MobSF 框架及源代码分析

Auth : Cryin’
Date : 2017.02.08
Link : https://github.com/Cryin/Paper

MobSF

MobSF，全称（Mobile-Security-Framework），是一款优秀的开源移动应用自动测试框架。该平台可对安卓、苹果应用程序进行静态、动态分析，并在 web 端输出报告。静态分析适用于安卓、苹果应用程序，而动态分析暂时只支持安卓应用程序。

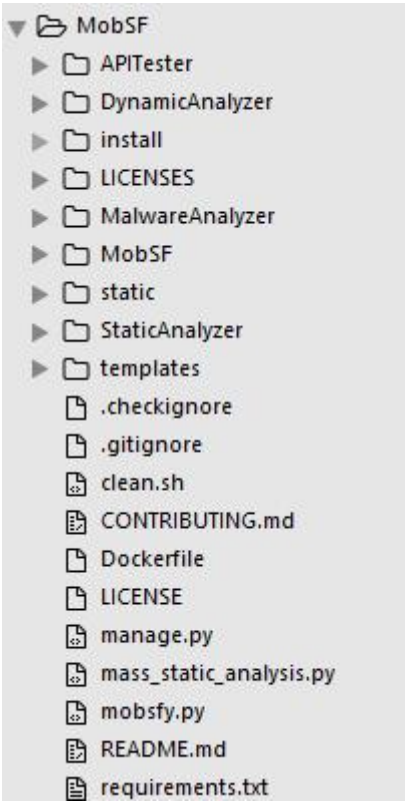
MobSF 使用 Django 框架开发，使用 sqlite 进行的存储，支持对 apk、ipa 及 zip 压缩的源代码进行扫描分析。

```
Only APK,IPA and Zipped Android/iOS Source code supported now!'
```

同时，MobSF 也能够通过其 API Fuzzer 功能模块，对 Web API 的安全性进行检测，如收集信息，分析安全头部信息，识别移动 API 的具体漏洞，如 XXE、SSRF、路径遍历，以及其他的与会话和 API 调用有关的逻辑问题。

代码结构

MobSF 源代码结构主要包含静态分析、动态分析、API Fuzzer 三个部分，本文不关注 Django 框架本身及 Web 处理相关的内容。如图:



静态分析实现原理

静态分析的处理流程集功代码在目录 StaticAnalyzer\views\android 下 static_analyzer.py 程序文件中。分析代码流程可知，在 MobSF 框架中静态分析主要包含三个部分，分别是 Manifest Analysis、Cert Analysis、Code Analysis。流程如下:

Manifest Analysis

在解压 apk 后,MobSF 使用 AXMLPrinter2.jar 工具提取 app 中的 AndroidManifest.xml 文件内容，并进行分析。

```
151         man_data_dic = {
152             'services' : svc,
153             'activities' : act,
154             'receivers' : brd,
155             'providers' : cnp,
156             'libraries' : lib,
157             'perm' : dvm_perm,
158             'packagename' : package,
159             'mainactivity' : mainact,
160             'min_sdk' : minsdk,
161             'max_sdk' : maxsdk,
162             'target_sdk' : targetsdk,
163             'androver' : androidversioncode,
164             'androvername' : androidversionname
165         }
```

```
man_an_dic = {
    'manifest_anal' : ret_value,
    'exported_act' : exported,
    'exported_cnt' : exp_count,
    'permissons' : format_permissions(man_data_dic['perm']),
    'cnt_act' : len(man_data_dic['activities']),
    'cnt_pro' : len(man_data_dic['providers']),
    'cnt_ser' : len(man_data_dic['services']),
    'cnt_bro' : len(man_data_dic['receivers'])
}
```

