Rooting Checker

탐지 우회 방안

2022-10-22

이성용

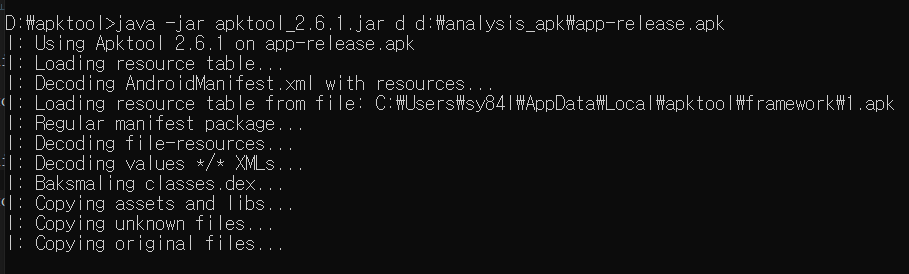
목차

1. Java Layer
   1. Decompile
   2. DetectModule\_1
   3. DetectModule\_2
   4. DetectModule\_3
   5. DetectModule\_3\_Item
   6. DetectModule\_4
   7. DetectModuleProxy
   8. ResultViewAdapter$ViewHolder
2. C Layer
   1. detect\_module\_1
   2. detect\_module\_2
   3. detect\_module\_3
   4. detect\_module\_4
   5. detect\_module\_5
   6. detect\_module\_6
   7. detect\_module\_7
   8. detect\_module\_8
   9. detect\_module\_9
3. 테스트
   1. APK 설치
   2. 적용 결과
4. 보완 방안

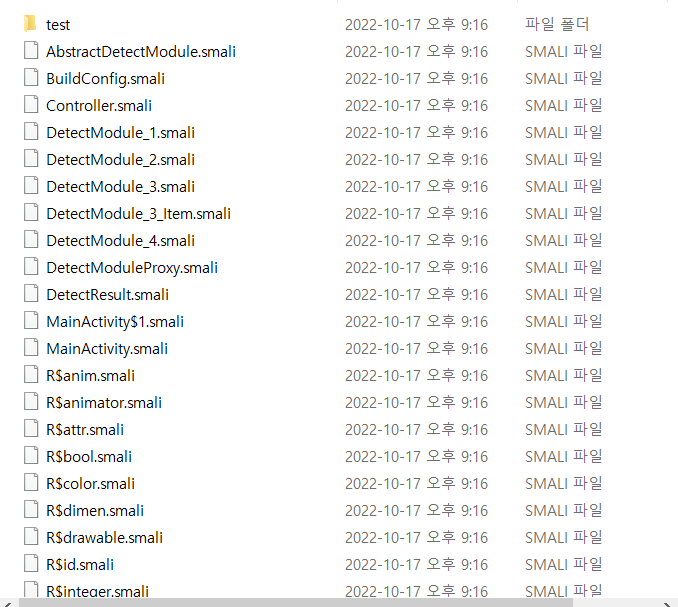
**1. Java Layer**

A. Decompile

1) Apktool을 다운로드 하고 apktool을 사용하여 apk를 decompile 수행

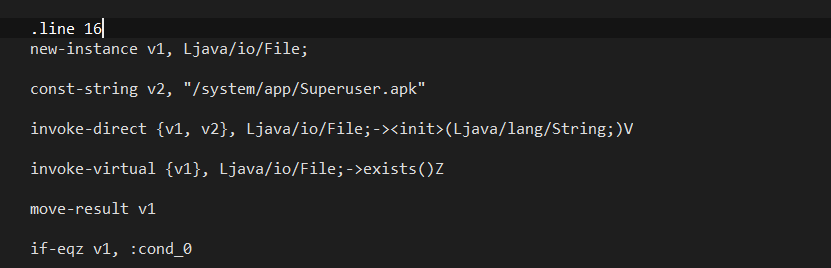


2) decompile된 파일들을 살펴보며 detect에 사용되는 파일을 유추



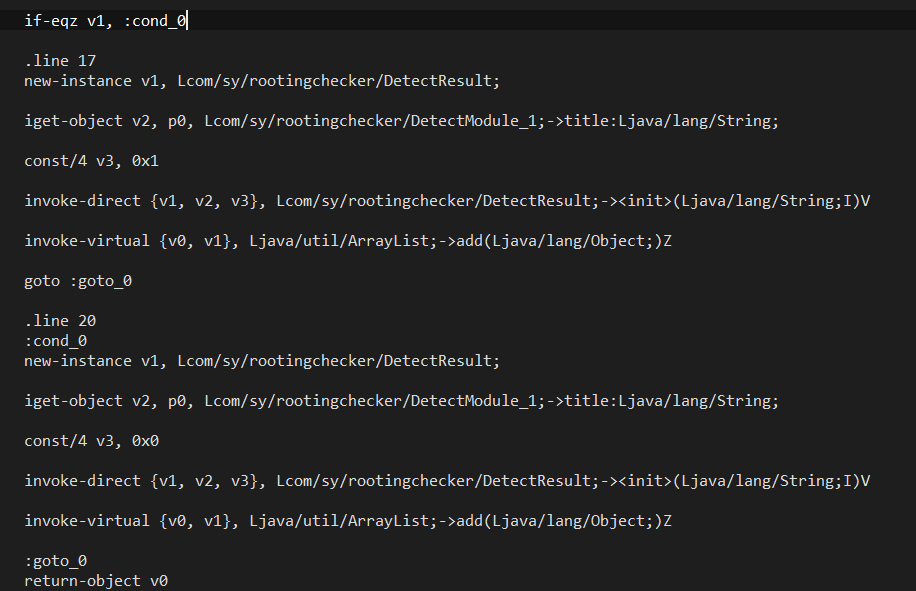
3) DetectModule\_1, DetectModule\_2, DetectModule\_3, DetectModule\_3\_Item, DetectModule\_4으로 유추

B. DetectModule\_1



DetectModule\_1.smali 을 살펴보면 “/system/app/Superuser.apk” String이 나오고 해당 String으로 File Class를 생성하고 exist()함수를 호출하여 결과값을 받는 것으로 보이는 코드가 나온다.

#방안1: “/system/app/Superuser.apk” String을 존재하지 않는 파일로 변경한다.

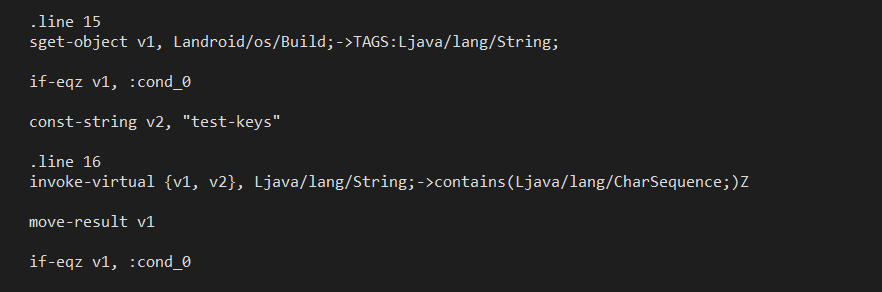


File Class의 Exist 결과값을 통해 “if-eqz v1, :cond\_0”을 통해 분기를 타고 있다.

“If-eqz v1, :cond\_0” v1이 0이면, cond\_0으로 점프 한다. v1은 File.exist의 결과값이므로 0이라면 파일이 존재하지 않는 것을 의미한다. 즉, cond\_0의 코드가 Detect 되지 않는 코드로 v3에 넣는 0값이 Not Detect을 의미한다고 볼 수 있다.

