数返回的一个值，一个非常重要的地址。

接下来，继续分析 可以观察到里面有大量的sub_9858函数。如下图

```
v63_chain = v5_chain;
v8 = sub_9858(v4_env, v5_chain, dword_8F14C, v7);
v10 = sub_9858(v4_env, v8, dword_8F154, v9);
v12 = sub_9858(v4_env, v10, dword_8F17C, v11);
v14 = sub_9858(v4_env, v10, dword_8F180, v13);
v64 = sub_9858(v4_env, v8, dword_8F184, v15);
v62 = v8;
v17 = sub_9858(v4_env, v8, dword_8F188, v16);
v19 = sub_B574(v4_env, v77, dword_8F1A4, v18);
sub_9858(v4_env, v19, dword_8F1A8, v12);
if ( v14 )
  sub_9858(v4_env, v19, dword_8F1A8, v14);
v21 = v17;
v65 = v19;
v22 = sub_9D34(v4_env, v17, dword_8F190, v20);
_aeabi_memclr4(&v72, 20);
v74 = &v72;
```

点进去是这样的，

```
jobject __fastcall sub_9858(JNIEnv *env, int a2, int a3, int a4)
{
  int v5; // [sp+14h] [bp-4h]

  v5 = a4;
  return (*env)->CallObjectMethodV(env, a2, a3, &v5);
}
```

在这里稍微解释下CallObjectMethodV函数，env就不多说了，a2是要调用的java函数所属的类，a3是methodID，v5是函数参数。

由此可知，sub_9858函数封装了一个调用函数。继续往下走，

```
v67 = v4_env;
sub_617CC(s1, "xy-platform-info", &s2);
v28 = *s1;
v29 = v19;
```

找到了这个字段，遗憾的是，并没有看到shield字段。但是我们似乎initializeNative函数，看完这个函数，就打开了一片新天地。我们心心念念的shield就在这里有出现，

```
sprintf(&v33, "platform=android&build=%lld&deviceId=%s", v6, v8, v12);
(*v1)->ReleaseStringUTFChars(v1, v10, v12);
(*v1)->DeleteLocalRef(v1, v10);
v13 = (*v1)->NewStringUTF(v1, &v33);
dword_8F1D8 = (*v1)->NewGlobalRef(v1, v13);
((*v1)->DeleteLocalRef)(v1, v13);
v14 = (*v1)->NewStringUTF(v1, "xy-platform-info");
dword_8F1D4 = (*v1)->NewGlobalRef(v1, v14);
((*v1)->DeleteLocalRef)(v1, v14);
v15 = (*v1)->NewStringUTF(v1, "shield");
dword_8F1D0 = (*v1)->NewGlobalRef(v1, v15);
((*v1)->DeleteLocalRef)(v1, v15);
result = _stack_chk_guard - v34;
```

在sub_73B78函数找到此处，

```
v55 = sub_9858(v50_env, v62, dword_8F158, v54);/
sub_9858(v50_env, v55, dword_8F15C, dword_8F1D0)
sub_9858(v50_env, v55, dword_8F15C, dword_8F1D4)
```

我们会发现，shield和类okhttp3.Request的内部类Builder中的一个方法header有关联。

```
public Builder header(String str, String str2) {
    this.headers.set(str, str2);
    return this;
}
```

此处的两个参数，一个是name，另一个是value。
这个结果我是从动态调试中得来的，如果没有进行动态调试，那么我们的静态分析到这里就结束了。

总结一下我们get到的点，

```
sub_9858(v50_env, v55, dword_8F15C, dword_8F1D0);
```

此处是关键点是确认无疑的，那么随之而来的疑问有两个，函数跑到此处的时候调用的java方法，即header方法的两个参数分别是什么，这个问题我们在java层就可以找到答案，点进去set函数，

```
public final Builder set(String str, String str2) {
    Headers.checkName(str);
    Headers.checkValue(str2, str);
    removeAll(str);
    addLenient(str, str2);
    return this;
}
```

到此，我们可以基本确定这两个参数分别是name和value。
那么下一个疑问是此处shield字段被赋值了没有，如果没有赋值，那么shield字段的计算在更后面，若被赋值了，那么shield的计算在此之前就已经完成。

二、native调试

1、ida动态调试

关于此方面，我想着重说一下它的防护手段。

1.java层：此方面的防护主要是anr，isDebuggerConnected，waitingForDebugger，或者再加上个这个ANRWatchDog。

2.native层：此处主要为libshield.so中的signature和tracepid。

解决这两方面的防护手段有两种方式。一种为hook，另一种为重打包。

先看hook方式，这里需要java层的hook+native层的hook。xposed举起了手，但是很可惜的是xposed和ida动态调试会产生化学反应，直接卡死。