ManifestAnalysis 主要功能是对 AndroidManifest.xml 进行解析，提取其中 permission、granturipermissions、application、activties、services、intents、actions 等，分析所有权限并对权限进行分级，包含正常、危险、签名、系统四个类别。对各属性配置进行检查，看是否存在不安全的配置，如 allowBackup、debuggable、exported 等属性设置。详细代码功能可见 manifest_analysis.py 程序文件。

```
194         for permission in permissions:
195             if permission.getAttribute("android:protectionLevel"):
196                 protectionlevel = permission.getAttribute(
197                     "android:protectionLevel")
198                 if protectionlevel == "0x00000000":
199                     protectionlevel = "normal"
200                 elif protectionlevel == "0x00000001":
201                     protectionlevel = "dangerous"
202                 elif protectionlevel == "0x00000002":
203                     protectionlevel = "signature"
204                 elif protectionlevel == "0x00000003":
205                     protectionlevel = "signatureOrSystem"
206 
207                 permission_dict[permission.getAttribute(
208                     "android:name")] = protectionlevel
209             elif permission.getAttribute("android:name"):
210                 permission_dict[permission.getAttribute(
211                     "android:name")] = "normal"
212 
213         # APPLICATIONS
214         for application in applications:
215 
216             if application.getAttribute("android:debuggable") == "true":
217                 ret_value = (
218                     ret_value + (
219                         '<tr><td>Debug Enabled For App <br>[android:debuggable=true]</td><td>'
220                         '<span class="label label-danger">high</span></td><td>Debugging was enable
221                         ' on the app which makes it easier for reverse engineers to hook a debugge
222                         ' to it. This allows dumping a stack trace and accessing debugging helper
223                         'classes.</td></tr>'
224                     )
225                 )
226             if application.getAttribute("android:allowBackup") == "true":
227                 ret_value = (
228                     ret_value+ (
229                         '<tr><td>Application Data can be Backed up<br>[android:allowBackup=true]'
230                         '</td><td><span class="label label-warning">medium</span></td><td>This fla
231                         ' allows anyone to backup your application data via adb. It allows users '
232                         'who have enabled USB debugging to copy application data off of the '
233                         'device.</td></tr>'
```

Cert Analysis

MobSF 证书分析功能函数在 cert_analysis.py 文件中，MobSF 首先尝试获取 Hardcoded Certificates/Keystores，然后通过 CertPrint.jar 工具解析 apk 中证书的信息，并完成证书相关问题的分析。

```
def cert_info(app_dir, tools_dir):
    """Return certificate information."""
    try:
        print "[INFO] Reading Code Signing Certificate"
        cert = os.path.join(app_dir, 'META-INF/')
        cp_path = tools_dir + 'CertPrint.jar'
        files = [f for f in os.listdir(cert) if os.path.isfile(os.path.join(cert,
        certfile = None
        dat = ''
        if "CERT.RSA" in files:
            certfile = os.path.join(cert, "CERT.RSA")
        else:
            for file_name in files:
                if file_name.lower().endswith(".rsa"):
                    certfile = os.path.join(cert, file_name)
                elif file_name.lower().endswith(".dsa"):
                    certfile = os.path.join(cert, file_name)
        if certfile:
            args = [settings.JAVA_PATH + 'java', '-jar', cp_path, certfile]
            issued = 'good'
            dat = subprocess.check_output(args)
            unicode_output = unicode(dat, encoding="utf-8", errors="replace")
            dat = escape(unicode_output).replace('\n', '</br>')
        else:
            dat = 'No Code Signing Certificate Found!'
            issued = 'missing'
        if re.findall("Issuer: CN=Android Debug|Subject: CN=Android Debug", dat):
            issued = 'bad'

        cert_dic = {
            'cert_info' : dat,
            'issued' : issued
        }
```