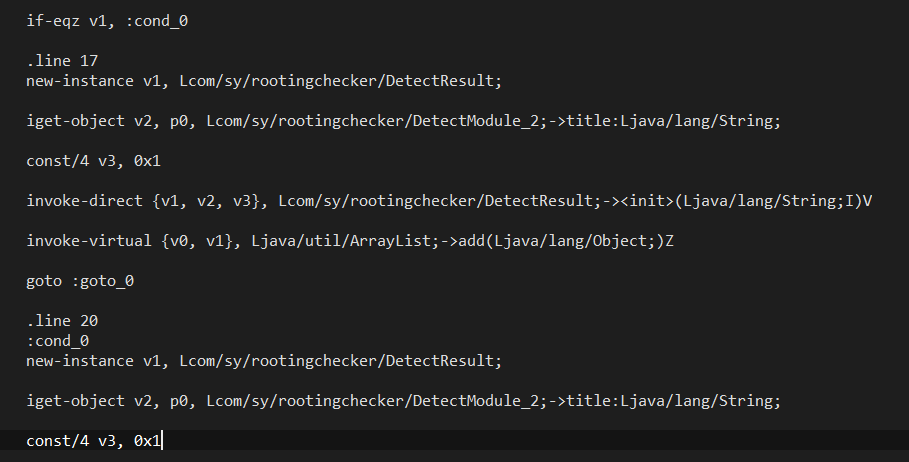
#방안2: “const/4 /3, 0x1”을 “const/4 /3, 0x0”으로 변경한다.

C. DetectModule\_2



DetectModule\_1과 유사한 패턴의 코드가 등장한다. TAGS의 값을 v1에 넣고 해당 값에 “test-keys”가 포함되어 있는지 유무로 결과를 판단하고 있는 것으로 유추된다.

#방안1: “test-keys” 문구를 다른 값으로 변경한다.

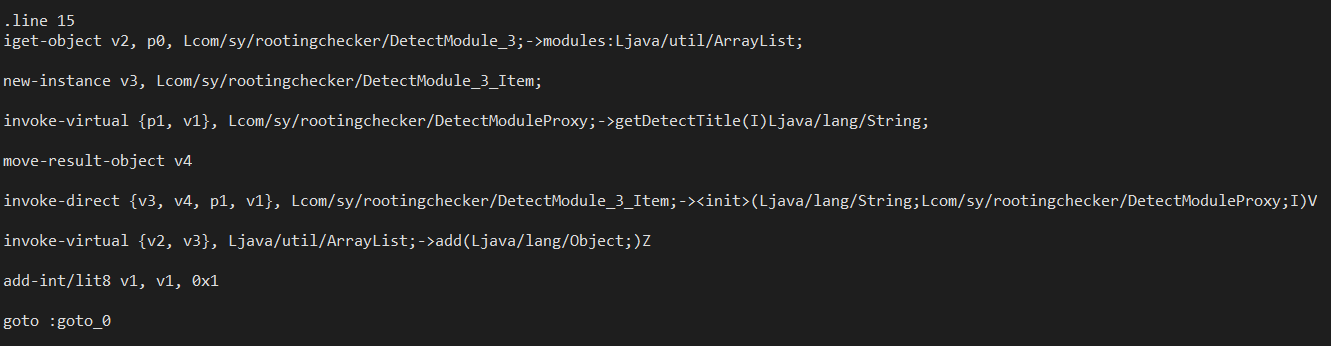


“test-keys” 포함의 결과에 따라 v3에 다른 값을 넣고 있다. Detect\_1에서 분석한 바 1이 Detect을 의미한다.

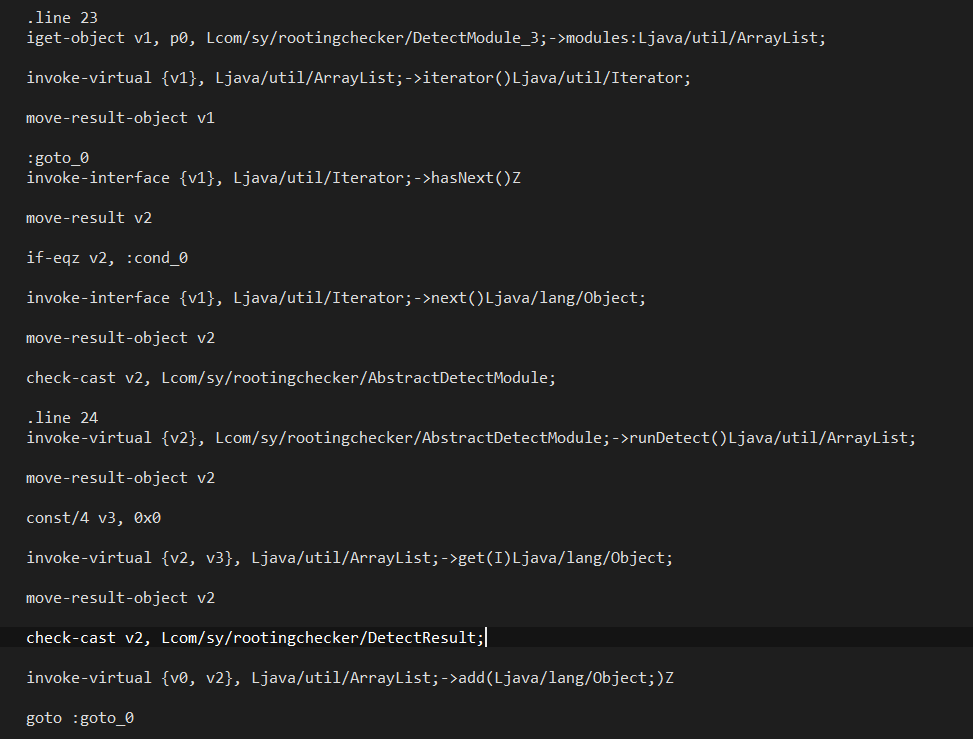
#방안2: v3의 값을 0으로 변경한다.

D. DetectModule\_3

DetectModule\_3에서는 결과 값을 제어하는 코드는 보이지 않는다.