到这里，hook似乎走进死胡同了。

接下来再看重打包方式，对于上一种，重打包似乎就显得比较高明一些，但要将整个apk整体反编译的时候，资源的混淆就会是另一个棘手的点。

当然， 对于此也有解决方案，即手动修改xml文件，修改或删除不存在的资源。但，奈何需要处理的xml太多了。故，果断放弃。那么，怎么解决呢

直接把apk的dex文件拖出来反编译成smali文件，进行smali文件的修改，然后再把smali文件回编译成dex文件，替换apk的dex文件，重新签名。

同时，还可以修改libshield.so文件。下面大体说一下，

```
v1 = a1;
v2 = (*(**(a1 + 76) + 24))();
v3 = v30;
if ( v2 < 0 )
  v3 = 0;
v4 = ((*v3)->FindClass)(v3, "android/app/Application");
v28 = v4;
v5 = ((*v3)->GetMethodID)(v3, v4, "getPackageManager", "()Landroid/content/pm/PackageManager;");
v7 = sub_9858(v3, *(v1 + 80), v5, v6);
v8 = ((*v3)->FindClass)(v3, "android/content/pm/PackageManager");
v9 = ((*v3)->GetMethodID)(v3, v8, "getPackageInfo", "(Ljava/lang/String;I)Landroid/content/pm/PackageInfo;");
v10 = sub_A3D8(v1);
v27 = v7;
v11 = sub_9858(v3, v7, v9, v10);
v12 = ((*v3)->FindClass)(v3, "android/content/pm/PackageInfo");
v13 = ((*v3)->GetFieldID)(v3, v12, "signatures", "[Landroid/content/pm/Signature;", 64);
v25 = ((*v3)->FindClass)(v3, "android/content/pm/Signature");
v14 = ((*v3)->GetMethodID)(v3, v25, "hashCode", "()I");
v26 = v11;
v15 = ((*v3)->GetObjectField)(v3, v11, v13);
v16 = ((*v3)->GetArrayLength)(v3, v15);
time(&timer);
```

此处为signature的检测点，我们只需要把这个函数给nop掉就可以。

```
prctl(15, "anti-ptrace");
v0 = getpid();
_aeabi_memclr8(&v5, 30);
sprintf(&v5, "proc/%d/status", v0);
v1 = fopen(&v5, "r");
if ( v1 )
{
  while ( 1 )
  {
    while ( feof(v1) )
    {
ABEL_5:
      fclose(v1);
      sleep(3u);
      v1 = fopen(&v5, "r");
      if ( !v1 )
        goto LABEL_6;
    }
    fgets(&v3, 256, v1);
    if ( !strncmp(&v3, "TracerPid", 9u) )
    {
      if ( atoi(&v4) )
        goto LABEL_8;
      goto LABEL_5;
    }
  }
}
```

此处为tracepid的检测点，同样的把这个函数给nop掉就可以。

到此处，我们通过重打包的方式解决了它的大部分防护手段。

将重打包后的apk安装进手机，apk能够跑起来，情况似乎好起来了。接下来进入ida动态调试环节，很遗憾的是进行ida动态调试的时候apk会崩溃，

崩溃的原因大概率是灾难性故障。似乎，这条路也让我们走窄了。。。

2、frida hook

接下来我们另觅它路，既然ida不行，那么我们就试试frida能不能获取到我们想要的函数参数信息和内存信息。下面展示一下hook代码。

```
function get_rva(module, offset) {
    var base_addr = Module.findBaseAddress(module);
     if (base_addr == null)
        send("its null");
send("module is found！ ");
    var target_addr = base_addr.add(offset);
send(target_addr);
    return target_addr;
}

var target_addr = get_rva("libshield.so", 0x73B78);

Interceptor.attach(ptr(target_addr), {
    onEnter: function(args) {
        send("function is found！ ");
    },
    onLeave: function(retval) {

    },
});
```