Code Analysis

MobSF 静态代码分析功能函数在 code_analysis.py 文件中，反编译的代码在 converter.py 中。其中使用 Dex2Jar 将 dex 转变为 jar 文件，使用 Dex2Smali 将 dex 转变为 smali 代码，使用 jd-core.jar、cfr_0_115.jar、procyon-decompiler-0.5.30.jar 将 jar 包转为为可读的 java 代码。

```
132                 dex_2_jar(app_dic['app_path'], app_dic['app_dir'], app_dic['tools_dir'])
133                 dex_2_smali(app_dic['app_dir'], app_dic['tools_dir'])
134                 jar_2_java(app_dic['app_dir'], app_dic['tools_dir'])
135
136                 code_an_dic = code_analysis(
137                     app_dic['app_dir'],
138                     app_dic['md5'],
139                     man_an_dic['permissons'],
140                     "apk"
141                 )
```

源代码分析部分主要利用正则表达式对 java 源码进行匹配来实现的。主要通过匹配常见方法中的关键词来提取源码中用到的方法。

通过匹配敏感关键词来提取账号密码等信息:

```
    code['d_ssl'].append(jfile_path.replace(java_src, ''))
if (
        'password = "' in dat.lower() or
        'secret = "' in dat.lower() or
        'username = "' in dat.lower() or
        'key = "' in dat.lower()
):
```

通过匹配常见 API 字符串来判定是否有调用这些 API:

```
#API Check

if re.findall(r"System.loadLibrary\(|System.load\(", dat):
    native = True
if (
        re.findall(
            (
                r'dalvik.system.DexClassLoader|java.security.ClassLoader|'
                r'java.net.URLClassLoader|java.security.SecureClassLoader'
            ),
            dat
        )
):
    dynamic = True
if (
        re.findall(
            'java.lang.reflect.Method|java.lang.reflect.Field|Class.forName',
            dat
        )
):
    reflect = True
if re.findall('javax.crypto|kalium.crypto|bouncycastle.crypto', dat):
    crypto = True
    code['crypto'].append(jfile_path.replace(java_src, ''))
if 'utils.AESObfuscator' in dat and 'getObfuscator' in dat:
    code['obf'].append(jfile_path.replace(java_src, ''))
    obfus = True

if 'getRuntime().exec(' in dat and 'getRuntime(' in dat:
    code['exec'].append(jfile_path.replace(java_src, ''))
```

要检测的 api 列表(部分)及对应的安全问题:

```
'd_sensitive' :
    (
        'Files may contain hardcoded sensitive informations like '
        'usernames, passwords, keys etc.'
    ),
'd_ssl':
    (
        'Insecure Implementation of SSL. Trusting all the certificates or accepting '
        'self signed certificates is a critical Security Hole. This application is '
        'vulnerable to MITM attacks'
    ),
'd_sqlite':
    (
        'App uses SQLite Database and execute raw SQL query. Untrusted user input in '
        'raw SQL queries can cause SQL Injection. Also sensitive information should be '
        'encrypted and written to the database.'
    ),
'd_con_world_readable':
    (
        'The file is World Readable. Any App can read from the file'
    ),
```

通过正则匹配 URL 的格式来提取源码中的 URL:

```
#URLs My Custom regex
pattern = re.compile(
    (
        ur'((?:https?://|s?ftps?://|file://|javascript:|data:|www\d{0,3}[.])'
        ur'[\w().=/;,#:@?&~*+!$%\'{}-]+)'
    ),
    re.UNICODE
)
```

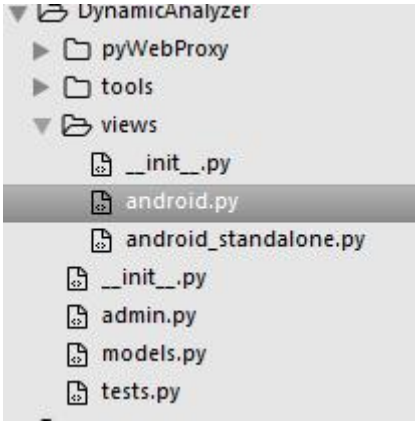
通过正则匹配 Email 的格式来提取源码中的 Email:

```
# Email Etraction Regex
regex = re.compile(r'[\w.-]+@[\w-]+\.[\w.]+')
eflag = 0
for email in regex.findall(dat.lower()):
    if (email not in emails) and (not email.startswith('//')):
        emails.append(email)
        eflag = 1
```