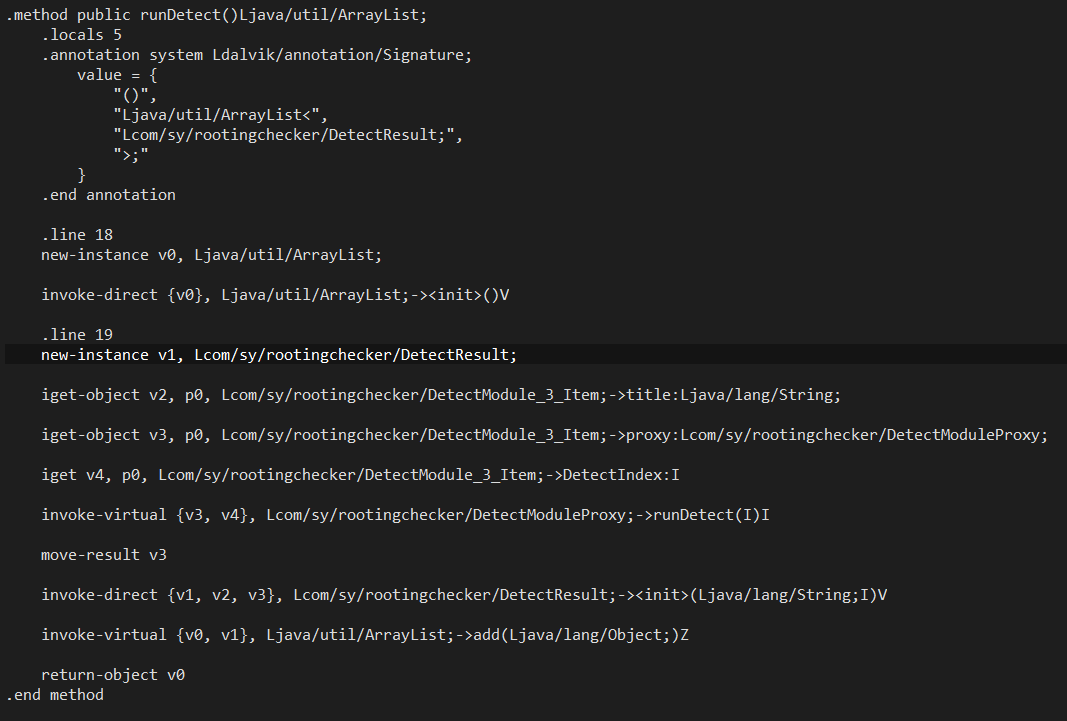
다만 DetectModule3에서 DetectModule3\_Item을 생성하여 ArrayList에 Add시키고



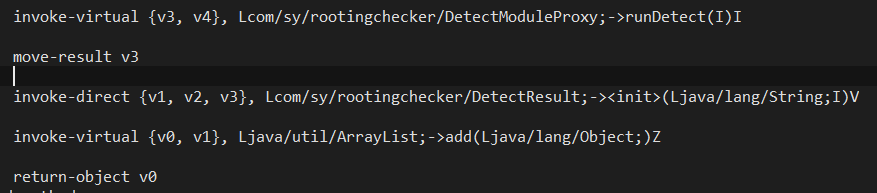
runDetect 함수에서 해당 ArrayList를 반복하면서 runDetect을 수행하는 것을 알 수 있다.

E. DetectModule\_3\_Item

DetectModule\_3\_Item에서도 결과값을 제어하는 것으로 추측되는 코드는 보이지 않는다.



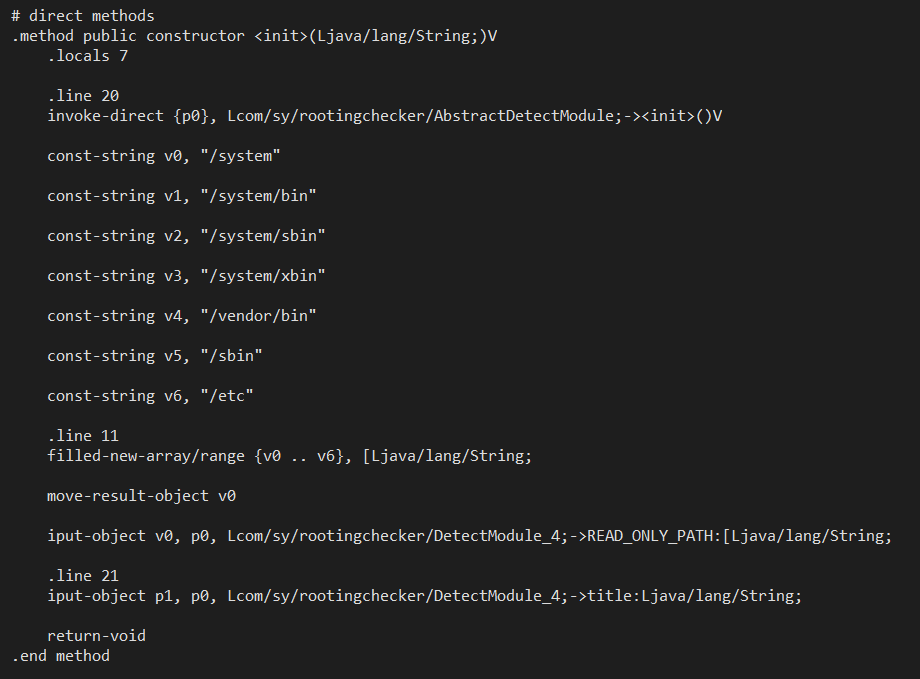
다만, runDetect함수에서 DetectModuleProxy의 runDetect을 호출하는 것을 알 수 있다.



받아온 결과값 v3을 통해 DetectResult을 생성하는 것으로 볼 수 있다. 즉, v3의 값은 0, 1 값으로 추측해볼 수 있다.

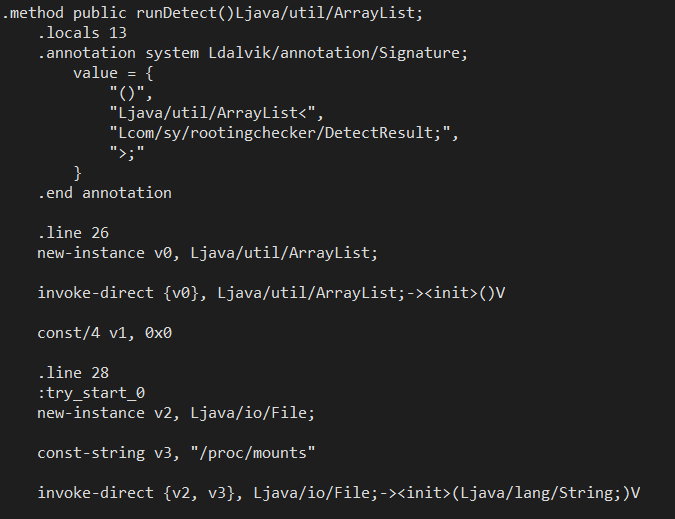
#방안1: v3에서 0값을 넣는다. “const/4 v3, 0x0”

F. DetectModule\_4



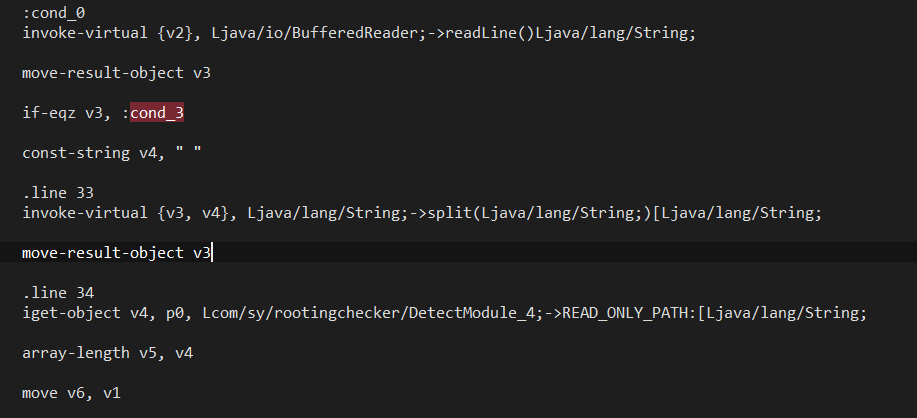
7개의 String이 정의되어 있고, 해당 String들은 READ\_ONLY\_PATH 배열임을 알 수 있다. 이를 통해 이 문자열을 비교해볼 것이라고 예측할 수 있다.

#방안1: 문자열들을 임의의 문자열로 변경한다.



runDetect함수를 보면 “proc/mounts” 파일을 Open하는 것을 볼 수 있다.