其实，这个hook代码有没有都无所谓，因为hook的时候程序会崩溃。起初，我以为此app有frida框架的检测，但是我没有找到有关的防护手段。

后来，我用一个没有frida检测的apk测试了一下，发现同样会崩溃。更让人意外的是，这种hook有时候可以，有时候会崩溃。最终，我并没有找到缘由。

就这样不了了之了。但，这段代码是可用的。

3、inline hook

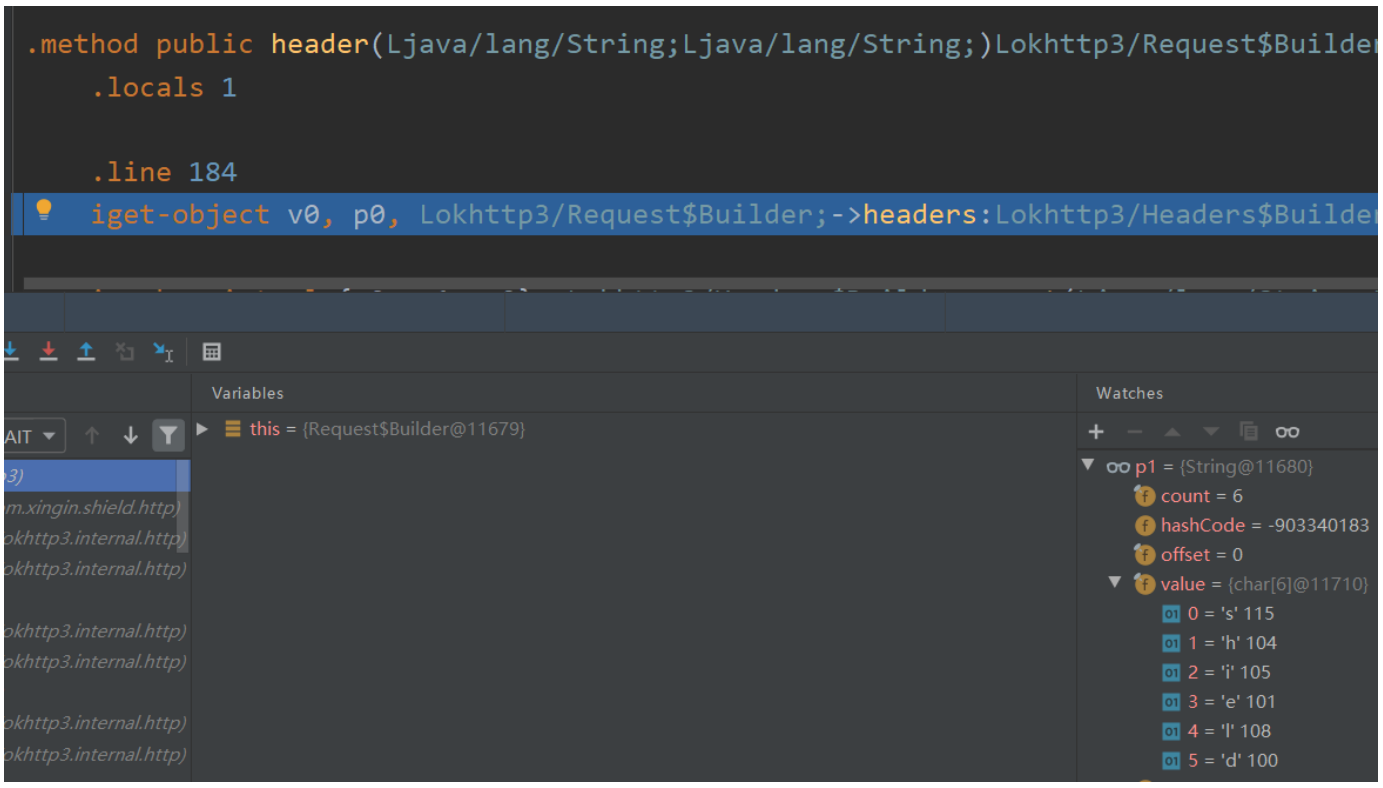
相对于我们要获取多个函数的参数信息来说，这条路显然有点繁琐。但好消息是，这条路确实能够走通。通过inline hook，我们基本上可以把shield算法