动态分析实现原理

MobSF 同时还支持对安卓程序的动态分析，使用 virtualbox 运行 app，并使用 virtualbox 提供的接口完成代理设置、虚拟机的操作，利用 adb 命令安装运行 app，并获取 app 的运行相关信息，进行分析。

其动态分析主要目录结构如下图：



主要功能代码在 DynamicAnalyzer/views 目录下的 android.py，根据其前端展示页面可知其主要功能有：

- Environment Created
- Start / Stop Screen
- Install / Remove MobSF RootCA
- Start Exported Activity Tester
- Start Activity Tester
- Take a Screenshot
- Finish

虚拟机运行环境参数、Web 代理设置、模拟设备设置在 settings 文件中

```
#===============DEVICE SETTINGS================
REAL_DEVICE = False
DEVICE_IP = '192.168.1.18'
DEVICE_ADB_PORT = 5555
DEVICE_TIMEOUT = 300
#================================================
#================VM SETTINGS ====================

# VM UUID
UUID = '408e1874-759f-4417-9453-53ef21dc2ade'
# Snapshot UUID
SUUID = '5c9deb28-def6-49c0-9233-b5e03edd85c6'
# IP of the MobSF VM
VM_IP = '192.168.56.101'
VM_ADB_PORT = 5555
VM_TIMEOUT = 100
#================================================
#================HOST/PROXY SETTINGS ===========

PROXY_IP = '192.168.56.1'  # Host/Server/Proxy IP
PORT = 1337  # Proxy Port
ROOT_CA = '0025aabb.0'
SCREEN_IP = PROXY_IP  # ScreenCast IP
SCREEN_PORT = 9339  # ScreenCast Port

#================================================
```

Environment Created

该功能主要由 GetEnv 函数实现 Web 代理设置，adb 命令接口实现 app 的安装、运行。MobSF 与虚拟运行环境连接后，即开始动态分析流程。

```
746 def InstallRun(TOOLSDIR, APKPATH, PACKAGE, LAUNCH, isACT):
747     print "\n[INFO] Starting App for Dynamic Analysis"
748     try:
749         adb = getADB(TOOLSDIR)
750         print "\n[INFO] Installing APK"
751         subprocess.call([adb, "-s", getIdentifier(), "install", "-r", APKPATH])
752         if isACT:
753             runApp = PACKAGE + "/" + LAUNCH
754             print "\n[INFO] Launching APK Main Activity"
755             subprocess.call([adb, "-s", getIdentifier(),
756                              "shell", "am", "start", "-n", runApp])
757         else:
758             print "\n[INFO] App Doesn't have a Main Activity"
759             # Handle Service or Give Choice to Select in Future.
760             pass
761         print "[INFO] Testing Environment is Ready!"
762     except:
763         PrintException("[ERROR]  Starting App for Dynamic Analysis")
764
```

Start / Stop Screen

MobSF 中提供实时操作功能，其实现主要利用屏幕录制软件 screencast 提供的服务，

其实现代码如下：

```
def ScreenCast(request):
    print "\n[INFO] Invoking ScreenCast Service in VM/Device"
    try:
        global tcp_server_mode
        data = {}
        if (request.method == 'POST'):
            mode = request.POST['mode']
            TOOLSDIR = os.path.join(
                settings.BASE_DIR, 'DynamicAnalyzer/tools/')  # TOOLS DIR
            adb = getADB(TOOLSDIR)
            IP = settings.SCREEN_IP
            PORT = str(settings.SCREEN_PORT)
            if mode == "on":
                args = [adb, "-s", getIdentifier(), "shell", "am", "startservice",
                        "-a", IP + ":" + PORT, "opensecurity.screencast/.StartScreenCast"]
                data = {'status': 'on'}
                tcp_server_mode = "on"
            elif mode == "off":
                args = [adb, "-s", getIdentifier(), "shell", "am",
                        "force-stop", "opensecurity.screencast"]
                data = {'status': 'off'}
                tcp_server_mode = "off"
            if (mode in ["on", "off"]):
                try:
                    subprocess.call(args)
                    t = threading.Thread(target=ScreenCastService)
                    t.setDaemon(True)
                    t.start()
                except:
                    PrintException("[ERROR] Casting Screen")
                    data = {'status': 'error'}
                    return HttpResponse(json.dumps(data), content_type='application/json')
            else:
                data = {'status': 'failed'}
```