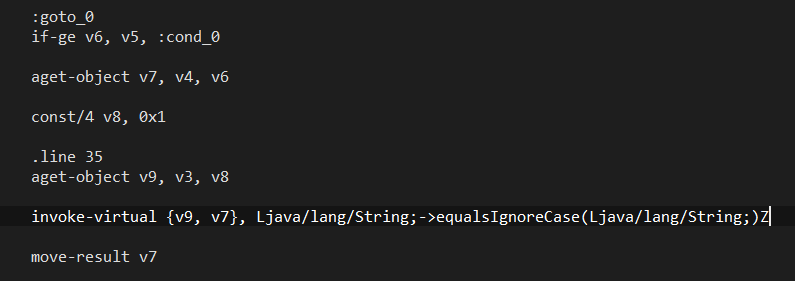
#방안2: 파일명을 변경한다.



파일로부터 readLine()을 수행하여 split한 값을 v3에 넣는다. 그리고 사전 정의된 READ\_ONLY\_PATH 배열은 v4에 넣는다.

#방안3: readLine() 수행한 값이 v3에 들어간 직후 v3에 임의의 문자열을 넣는다.

v4의 length는 v5에 넣고 v1의 값은 v6에 넣는다. V1에는 0이 들어있다.



V3: line의 split array

V4: READ\_ONLY\_PATH array

V5: READ\_ONLY\_PATH length

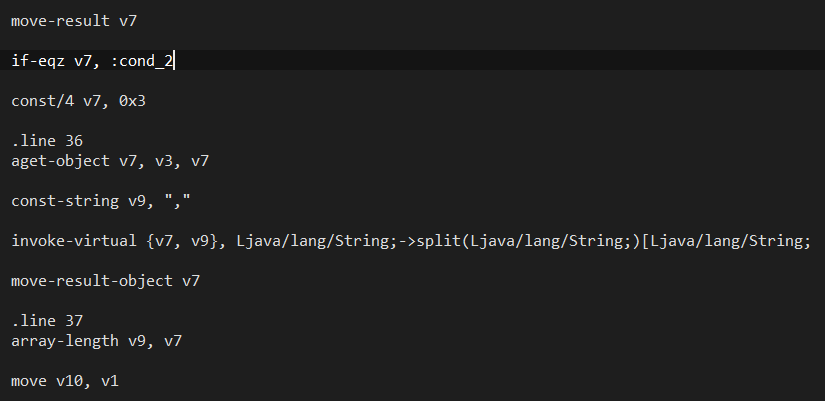
V6: 0

v7: v4의 객체

v8: 1

v9: line의 split array의 1번째

v9와 v7을 파라미터로 equalsIgnoreCase함수를 통해 비교하고 결과값을 v7에 넣는다.



v7: 3

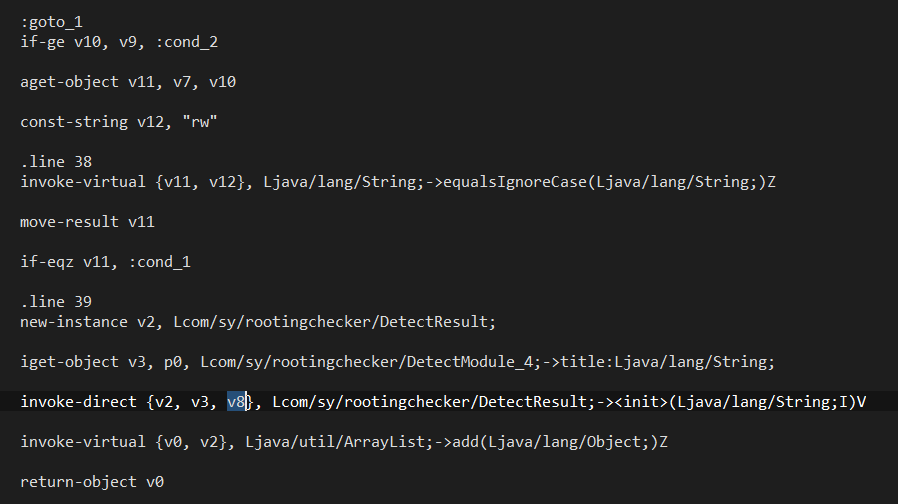
v7: line의 split array의 3번째

v7: line의 split array의 3번째의 split

v9: v7의 length

v10: 0

Read된 Line과 READ\_ONLY\_PATH의 비교 결과 값에 따라 동일하면 v7에는 line split된 3번째를 v7에 넣는다.



v11: line의 split array의 3번째의 split한 객체

v12: rw

v11의 값 중에서 rw가 있는지 확인하는 코드다.

#방안4: rw을 변경한다.

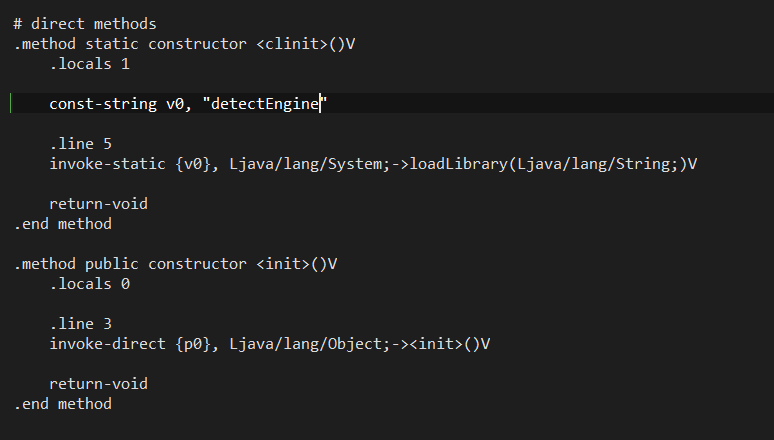
비교한 결과값은 v11에 들어가고, rw가 있다면 DetectResult을 생성하는 것을 볼 수 있다.

이때 넣는 값은 v8 즉, 1이다.

#방안5: v8의 값을 0으로 변경하거나 v8대신 0을 넣는다.

G. DetectModuleProxy

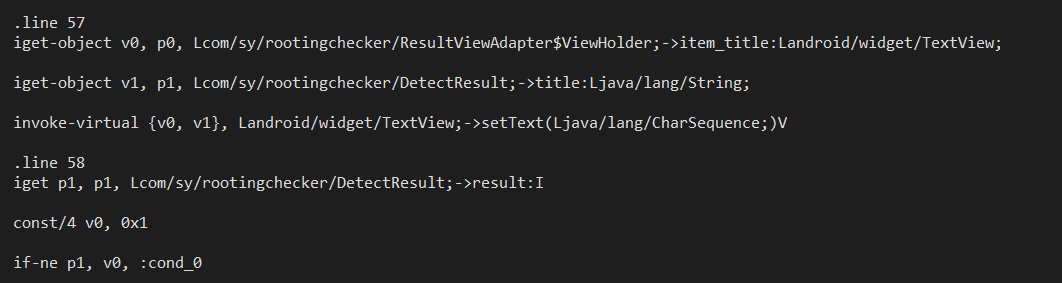
탐지 결과값을 제어하는 코드는 없다. 다만 Native Library를 Load하는 것을 볼 수 있다.



