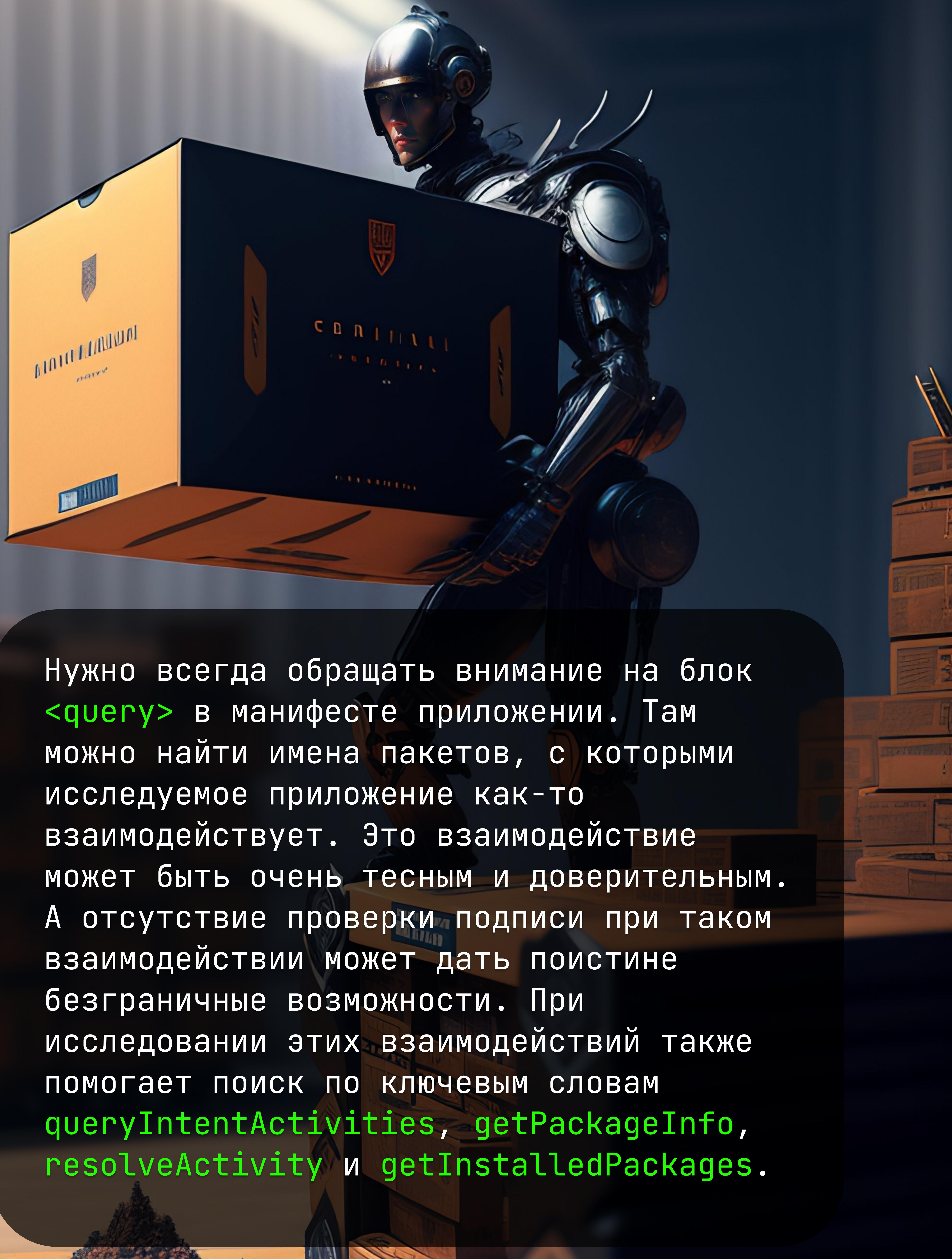
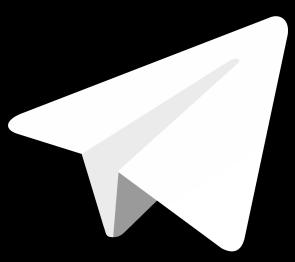


При изучении сложных конструкций в декомпилированном коде, или даже оригинальном исходном коде, полезно в динамике понимать как работает тот или иной алгоритм или его часть. Чтобы не переписывать это на какой-нибудь Python или не создавать новый проект в Android Studio/IntelliJ IDEA - можно воспользоваться **scratch-файлами**. Кроме того, что они упрощают написание PoC-ов, они также являются кросс-проектными, а значит можно написать несколько базовых скриптов и они сразу будут доступны во всех новых исследуемых приложениях. При условии, что вы используете Android Studio/IntelliJ IDEA конечно.



Нужно всегда обращать внимание на блок `<query>` в манифесте приложения. Там можно найти имена пакетов, с которыми исследуемое приложение как-то взаимодействует. Это взаимодействие может быть очень тесным и доверительным. А отсутствие проверки подписи при таком взаимодействии может дать поистине безграничные возможности. При исследовании этих взаимодействий также помогает поиск по ключевым словам `queryIntentActivities`, `getPackageInfo`, `resolveActivity` и `getInstalledPackages`.

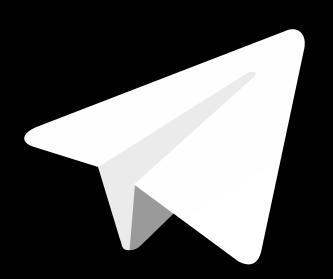


Если приложение использует биометрию, то нужно сразу искать флаги `setUserAuthenticationRequired` и `setInvalidateByBiometricEnrollment`.