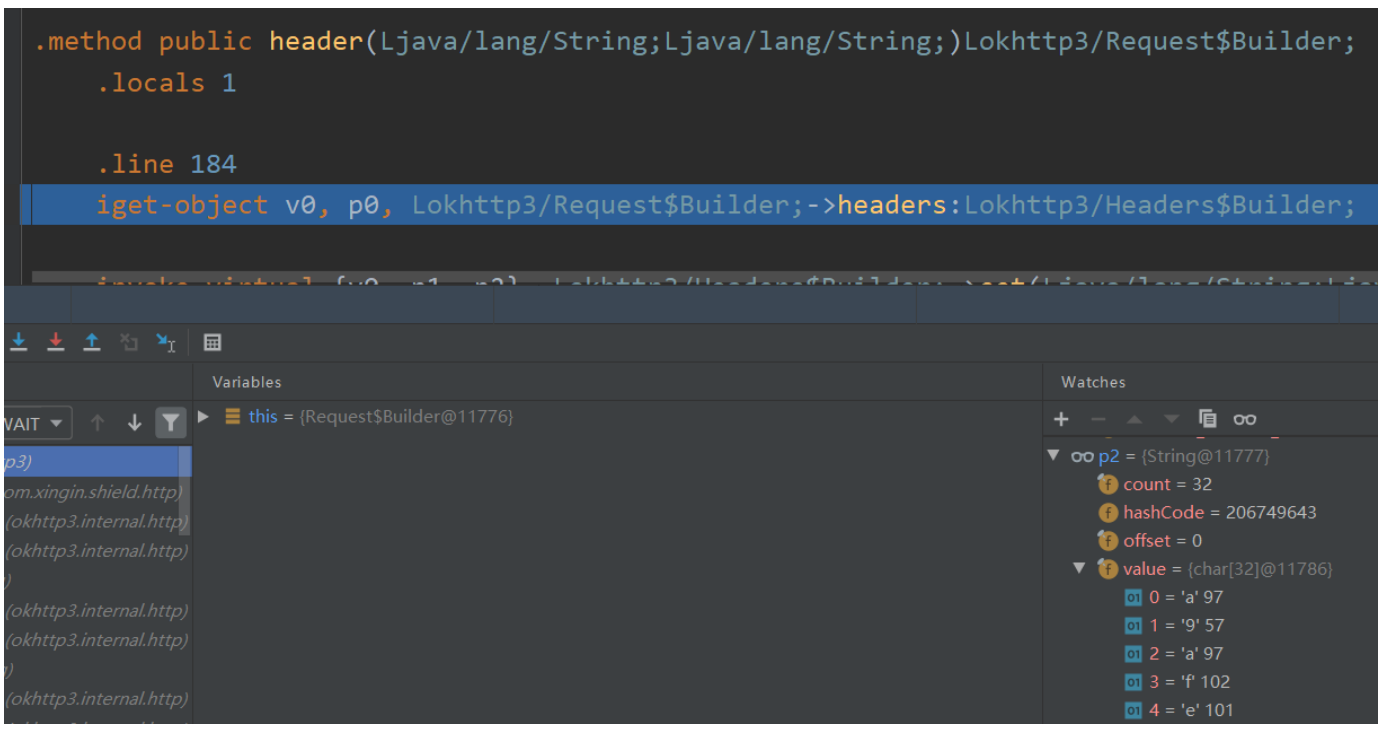
给分析出来。代码就不发了，网上教程挺多的。。。

三、AS动态调试

还记得前面提到的 Request$Builder 类中的header方法吗，从这里我们把shield字段分成了有没有被赋值两个部分，那么下面来验证一下这个函数的

两个参数分别是什么。





p1和p2就是函数header的两个参数，分别是 "shield" ，"a9afefd8f6fc8cbf29cdb876eecaf848" 。由此，我们才可以真正的确定shield字段在此时已经

被赋值完成了。

那么，我们下一步要做的就是大体定位一下计算shield字段的函数在哪个地方。

在函数的前半部分，我发现了一个循环。如图

```
v74 = &v72;
v73 = &v72;
v66 = v22;
if ( v22 >= 1 )                                 // v22=3
{
  v23 = 0;
  v61 = v21;
  do
  {
    v24 = sub_9858(v4_env, v21, dword_8F194, v23);// dword_8F194 name
    v25 = (*v4_env)->GetStringUTFChars(v4_env, v24, 0);
    *s1 = 0;
    if ( strlen(v25) >= 4 )
    {
      v26 = *v25;
      s1[2] = v25[2];
      *s1 = v26;
      if ( !strcmp(s1, &dword_74078) )
      {
        v27 = sub_9858(v4_env, v21, dword_8F198, v23);// dword_8F198 values
        sub_617CC(&s2, v25, &v68);
        *sub_B5C4(&v71, &s2) = v27;
        sub_5F3A4(&s2);
        v21 = v61;
      }
    }
    (*v4_env)->ReleaseStringUTFChars(v4_env, v24, v25);
    (*v4_env)->DeleteLocalRef(v4_env, v24);
    ++v23;
  }
  while ( v23 < v66 );
}
```