Install / Remove MobSF RootCA

```
300  def MobSFCA(request):
301      try:
302          if request.method == 'POST':
303              data = {}
304              act = request.POST['action']
305              TOOLSDIR = os.path.join(
306                  settings.BASE_DIR, 'DynamicAnalyzer/tools/')  # TOOLS DIR
307              ROOTCA = os.path.join(
308                  settings.BASE_DIR, 'DynamicAnalyzer/pyWebProxy/ca.crt')
309              adb = getADB(TOOLSDIR)
310              if act == "install":
311                  print "\n[INFO] Installing MobSF RootCA"
312                  subprocess.call([adb, "-s", getIdentifier(), "push",
313                                   ROOTCA, "/data/local/tmp/" + settings.ROOT_CA])
314                  subprocess.call([adb, "-s", getIdentifier(), "shell", "su", "-c", "cp", "/data/l
315                                   settings.ROOT_CA, "/system/etc/security/cacerts/" + settings.RO
316                  subprocess.call([adb, "-s", getIdentifier(), "shell", "su", "-c",
317                                   "chmod", "644", "/system/etc/security/cacerts/" + settings.ROOT_
318                  subprocess.call([adb, "-s", getIdentifier(), "shell",
319                                   "rm", "/data/local/tmp/" + settings.ROOT_CA])
320                  data = {'ca': 'installed'}
321              elif act == "remove":
322                  print "\n[INFO] Removing MobSF RootCA"
323                  subprocess.call([adb, "-s", getIdentifier(), "shell", "su", "-c",
324                                   "rm", "/system/etc/security/cacerts/" + settings.ROOT_CA])
325                  data = {'ca': 'removed'}
326              return HttpResponse(json.dumps(data), content_type='application/json')
327          else:
328              return HttpResponseRedirect('/error/')
329      except:
330          PrintException("[ERROR] MobSF RootCA Handler")
331          return HttpResponseRedirect('/error/')
332
```

Start /Stop Exported Activity Tester

这部分主要是想尽量多的触发样本中所有行为，MobSF 的做法是：遍历 AndroidManifest.xml 中的所有 Exported Activity，并利用 am start 来依次启动，以方便 xposed 能获取到更多的日志。

```
def ExportedActivityTester(request):
    print "\n[INFO] Exported Activity Tester"
    try:
        MD5 = request.POST['md5']
        PKG = request.POST['pkg']
        m = re.match('^[0-9a-f]{32}$', MD5)
        if m:
            if re.findall(";|\$\(|\|\||&&", PKG):
                print "[ATTACK] Possible RCE"
                return HttpResponseRedirect('/error/')
            if request.method == 'POST':
                DIR = settings.BASE_DIR
                APP_DIR = os.path.join(settings.UPLD_DIR, MD5 + '/')
                TOOLS_DIR = os.path.join(
                    DIR, 'DynamicAnalyzer/tools/')  # TOOLS DIR
                SCRDIR = os.path.join(APP_DIR, 'screenshots-apk/')
                data = {}
                adb = getADB(TOOLS_DIR)

                DB = StaticAnalyzerAndroid.objects.filter(MD5=MD5)
                if DB.exists():
                    print "\n[INFO] Fetching Exported Activity List from DB"
                    EXPORTED_ACT = python_list(DB[0].EXPORTED_ACT)
                    if EXPORTED_ACT:
                        n = 0
                        print "\n[INFO] Starting Exported Activity Tester..."
                        print "\n[INFO] " + str(len(EXPORTED_ACT)) + " Exported Activities Identi
                        for line in EXPORTED_ACT:
                            try:
                                n += 1
                                print "\n[INFO] Launching Exported Activity - " + str(n) + ". " +
                                subprocess.call(
                                    [adb, "-s", getIdentifier(), "shell", "am", "start", "-n", PK
                                Wait(4)
                                subprocess.call(
                                    [adb, "-s", getIdentifier(), "shell", "screencap", "-p", "/da
                                #? get appended from Air :-() if activity names are used
                                subprocess.call(
                                    [adb, "-s", getIdentifier(), "pull", "/data/local/screen.png"
                                print "\n[INFO] Activity Screenshot Taken"
                                subprocess.call(
                                    [adb, "-s", getIdentifier(), "shell", "am", "force-stop", PKG
```