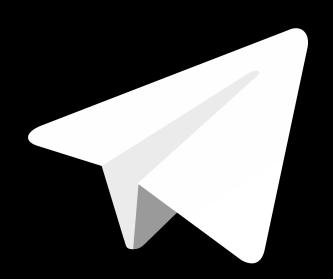
Если они установлены в `false`, то это может позволить обойти биометрическую аутентификацию при наличии физического доступа к устройству.

Первый флаг устанавливается когда нужно ограничить использовать ключа шифрования биометрическими данными. Без правильной биометрии - ключ не будет доступен для использования. Соответственно, если он `false`, то биометрический диалог можно просто скрыть с помощью Frida.


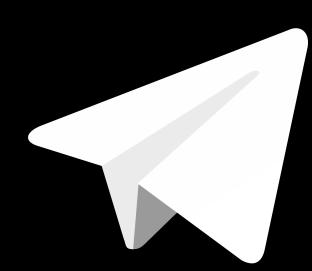
Второй флаг, установленный в `false` позволяет злоумышленнику добавить в систему свой палец/лицо и получить доступ к приложению используя уже свои биометрические данные.



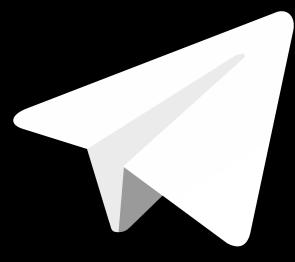
После декомпиляции приложения нужно поискать секреты в коде. Для этого можно использовать `trufflehog` или любую другую подобную утилиту. Пример запуска `trufflehog`: `trufflehog filesystem --directory=`pwd` --only-verified` Но иногда, отсутствие флага `only-verified` дает интересные результаты. Поэтому обязательно попробуйте. А как быть с найденными секретами дальше, подскажет `keyhacks`



Когда исследуешь сильно обfuscированное приложение, то приходится заниматься восстановлением оригинальных имен переменных и функций вручную. Или придумывать такие имена самостоятельно исходя из контекста использования. Кроме этого, иногда приходится удалять целые куски мусорного кода и совершать прочие рефакторинги. Чтобы проще было контролировать происходящее, откатываться или проверять гипотезы без вреда для кодовой базы - можно добавить декомпилированный код в локальный `git` репозиторий и пользоваться всеми прелестями этого инструмента.

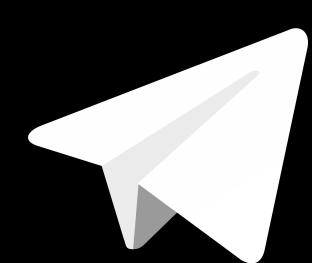


Перепаковка приложения может быть довольно болезненной процедурой. Чтобы избежать возможных глюков после пересборки, можно воспользоваться специальными флагами apktool: `-s` распакует приложение, но не будет декомпилировать код, а `-r` распакует без декодирования ресурсов. Если нужно только убрать ssl pinning через `network_security_config.xml`, то нет нужды декомпилировать код. А если требуется убрать проверку на root, то ресурсы можно не декодировать.

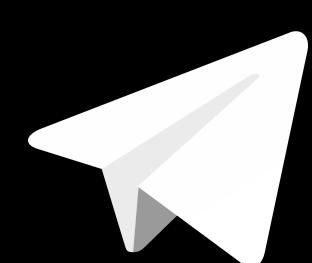


WARNING

Большие компании часто выпускают разные приложения, но используют в них какие-то общие библиотеки, архитектуру и подходы. Поэтому, для разбора алгоритмов смысл которых непонятен из-за обfuscации, можно попробовать разобрать другие приложения этой компании. Их могли писать другие команды с низкой социальной ответственностью, а значит там может вообще не быть обfuscации или она будет не такая злая.



Даже если на устройстве есть root доступ, команды `adb push/pull` все равно будут работать в пользовательском режиме. Это значит, что не выйдет скачать или отправить файлы в директории доступные только суперпользователю. Но выход есть. Вместо `adb push` будет `cat file.txt | adb shell su -c "dd of=/data/file.txt"`, а для замены `adb pull` - `adb shell su -c "dd if=/data/file.txt" >> file.txt`. В целом, так можно выполнить любую команду с правами root - `adb shell su -c "<command>"`.



# HACKSES

Обфусцированный код, при более пристальном рассмотрении, порой оказывается не таким уж и обфусцированным. Например по методу `toString()` часто можно восстановить оригинальное имя класса и имена полей. Для этого можно поискать по коду все объявления этого метода с помощью ключевого слова `toString() {` и найти что-то вроде этого:

```
public final String toString() {  
    StringBuilder m20399b =  
        C0002b.m20399b("Insets(left=");  
    m20399b.append(this.f39a);  
    m20399b.append(", top=");  
    m20399b.append(this.f40b);  
    m20399b.append(", right=");  
    m20399b.append(this.f41c);  
    m20399b.append(", bottom=");  
    return C0047d.m19892c(m20399b,  
        this.f42d, ')');  
}
```

Теперь можно переименовывать класс и поля, что сильно упростит дальнейшие исследования.

Если приложение закрыто пин-кодом и он реализован локально, то есть несколько типичных ошибок, которые допускают разработчики.

1. Приложение никак не ограничивает количество попыток ввода;
2. После перезпуска приложения - счетчик попыток сбрасывается;
3. Приложение некорректно проверяет метку времени и можно сбросить счетчик путем перевода времени в системе;
4. Окно с пин-кодом не является блокирующим и можно запустить основной экран напрямую через Intent.

Подбирать пин-код в автоматическом режиме можно например с помощью [Ui-automator](#). Для него также есть вполне рабочая библиотека на Python - [xiaosong/uiautomator](#).