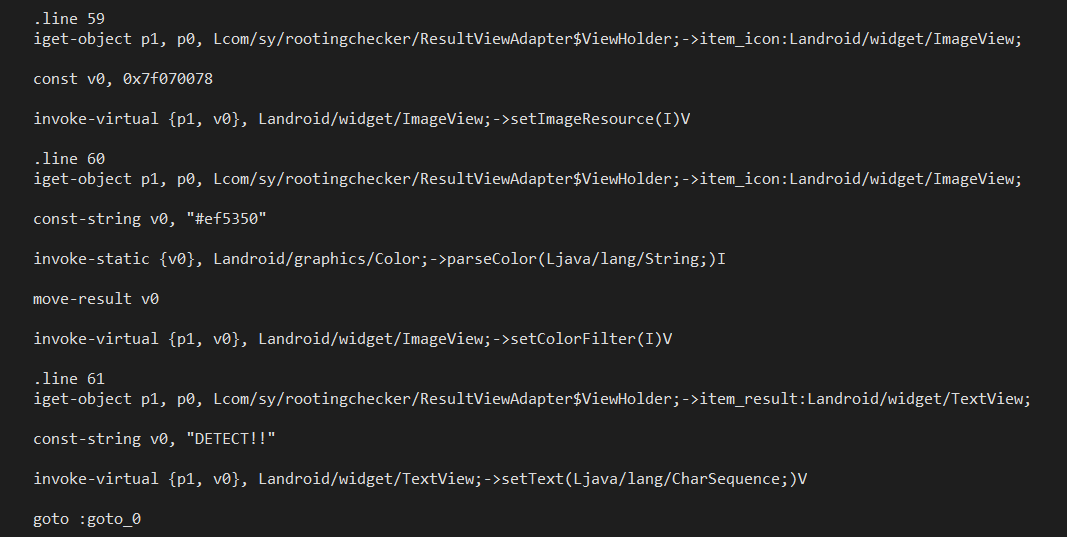
#방안1: Native Library의 이름을 임의로 변경하여 실행되지 않게 만들 수 있다.

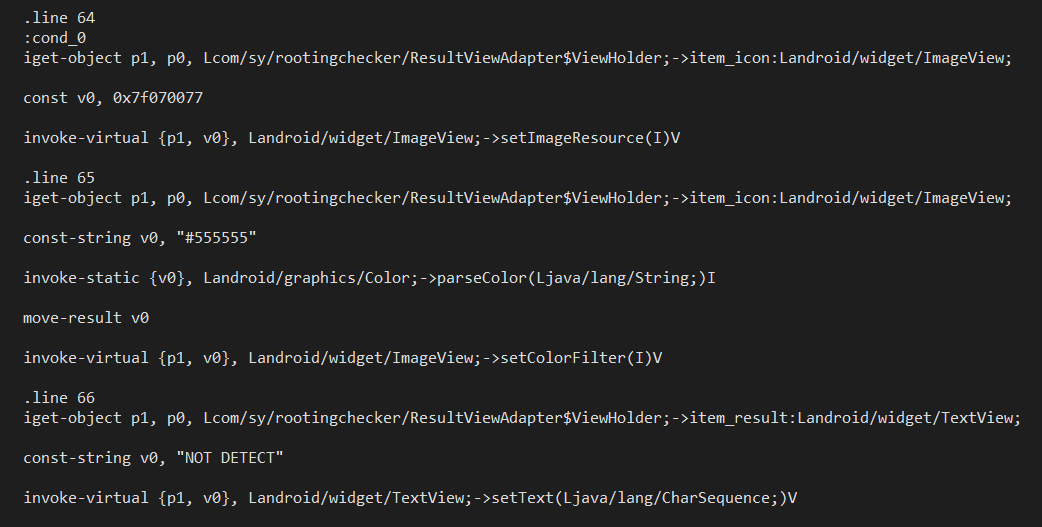
H. ResultViewAdapter$ViewHolder

결과값에 따라 UI를 설정하는 코드를 확인할 수 있다.



#방안1: const/4 v0, 0x1에서 0x1대신 0x0으로 변경하여 NOT\_DETECT로 분기되게 변경한다.

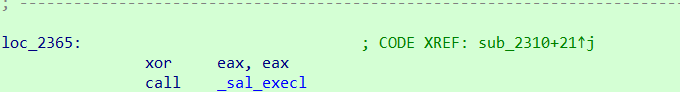




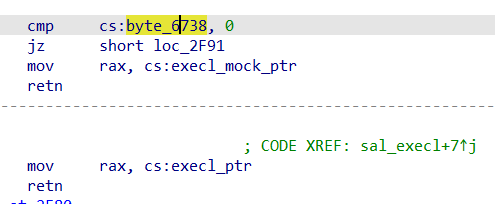
#방안2: DETECT에 설정되는 값들을 NOT DETECT에 설정되는 값들과 동일하게 변경한다.

2. C Layer

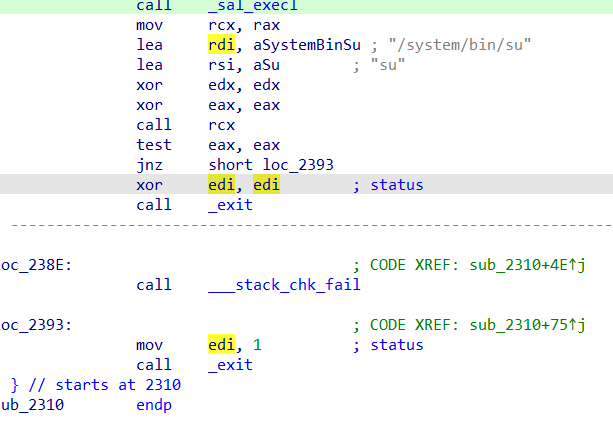
A. detect\_module\_1  
detect\_module\_1 function의 assembly를 살펴보면 아래와 같이 \_sal\_execl 함수를 호출하는 것을 볼 수 있다.



\_sal\_execl을 따라가보면 아래와 같이 rax 레지스터에 “execl\_mock\_ptr” 또는 “execl\_ptr”을 호출할 수 있는 값이 저장됨을 알 수 있다.



다시 돌아와 살펴보면 rdi와 rsi 레지스터에 각각 “/system/bin/su”와 “su”라는 값을 저장하고 rax에 저장되었던 함수가 호출됨을 알 수 있다. 즉, 해당함수의 parameter로 rdi와 rsi 레지스터에 값이 쓰임을 추측할 수 있다.



함수 호출 후 “jnz short loc\_2393” 구문 아래로 보면 edi을 0으로 만들거나, 1로 만든 곳으로 분기를 타는 것을 볼 수 있다. 이것이 결과값을 제어하는 부분을 추측할 수 있다. 추측한대로 mov edi, 1을 mov edi 0으로 바꿔보았지만 변화가 없다. 그렇다면 탐지가 되었을 때 해당코드를 탄다고 볼 수 없으므로 분기가 반대로 발생할 수 있게 수정해보자.