其主要流程是:

1) 获取静态分析得到的 exported activity 列表
2) 遍历 activity，并用 adb -s IP:PORT shell am start -n PACKAGE/ACTIVITY 启动相应的 activity
3) 获取当前 activity 运行时的屏幕截图 adb -s IP:PORT shell screencap -p /data/local/screen.png
4) 保存该截屏
5) 强制关闭该应用 adb -s IP:PORT shell am force-stop PACKAGE

Start / Stop Activity Tester

与 Exported Activity 不同的是，这个测试将会遍历 AndroidManifest.xml 中所有 Activity，而不单单是 Exported。其流程与处理 Exported Activity 基本相同。

Take a Screenshot

截取屏幕并将图片保存在本地，代码如下:

```
def TakeScreenShot(request):
    print "\n[INFO] Taking Screenshot"
    try:
        if request.method == 'POST':
            MD5 = request.POST['md5']
            m = re.match('^[0-9a-f]{32}$', MD5)
            if m:
                data = {}
                r = random.randint(1, 1000000)
                DIR = settings.BASE_DIR
                # make sure that list only png from this directory
                SCRDIR = os.path.join(
                    settings.UPLD_DIR, MD5 + '/screenshots-apk/')
                TOOLSDIR = os.path.join(
                    DIR, 'DynamicAnalyzer/tools/')  # TOOLS DIR
                adb = getADB(TOOLSDIR)
                subprocess.call([adb, "-s", getIdentifier(), "shell",
                                 "screencap", "-p", "/data/local/screen.png"])
                subprocess.call([adb, "-s", getIdentifier(), "pull",
                                 "/data/local/screen.png", SCRDIR + "screenshot-" + str(r) + ".pn
                print "\n[INFO] Screenshot Taken"
                data = {'screenshot': 'yes'}
                return HttpResponse(json.dumps(data), content_type='application/json')
            else:
                return HttpResponseRedirect('/error/')
        else:
            return HttpResponseRedirect('/error/')
    except:
        PrintException("[ERROR] Taking Screenshot")
        return HttpResponseRedirect('/error/')
```

Finish

在 FinalTest 函数中 MobSF 会将程序运行过程中的所有 dalvikvm 的 Warning 和 ActivityManager 的 Information 收集起来。

```
os.system(adb + ' -s ' + getIdentifier() +
          ' logcat -d dalvikvm:W ActivityManager:I > "' + APKDIR + 'logcat.txt"')
print "\n[INFO] Downloading Logcat logs"
#os.system(adb+' -s '+getIdentifier()+' logcat -d Xposed:I *:S > "'+APKDIR + 'x_logcat.txt"')
subprocess.call([adb, "-s", getIdentifier(), "pull",
                 "/data/data/de.robv.android.xposed.installer/log/error.log", APKDIR + "x_logcat.txt"])

print "\n[INFO] Downloading Droidmon API Monitor Logcat logs"
# Can't RCE
os.system(adb + ' -s ' + getIdentifier() +
          ' shell dumpsys > "' + APKDIR + 'dump.txt"')
print "\n[INFO] Downloading Dumpsys logs"
```