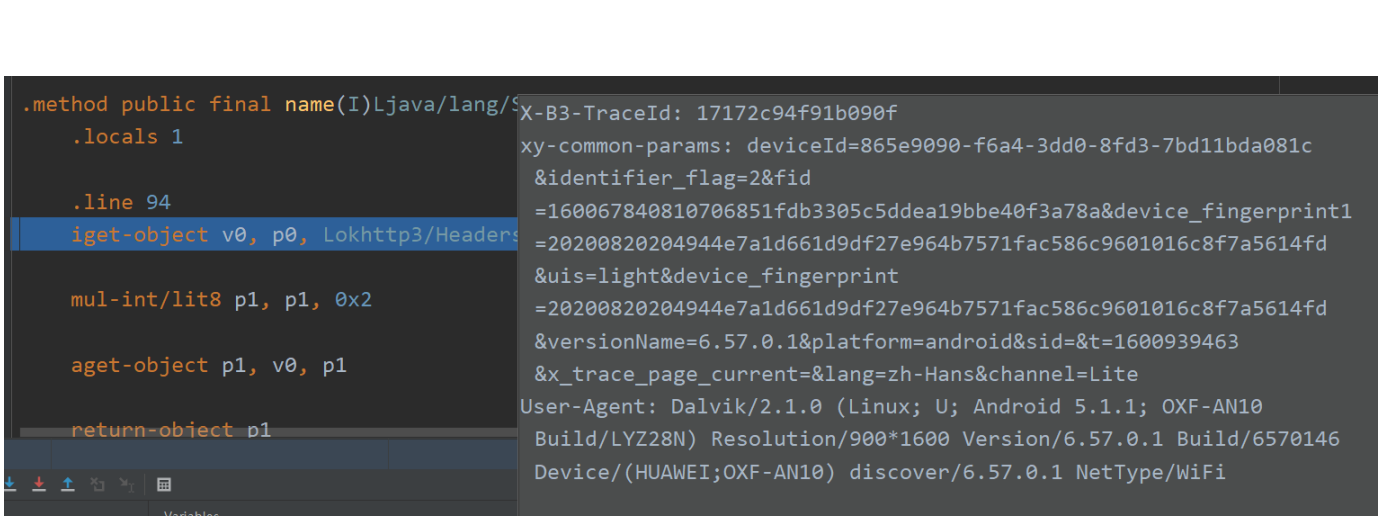
从图上可以看出，我已经写了一部分注释。那么下面的这两个函数在java中的哪个类中呢

```
dword_8F194 = ((*v1)->GetMethodID)(v1, v26, "name", "(I)Ljava/lang/String;");
dword_8F198 = ((*v1)->GetMethodID)(v1, v26, "value", "(I)Ljava/lang/String;");
```

当然，它并不难找，所以我就不分析了，直接告诉大家。它在 okhttp3.Headers 类中

```
public final String name(int i) {
    return this.namesAndValues[i * 2];
}

public final String value(int i) {
    return this.namesAndValues[(i * 2) + 1];
}
```

接下来，我们看看这个循环中有没有牵扯到shield字段。



发现这里并没有shield字段的信息，那么我们继续往下找。我们发现了这个字段

```
v67 = v4_env;
sub_617CC(s1, "xy-platform-info", &s2);
v28 = *s1;
v29 = v19;
```