#방안1: \_sal\_execl의 리턴값을 1로 고정. 즉, eax의 값을 1로 고정

xor edx, edx

xor eax, eax

call rcx

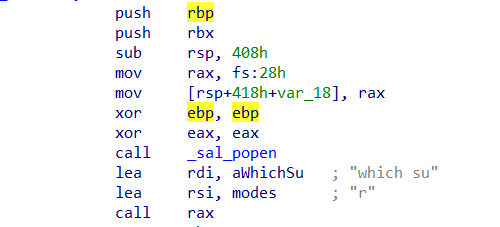
에 해당하는 hex 값 “0x31, 0xD2, 0x31, 0xC0, 0xFF, 0xD1”를

mov eax, 1

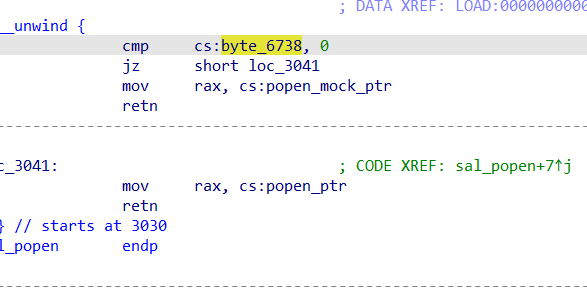
nop

에 해당하는 hex 값 “0xb8 0x01 0x00 0x00 0x00 0x90”로 변경한다.

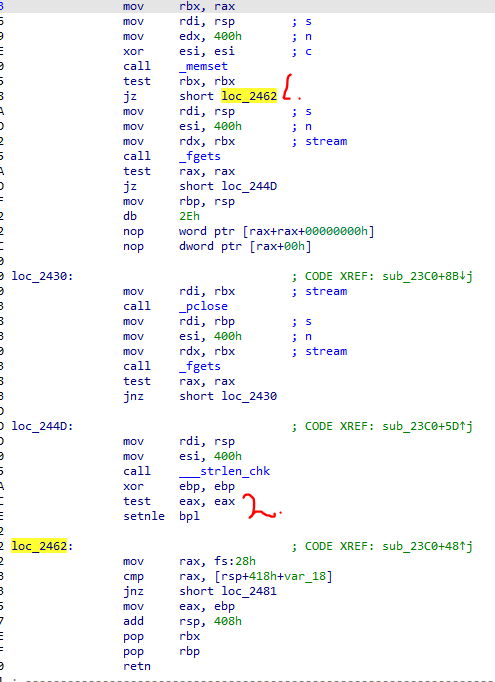
B. detect\_module\_2



“which su”, “r” parameter를 통해 함수를 호출하는 것으로 보인다.



호출되는 함수는 \_sal\_popen함수를 트래킹하여 보면 popen\_mock 또는 popen로 보인다. 함수 이름으로 추측하였을 때 “which su”을 실행한 결과로 루팅 여부를 판단하고 있다고 예측할 수 있다.



루팅 탐지를 회피하기 위해서 명령어 실행 결과를 fget으로 read 하기 전 또는 strlen 후 결과값을 check하는 부분을 조작할 수 있을 것으로 보인다.

#방안1:

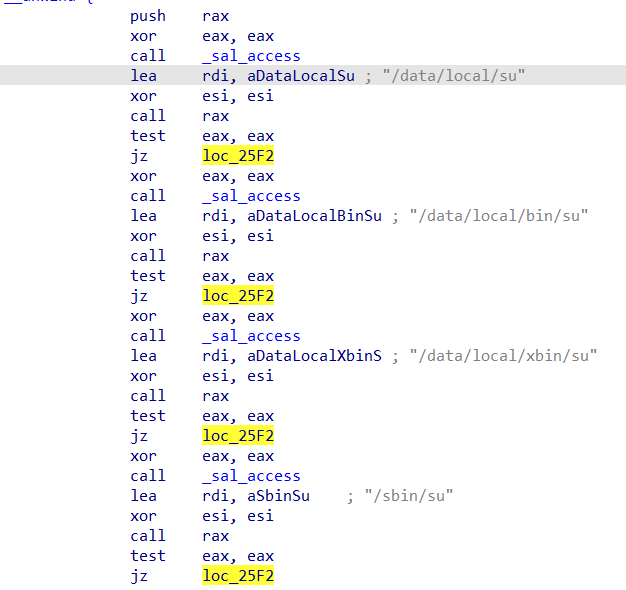
xor esi, esi  
call \_memset

에 해당하는 “0x31, 0xF6, 0xE8, 0x00, 0x00, 0x00, 0x00” 대신

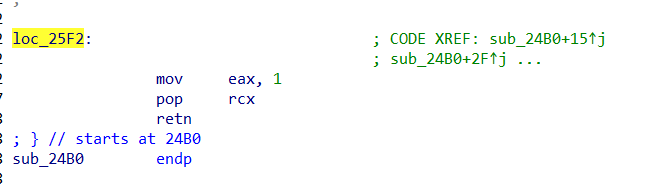
mov rbx, 0

에 해당하는 “0x48 0xc7 0xc3 0x00 0x00 0x00 -x00”를 삽입한다.

C. detect\_module\_3



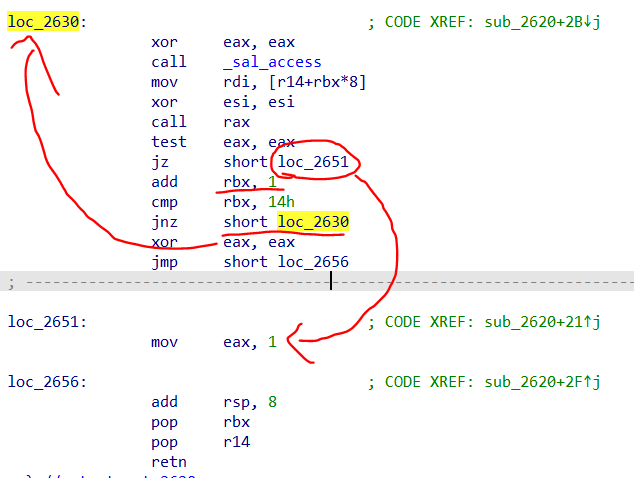
특정 문자열들을 parameter로 \_sal\_access함수를 호출한 결과를 통해 loc\_25F2로 점프하는 것을 볼 수 있다.



\_sal\_access 함수 호출을 통해 다른 문자열들을 확인하지 않고 점프한다는 것은 루팅 탐지가 되었다고 판단한 것으로 추측할 수 있다.

#방안1: loc\_25F2의 mov eax, 1을 0으로 변경한다.

D. detect\_module\_4



반복문의 패턴을 보이고 있다. 즉, rbx 레지스터의 값을 1씩 증가시키며 14가 될 때까지 반복하고 있다.

add rbx, 1

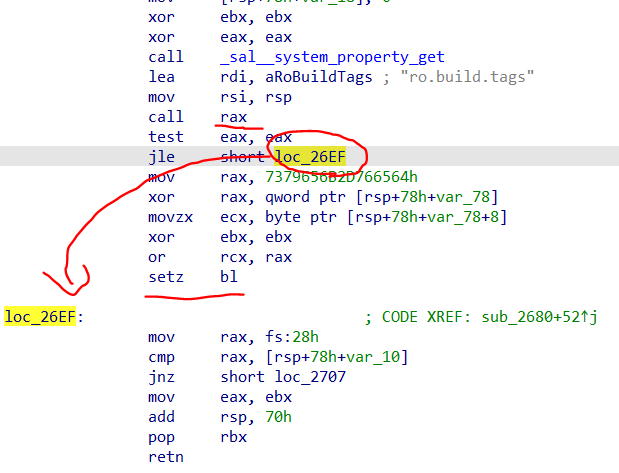
cmp rbx, 14h

jnz short loc\_2630

반복문 중간에 jz short loc\_2651은 반복문을 빠져나가는 구문이 있다. 즉, 반복문을 끝내지 않고 중간에 나간다는 것은 루팅 탐지를 했다는 것으로 추측할 수 있다.