MobSF 对日志的分析功能主要在 APIAnalysis 和 RunAnalysis 两个函数中，和静态日志分析一样，动态日志分析也是以正则匹配为主，APIAnalysis 主要对 x_logcat.txt 中 Droidmon.apk 产生的日志进行处理，主要进行 API 调用分析，包括 API 的 class、参数、返回值等，对需要监控的 api 函数在 DynamicAnalyzer\tools\onDevice 目录下的 hooks.json 文件中。包含监控函数详细的类名、方法名称等。

```
MTD = str(APIs["method"])
CLS = str(APIs["class"])
# print "Called Class: " + CLS
# print "Called Method: " + MTD
if APIs.get('return'):
    RET = str(APIs["return"])
    # print "Return Data: " + RET
else:
    # print "No Return Data"
    RET = "No Return Data"
if APIs.get('args'):
    ARGS = str(APIs["args"])
    # print "Passed Arguments" + ARGS
else:
    # print "No Arguments Passed"
    ARGS = "No Arguments Passed"
# XSS Safe
D = "</br>METHOD: " + \
    escape(MTD) + "</br>ARGUMENTS: " + escape(ARGS) + \
    "</br>RETURN DATA: " + escape(RET)

if re.findall("android.util.Base64", CLS):
    # Base64 Decode
    if ("decode" in MTD):
        args_list = python_list(ARGS)
        if isBase64(args_list[0]):
            D += '</br><span class="label label-info">Decoded String:</span> ' + \
                escape(base64.b64decode(args_list[0]))
    API_BASE64.append(D)
if re.findall('libcore.io|android.app.SharedPreferencesImpl$EditorImpl', CLS):
    API_FILEIO.append(D)
if re.findall('java.lang.reflect', CLS):
    API_RELECT.append(D)
if re.findall('android.content.ContentResolver|android.location.Location|android.media.AudioR
    API_SYSPROP.append(D)
if re.findall('android.app.Activity|android.app.ContextImpl|android.app.ActivityThread', CLS)
    API_BINDER.append(D)
if re.findall('javax.crypto.spec.SecretKeySpec|javax.crypto.Cipher|javax.crypto.Mac', CLS):
    API_CRYPTO.append(D)
if re.findall('android.accounts.AccountManager|android.app.ApplicationPackageManager|android.
```

RunAnalysis 函数主要处理样本运行后留下的 WebTraffic.txt、logcat.txt、x_logcat.txt 日志文件。

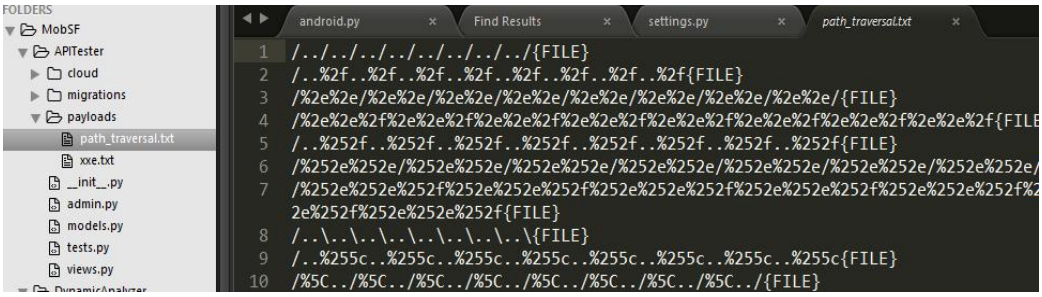
```
def RunAnalysis(APKDIR, MD5, PACKAGE):
    print "\n[INFO] Dynamic File Analysis"
    Web = os.path.join(APKDIR, 'WebTraffic.txt')
    Logcat = os.path.join(APKDIR, 'logcat.txt')
    xLogcat = os.path.join(APKDIR, 'x_logcat.txt')
    traffic = ''
    wb = ''
    xlg = ''
    DOMAINS = {}
    logcat_data = []
    CLIPBOARD = []
    CLIP_TAG = "I/CLIPDUMP-INFO-LOG"
    try:
        with io.open(Web, mode='r', encoding="utf8", errors="ignore") as f:
            wb = f.read()
    except:
        pass

    with io.open(Logcat, mode='r', encoding="utf8", errors="ignore") as f:
        logcat_data = f.readlines()
        traffic = ''.join(logcat_data)
    with io.open(xLogcat, mode='r', encoding="utf8", errors="ignore") as f:
        xlg = f.read()
    traffic = wb + traffic + xlg
    for log_line in logcat_data:
        if log_line.startswith(CLIP_TAG):
            CLIPBOARD.append(log_line.replace(CLIP_TAG, "Process ID "))
    URLS = []
    # URLs My Custom regex
    p = re.compile(ur'((?:https?://|s?ftps?://|file://|javascript:|data:|www\d{0,3}[.])[\w().
    urllist = re.findall(p, traffic.lower())
    # Domain Extraction and Malware Check
    print "[INFO] Performing Malware Check on extracted Domains"
    DOMAINS = MalwareCheck(urllist)
```