紧接着是一段比较长的逻辑，下面贴一下图

```
v28 = *s1;
v29 = v19;
if ( *(*s1 - 4) <= -1 || (sub_604D8(s1), v28 = *s1, *(*s1 - 4) < 0) )
{
  v30 = v28;
}
else
{
  sub_604D8(s1);
  v30 = *s1;
}
v31 = *(v30 - 3);
if ( *(v30 - 1) <= -1 )
{
  v32 = v30;
}
else
{
  sub_604D8(s1);
  v32 = *s1;
}
v33 = &v30[v31];
while ( v33 != v28 )
{
  v34 = *v28++;
  *v32++ = toupper(v34);
}
if ( v75 )
{
  v35 = v73;
  if ( v73 == &v72 )
    goto LABEL_57;
  v36 = 0;
  do
  {
    sub_60FBC(&s2, v35 + 4);

      v37 = s2;
      if ( *(s2 - 1) <= -1 || (sub_604D8(&s2), v37 = s2, *(s2 - 1) < 0) )
      {
        v38 = v37;
      }
      else
      {
        sub_604D8(&s2);
        v38 = s2;
      }
      v39 = *(v38 - 3);
      if ( *(v38 - 1) <= -1 )
      {
        v40 = v38;
      }
      else
      {
        sub_604D8(&s2);
        v40 = s2;
      }
      v41 = v38 + v39;
      if ( v41 != v37 )
      {
        do
        {
          v42 = *v37++;
          *v40++ = toupper(v42);
        }
        while ( v41 != v37 );
        v40 = s2;
      }
      v43 = *(v40 - 3);
      v44 = v35[5];
      v45 = *(*s1 - 12);
      v46 = *(*s1 - 12);
      if ( v45 > v43 )
```

```
      v46 = *(v40 - 3);
    v47 = memcmp(*s1, v40, v46);
    if ( !v47 )
      v47 = v45 - v43;
    v50 = __OFSUB__(v47, -1);
    v48 = v47 == -1;
    v49 = v47 + 1 < 0;
    v51 = 0;
    if ( !((v49 ^ v50) | v48) )
      v51 = 1;
    if ( !((v51 | v36) << 31) )
    {
      sub_9858(v67, v65, dword_8F1A8, dword_8F1D8);// dword_8F1A8 writeString
      v36 = 1;
    }
    sub_9858(v67, v65, dword_8F1A8, v44);
    (*v67)->DeleteLocalRef(v67, v44);
    sub_5F3A4(&s2);
    v35 = sub_4D50C(v35);
  }
  while ( v35 != &v72 );
  v29 = v65;
  if ( !(v36 << 31) )
LABEL_57:
    sub_9858(v67, v29, dword_8F1A8, dword_8F1D8);
}
else
{
  sub_9858(v67, v65, dword_8F1A8, dword_8F1D8);
}
```

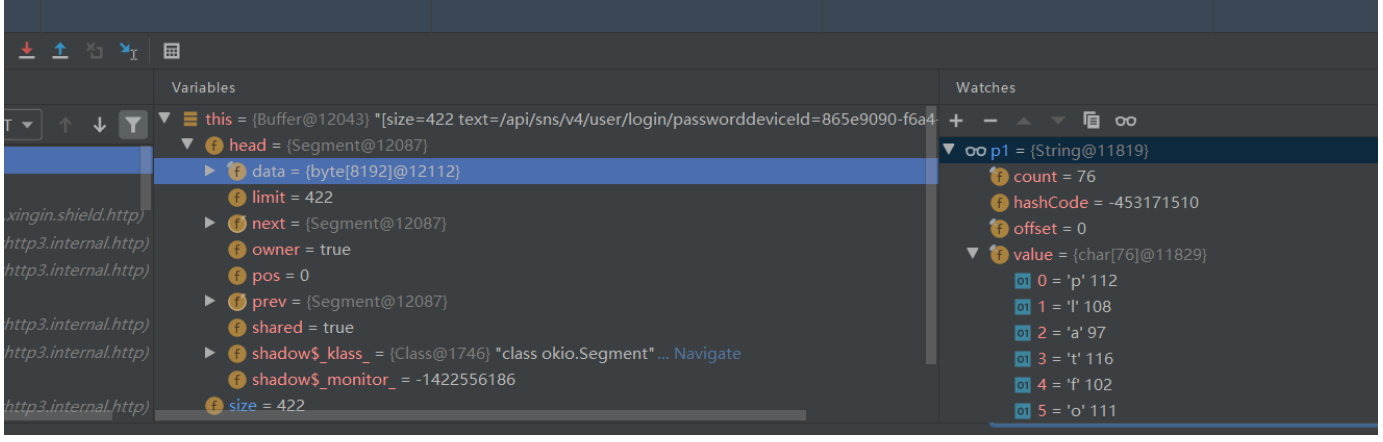
我们并不能确定这段逻辑里面有没有shield字段的信息，所以我找了一个点来确定一下，就是我打注释的那个java层的函数。

这个函数在 okio.Buffer 类里，如下图

```
public final Buffer writeString(String str, Charset charset) {
    return writeString(str, 0, str.length(), charset);
}

public final Buffer writeString(String str, int i, int i2, Charset charset) {
    if (str == null) {
        throw new IllegalArgumentException("string == null");
    } else if (i < 0) {
        throw new IllegalAccessError("beginIndex < 0: " + i);
    } else if (i2 < i) {
        throw new IllegalArgumentException("endIndex < beginIndex: " + i2 + " < " + i);
    } else if (i2 > str.length()) {
        throw new IllegalArgumentException("endIndex > string.length: " + i2 + " > " + str.length());
    } else if (charset == null) {
        throw new IllegalArgumentException("charset == null");
    } else if (charset.equals(Util.UTF_8)) {
        return writeUtf8(str, i, i2);
    } else {
        byte[] bytes = str.substring(i, i2).getBytes(charset);
        return write(bytes, 0, bytes.length);
    }
}
```