#방안1: loc\_2651:에서 mov eax, 1을 0으로 변경한다.

E. detect\_module\_5



\_sal\_\_system\_property\_get이라는 함수를 호출하고 build property key word인 “ro.build.tags”가 있는 것이 보이는 것으로 “ro.build.tags” property 값을 read하여 확인하는 것으로 추측할 수 있다. 함수 호출 후 setz bl을 통해 return값을 변경하는 것으로 보이며, test eax, eax 수행 후 결과 값에 따라 loc\_26EF로 점프하는 것은 return 값을 설정하지 않고 return하는 것으로 보인다.

#방안1: setz bl까지 가지 않고 loc\_26EF로 점프하게 수정한다.

즉,

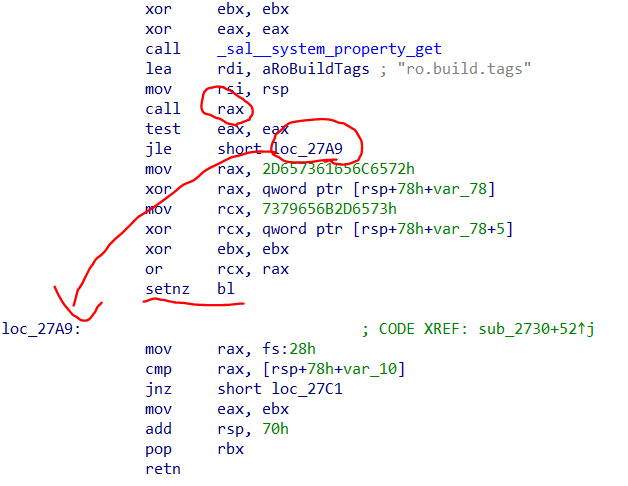
mov rsi, rsp

call rax

에 해당하는 “0x31 0xC0 0xE8 0x8C 0xF8”를 mov eax, 0에 해당하는 “b8 00 00 00 00”으로 교체한다.

F. detect\_module\_6

Detect\_module\_5와 거의 같은 패턴으로 보인다.



setnz bl 수행하기 전에 코드를 수정한다.

#방안1:

mov rsi, rsp

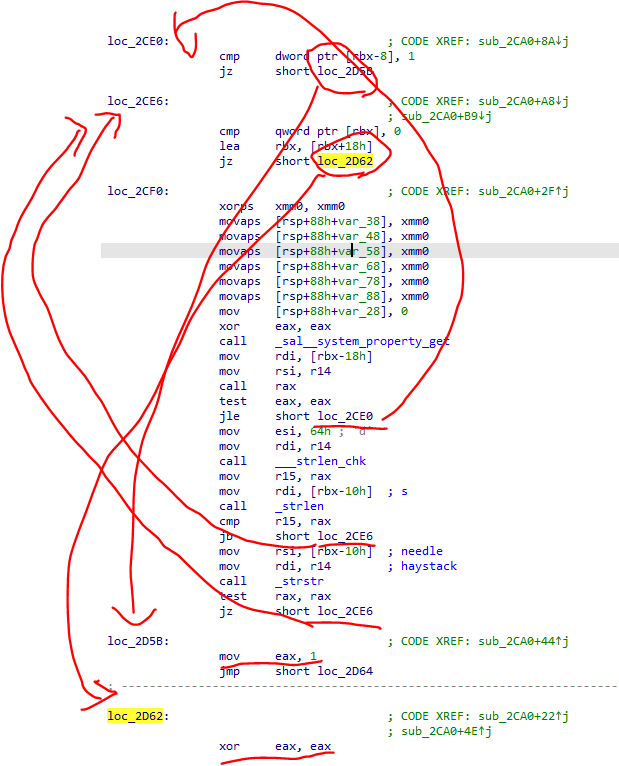
Call rax

에 해당하는 “0x48 0x89 0xE6 0xFF 0xD0”를

mov eax, 0

에 해당하는 “0xb8 0x00 0x00 0x00 0x00”로 변경한다.

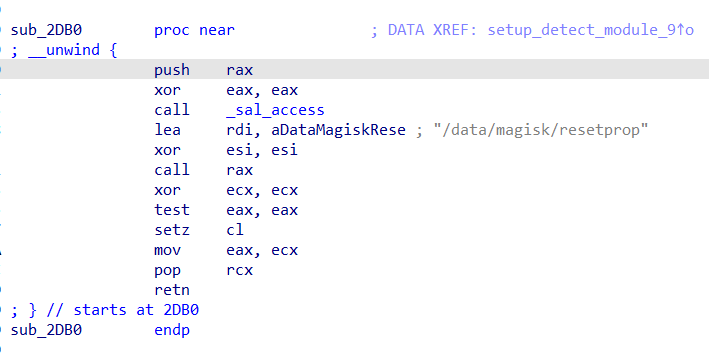
G. detect\_module\_7



분기가 어려 방향으로 가고 있다. 하지만 함수의 리턴값으로 쓰이는 eax 레지스터의 값을 제어하는 2개의 분기로 결국 마치고 있다. 즉, 루팅탐지가 detect되었을 경우, loc\_2D5B, 탐지된 것이 없을 경우 loc\_2D62로 가는 것으로 예측할 수 있다.

#방안1: loc\_2D5B에서 mov eax, 1을 0으로 변경한다.

H. detect\_module\_8



rdi에 저장된 값을 \_sal\_access를 통해 확인하는 것으로 보인다. 함수 호출 후 결과값을 통해 setz cl로 루팅 탐지 결과를 반환한다.

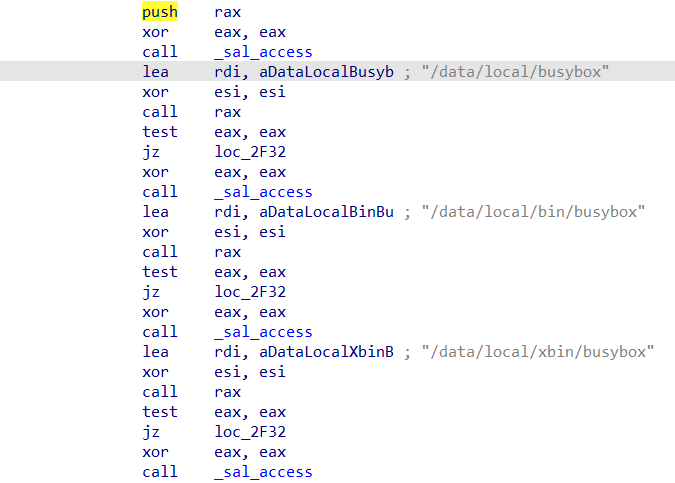
#방안1:

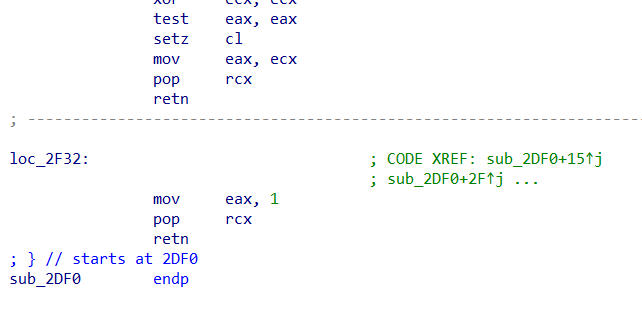
setz cl

mov eax, ecx

에 해당하는 “0x0F 0x94 0xC1 0x89 0xC8”를 “0xb8 0x00 0x00 0x00 0x00”로 변경한다.