在 RunAnalysis 函数中，MobSF 首先用正则匹配出所有可能的 url，然后再一一对 url 进行相应分析。

API Fuzzer

MobSF 框架中 API Fuzzer 模块主要针对 URL 进行扫描测试，目前支持 SSRF、XXE、Path Traversal 等漏洞的扫描，XXE 及 Path Traversal 测试 Payloads 在 APITester\payloads\ 路径下：



通过在 setting 文件中预定义设置的特征来匹配检测结果，同时 MobSF 也支持与云端的连接,从而进一步准确和全面的检测安全漏洞。

```
#==============RESPONSE VALIDATION==============
XXE_VALIDATE_STRING = "m0bsfxx3"
#================================================

#=========Path Traversal - API Testing==========
CHECK_FILE = "/etc/passwd"
RESPONSE_REGEX = "root:|nobody:"
#================================================

#=========Rate Limit Check - API Testing========
RATE_REGISTER = 20
RATE_LOGIN = 20
#================================================

#===============MobSF Cloud Settings============
CLOUD_SERVER = 'http://opensecurity.in:8080'
'''
This server validates SSRF and XXE during Web API Testing
See the source code of the cloud server from APITester/cloud/cloud_server.py
You can also host the cloud server. Host it on a public IP and point CLOUD_SERVER to that IP
```

总结

通过对 MobSF 源代码的分析可以了解 MobSF 的基本工作原理以及流程。

静态分析

静态分析时，MobSF 主要使用了现有的 dex2jar、dex2smali、jar2java、AXMLPrinter、CertPrint 等工具。其主要完成了两项工作：解析 AndroidManifest.xml 得到了应用程序的各类相关信息、对 apk 进行反编译得到 java 代码，而后利用正则匹配找出该 app 包含的 API 函数、URL、邮箱集帐号密码等敏感信息。

动态分析

而动态分析部分，MobSF 主要利用到了 Xposed 框架、Droidmon 实现对应用程序调用 API 的情况进行监控，并且详细列出了需要分析的 API 列表。同时，MobSF 还使用了 ScreenCast 结合 adb shell input 完成对手机的远程控制功能。动态分析主要操作有：

- ✧ 利用 webproxy 实现代理进而拦截样本流量。
- ✧ 安装证书以便拦截 https 流量。
- ✧ 遍历所有 activity，尽量多的获取各 activity 运行得到的日志。
- ✧ 利用正则匹配出 API 及参数和返回值。
- ✧ 尽可能多的匹配出 URL 信息，对 URL 进行后续分析及恶意 URL 查杀。

参考

[1] http://www.freebuf.com/sectool/99475.html
[2] https://github.com/MobSF/Mobile-Security-Framework-MobSF
[3] http://www.droidsec.cn/移动 app 漏洞自动化检测平台建设/
[4] https://github.com/Cryin/Paper