下面，我们来看一下这个函数的参数信息。

```
.method public final writeString(Ljava/lang/String;IILjava/nio/charset/Charset;)Lokio/Buffer;
    .locals 1

    if-eqz p1, :cond_5

    if-ltz p2, :cond_4

    if-lt p3, p2, :cond_3

    .line 1087
    invoke-virtual {p1}, Ljava/lang/String;->length()I
```



可以看到p1的值与我们抓到的包的 xy-platform-info 字段可以对应起来。那么，到这里，我们可以确定前面的那部分逻辑跟shield字段并没有

关系，只是参与了 xy-platform-info 字段的生成。分析到这里，jni动态注册的native层函数就只剩一小部分逻辑了。如下图

```
if ( v64 )
  sub_98A8(v67, v64, dword_8F164, v29);
if ( *(a4 + 650) )                                  //
{
  v50_env = v67;                                    //
  sub_ABB8(v67, v29, a4, v52);
}
else
{
  v50_env = v67;
  sub_AD14(v67, v29, dword_8F1CC, a4);
}
v55 = sub_9858(v50_env, v62, dword_8F158, v54);
```

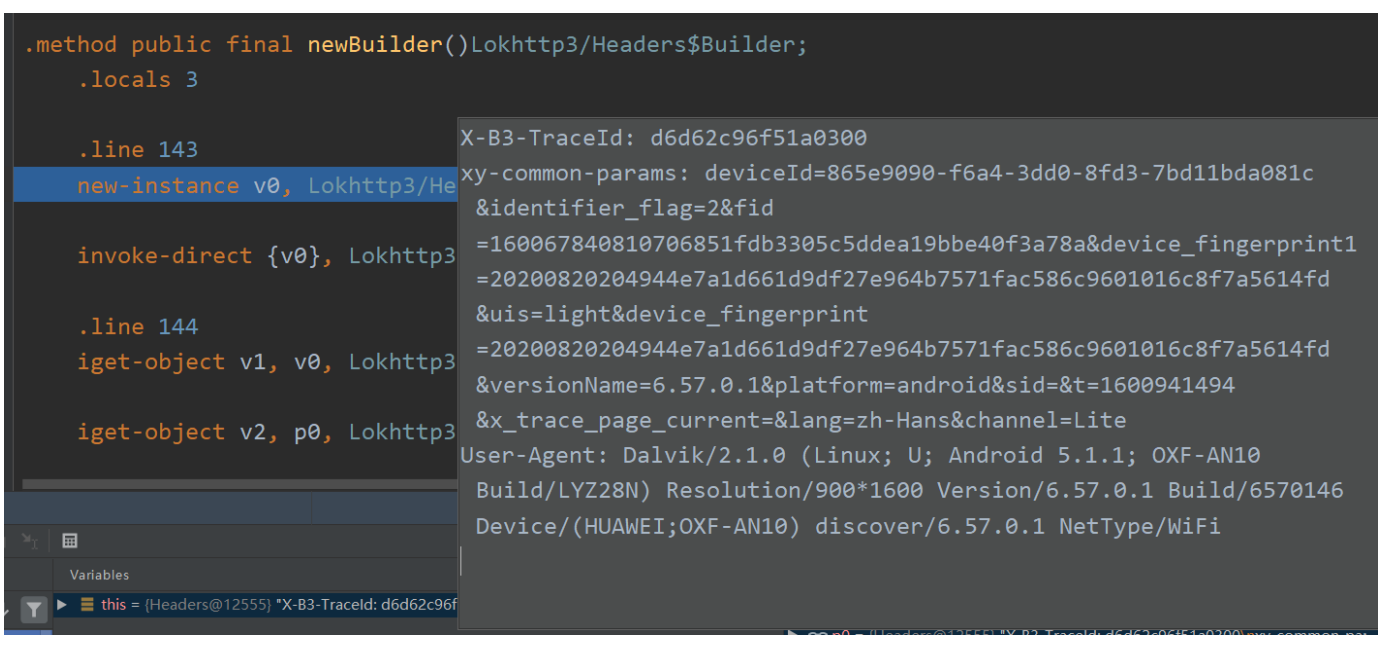
接下来我们分析一下 dword_8F158 所代表的函数，即此函数

```
public final Builder newBuilder() {
    return new Builder(this);
}
```