I. detect\_module\_9





detect\_module\_3와 똑 같은 패턴이다.

#방안1: 최종 detect 결과값을 return 하는 mov eax, 1을 0으로 변경한다.

**3. 테스트**

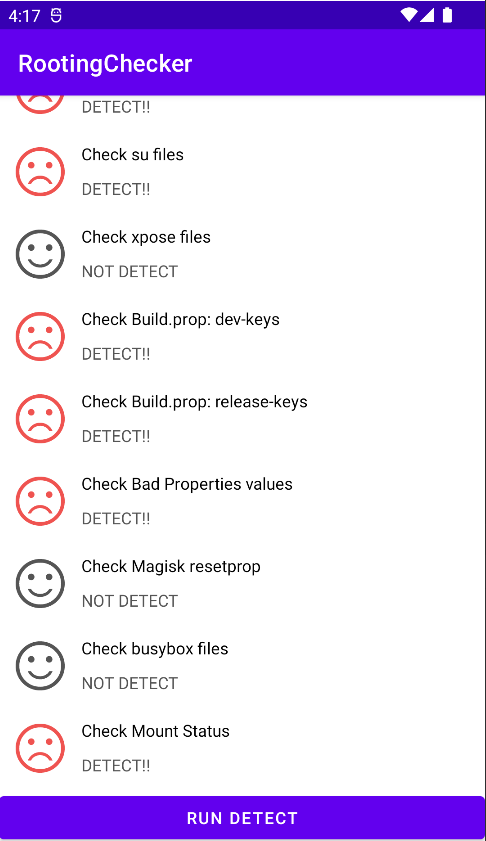
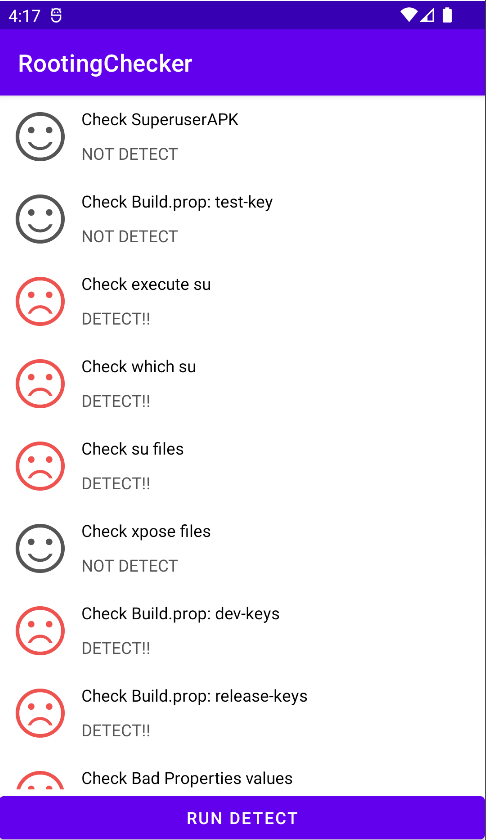
A. APK 설치

분석된 기법을 바탕으로 apk를 수정 후 아래와 같이 apk를 새로 설치

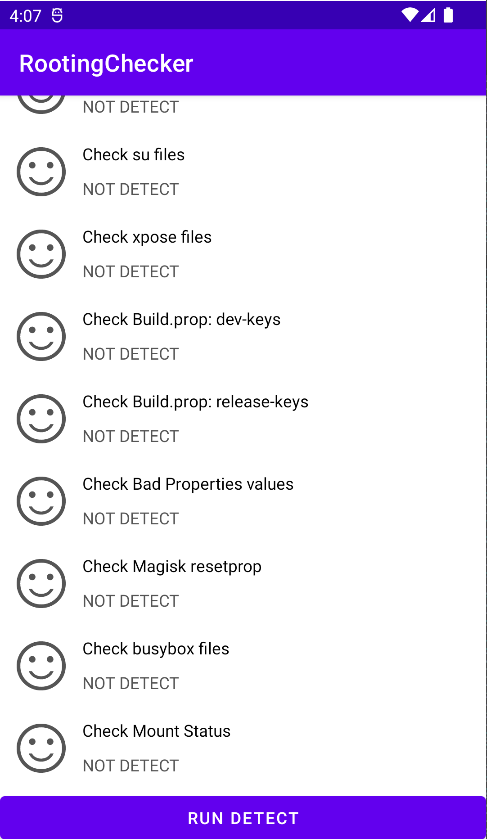
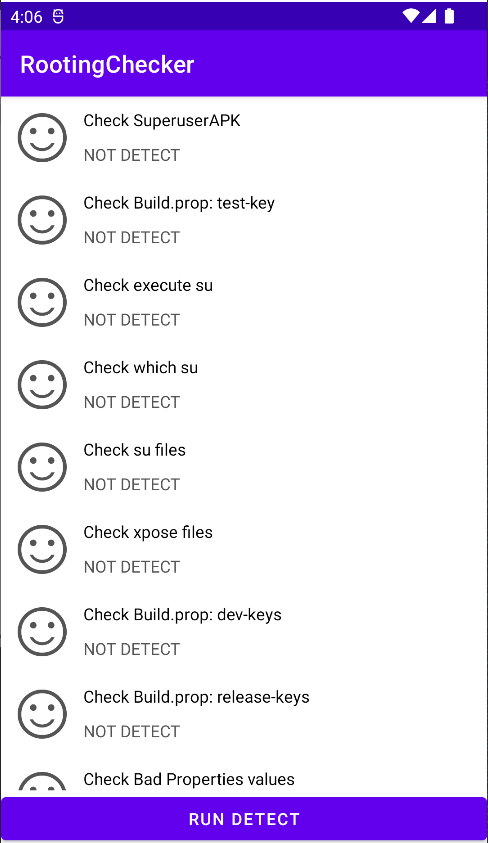
1. android:extractNativeLibs="false"
2. java -jar apktool\_2.6.1.jar d d:\analysis\_apk\app-release.apk -f
3. java -jar apktool\_2.6.1.jar b d:\apktool\app-release -o modified.apk
4. zipalign.exe -f -p 4 d:\apktool\modified.apk d:\apktool\output.apk
5. apksigner.bat sign -ks D:\Study\Dev\key\signkey.jks d:\apktool\output.apk
6. adb install -r output.apk

B. 적용 결과

1. Rooting 탐지 회피 적용 전



1. Rooting 탐지 회피 적용 후



**4. 보완 방안**

1. 개발 파일들의 이름을 명시적으로 해당 클래스가 하는 역할에 따라 작성하는 것보다 의미 없는 이름으로 저장한다면 파일을 특정하는데 방해를 할 수 있다.
2. 문자열을 Hard Cording하여 명시하는 것보다 문자열에 대한 Hash 값을 생성하여 코드에 삽입하고 검색 대상의 해시 값과 비교한다면 명시적으로 어떤 문자열을 회피하려는 것을 보완할 수 있다.
3. 검색 문자열에 대해 signature을 생성한다. Rooting 탐지 전 검색하려는 문자열에 대한 Signature verify를 수행하고 실패한다면 Rooting 탐지회피를 위한 특정 노력이 적용된 것으로 볼 수 있다.
4. Java Code로 작성하는 것보다 Native Code로 작성하는 것이 Decompile 수행 후 분석하는데 어려움을 줄 수가 있다.
5. 코드는 가독보다 난독에 중점을 두어 작성한다.