33

未整理

一 混淆

对核心函数必须做强混淆，比如涉及到核心数据的解析，模型的使用这些。
工具：obfuscator
我已经对这个工具做了简单的测试，可用，效果比较好。
工具自带了三个混淆的方式，有时间的话也可以增加自定义的混淆方式。
使用方式：
首先编译：https://github.com/obfuscator-llvm/obfuscator/wiki/Installation
测试的**NDK版本：14.1.3816874**
在NDK的toolchains目录新建 `obfuscator-llvm-3.6` 目录，将编译出来工具链的 `bin` 和 `lib` 复制到新建的 `obfuscator-llvm-3.6` 目录。在 `$NDK/build/core/toolchains` 下，新建 `arm-linux-androideabi-clang3.6-obfuscator` 目录，复制 `arm-linux-androideabi-4.9` 目录中的 `config.mk` 和 `setup.mk` 到新建目录，并将 `setup.mk` 中的

```
LLVM_TOOLCHAIN_PREBUILT_ROOT := $(call get-toolchain-root,llvm)
```

修改成：

```
LLVM_TOOLCHAIN_PREBUILT_ROOT := $(call get-toolchain-root,obfusc
ator-llvm-3.6)
```

使用时需要在Android.mk增加相应的混淆选项：

```
LOCAL_CFLAGS := -mllvm -sub -mllvm -fla -mllvm -bcf
```

同时在Application.mk中指定工具链：

```
NDK_TOOLCHAIN_VERSION :=  clang3.6-obfuscator
```

工具没有单独对某个函数混淆的方法，需要你再研究下（把需要混淆的函数独立出来编译，后面再做链接？）。整个混淆可能导致运行时性能消耗会比较大，你可以做一些测试。
可以和Safengine 结合使用，我想到的是把我们混淆过部分的先编译成库文件，然后这边静态链接过来。

二 动态生成

各种检验函数，比如license的验证函数，反调试函数等关键函数，必须要使用运行时动态修改的方法，即调用关键函数前解密，调用后再次加密。举个例子，so中的 `check` 函数可以是这样：

```
void check(){
    retrun;
}
```

然后在运行时动态修改回 `check` 函数的真实逻辑。

```
void check(){
    if(invalid)
        exit;
    else
        return;
}
```

以上两种方法可以结合使用，看使用场景吧。

重点：**各类检验函数的逻辑应该动态生成**

三 license

license的验证逻辑我不大了解，之前师兄发给我那个sdk，有好几个条件需要做验证，但是由于没有对验证函数做保护，所以再复杂也没有什么卵用。所以验证license这个逻辑，必须动态生成。
那个SDK，会把license压进一个全局的数据结构（下图中的 `LicenseChain`），然后每次函数调用前都会检测这个数据结构时候是否存在可用的license。这个逻辑的问题是，假如License文件结构是对的，可以被正常添加进 `LicenseChain`，但是包名，时间不对，虽然有校验，还是轻易就可以通过修改so里面的汇编指令绕过。

四 其他

1. **加反调试**，这个网上资料比较多，要使用多种方法来检测，请参考 http://www.droidsec.cn/anti-debugging-skills-in-apk/和 http://paper.seebug.org/204/

2. **一些验证函数，可以单启一个进程或者线程来执行，比如反调试函数等等。**
3. **需要对加完壳（UPX）的so做section字段修改，使得IDA等工具不能识别。**

五 加壳

还有就是考虑单独对so加壳，这块好像都是付费服务。也可以自己写，不过这块简单的容易被破解，难度大的开发成本又太高。当然了，没有绝对安全的系统，再厉害的壳还是给可以被脱掉的。

这里提供一个我最近做的CTF题的思路，分为两个so，一个jni的so还有一个核心算法的so，核心算法的so是在内存中动态解密生成，由jni的so自己加载，然后jni的so动态解密部分函数，由这些动态生成的函数来调用核心算法so里的函数。

反正先做好混淆吧，我觉得加壳不是做保护的重点，重点在第一步的强混淆，其他都是尽可能提高逆向的成本。

附：不好的license验证例子

首先把license放进了 `LicenseChain` 中， `LABEL_115` 表示license正常，那么我在第二个红框处，修改跳转条件，使用demo的license即可绕过的验证。

```
285     std::string::string(&v77, (const unsigned __int8 *)v46, &v75);
286     v40 = protector::LicenseChain::AddLicense((protector::LicenseChain *)v16, &v77);
287     if ( v77._M_dataplus._M_p - 12 != v21 )
288     {
289       v66 = (std::basic_string<char,std::char_traits<char>,std::allocator<char> >::_Rep *
290       v67 = v40;
291       v64 = _sync_fetch_and_add_4((int *)v77._M_dataplus._M_p - 1, -1);
292       v40 = v67;
293       if ( v64 <= 0 )
294       {
295         std::string::_Rep::_M_destroy(v66, (const std::allocator<char> *const )&v76);
296         v40 = v67;
297       }
298     }
299     if ( !v40 )
300     {
301       free(v70);
302       goto LABEL_84;
303     }
304     ++v44;
305 LABEL_46:
306     if ( !v43 )
307       goto LABEL_47;
```

```
LABEL_84:
    protector::LicenseChain::PopLicenses((protector::LicenseChain *)v16, v44);
    return -13;
  }
  if ( !protector::LicenseChain::ValidChain((protector::LicenseChain *)v16) )
    goto LABEL_115;
  if ( __sa )
  {
    if ( protector::LicenseChain::ValidChain((protector::LicenseChain *)v16) )
    {
      v72 = 8;
      goto LABEL_53;
    }
LABEL_106:
    v51 = -13;
    goto LABEL_59;
  }
  if ( !protector::LicenseChain::ValidChain((protector::LicenseChain *)v16) )
    goto LABEL_106;
  if ( !protector::LicenseChain::CheckTimelimit((protector::LicenseChain *)v16) )
  {
    v51 = -15;
    protector::LicenseChain::PopLicenses((protector::LicenseChain *)v16, v44);
    return v51;
  }
LABEL_53:
  if ( !protector::LicenseChain::CheckAppId((protector::LicenseChain *)v16) )
  {
    v51 = -14;
    protector::LicenseChain::PopLicenses((protector::LicenseChain *)v16, v44);
    return v51;
  }
  if ( !protector::LicenseChain::CheckUUID((protector::LicenseChain *)v16, v73) )
  {
    v51 = -16;
```

```
  {
LABEL_115:
    v60 = 0;
  }
  return v60;
}
```