下面，我们看看此函数有没有shield字段的信息

```
.method public final newBuilder()Lokhttp3/Headers$Builder;
    .locals 3

    .line 143
    new-instance v0, Lokhttp3/He

    invoke-direct {v0}, Lokhttp3

    .line 144
    iget-object v1, v0, Lokhttp3

    iget-object v2, p0, Lokhttp3
```

```
X-B3-TraceId: d6d62c96f51a0300
xy-common-params: deviceId=865e9090-f6a4-3dd0-8fd3-7bd11bda081c
 &identifier_flag=2&fid
 =160067840810706851fdb3305c5ddea19bbe40f3a78a&device_fingerprint1
 =20200820204944e7a1d661d9df27e964b7571fac586c9601016c8f7a5614fd
 &uis=light&device_fingerprint
 =20200820204944e7a1d661d9df27e964b7571fac586c9601016c8f7a5614fd
 &versionName=6.57.0.1&platform=android&sid=&t=1600941494
 &x_trace_page_current=&lang=zh-Hans&channel=Lite
User-Agent: Dalvik/2.1.0 (Linux; U; Android 5.1.1; OXF-AN10
 Build/LYZ28N) Resolution/900*1600 Version/6.57.0.1 Build/6570146
 Device/(HUAWEI;OXF-AN10) discover/6.57.0.1 NetType/WiFi
```



这个函数里并没有shield字段的信息，所以我们到现在可以确定shield字段的计算就在那个if else 结构中。

最后，我们随便点进去那两个函数去看看，结果收到了一份预料之中的惊喜。

```
if ( !strcmp(dest, "S1") )
{
  sub_1E450(v21, a5, v20);
  *a6 = 16;
}
else if ( !strcmp(dest, "S2") )
{
  sub_1E528(v21, a5, v20);
  *a6 = 16;
}
else if ( !strcmp(dest, "S3") )
{
  sub_1E600(v21, a5, v20);
  *a6 = 32;
}
else if ( !strcmp(dest, "S4") )
{
  sub_1E6D8(v21, a5, v20);
  *a6 = 32;
}
else if ( !strcmp(dest, "S5") )
{
  sub_21F90(v21, a5, v20);
  *a6 = 16;
}
else if ( !strcmp(dest, "S6") )
{
  sub_229D4(v21, a5, v20);
  *a6 = 16;
}
else if ( !strcmp(dest, "S7") )
{
  sub_23388(v21, a5, v20);
  *a6 = 16;
}
else if ( !strcmp(dest, "S8") )
{
  sub_23D3C(v21, a5, v20);
  *a6 = 16;
}
else if ( !strcmp(dest, "S9") )
{
  sub_24780(v21, a5, v20);
  *a6 = 16;
}
else if ( !strcmp(dest, "S10") )
{
  sub_25134(v21, a5, v20);
  *a6 = 16;
}
else if ( !strcmp(dest, "S11") )
{
  sub_25AD8(v21, a5, v20);
  *a6 = 16;
}
else if ( !strcmp(dest, "S12") )
{
  sub_2648C(v21, a5, v20);
  *a6 = 16;
}
if ( src )
  operator delete[](src);
if ( dest )
  operator delete[](dest);
return 0;
```

接触过shield算法的大概都知道，它分为s1-s12，这里就出现了很明显的特征。

四、完结

到此为止，关于shield的分析就结束了。经过上面的分析，我感jiao路已经挺宽了，剩下的可以靠inline hook去分析了，后面的部分就不发出来了，毕竟这牵扯到一些大牛的利益。。。希望各位朋友把这篇帖子能当做一个参考吧，总体来说此款apk的防护力度也不是很大，还是有很多可操作的空间吧！
最后我会把我在分析过程中用到的一些东西发到附件上。

本来一开始也没想写这么啰嗦的，但是写着写着就写成小作文了。望大家见谅吧！！

[公告]请完善个人简历信息，好工作来找你！

上传的附件：

附件.zip （2.30MB，2次下载）

最新回复 (1)

mb_foyotena 9小时前 2 楼 0

临时

小红书比较简单，毕竟也没啥东西，还有网页版

wx_nu无情

内容

回帖 表情 高级回复

返回