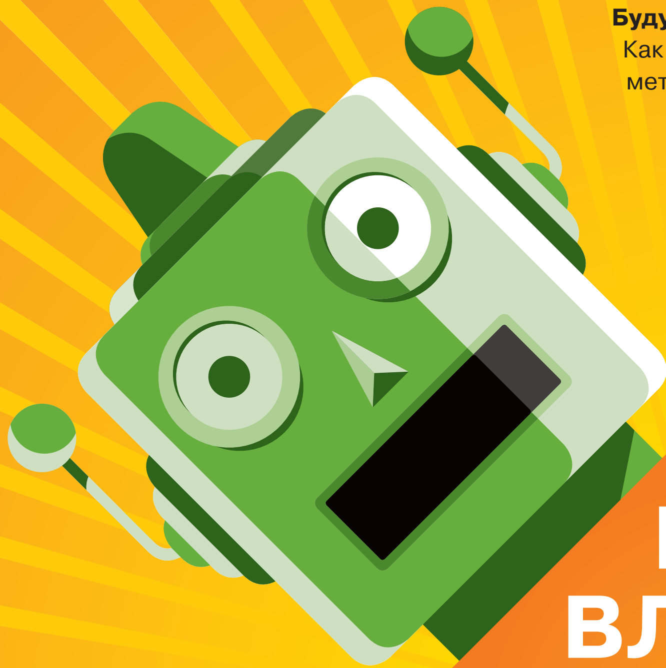
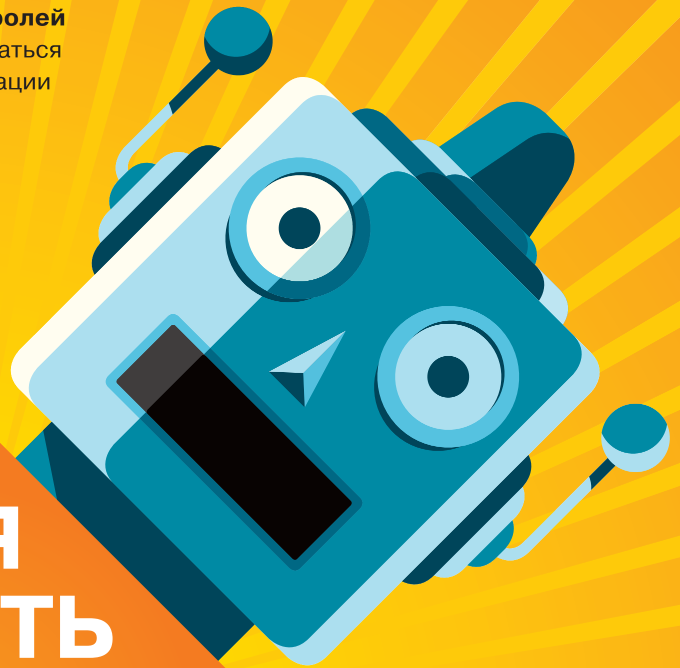
02 (181) 2014

ХАКЕР

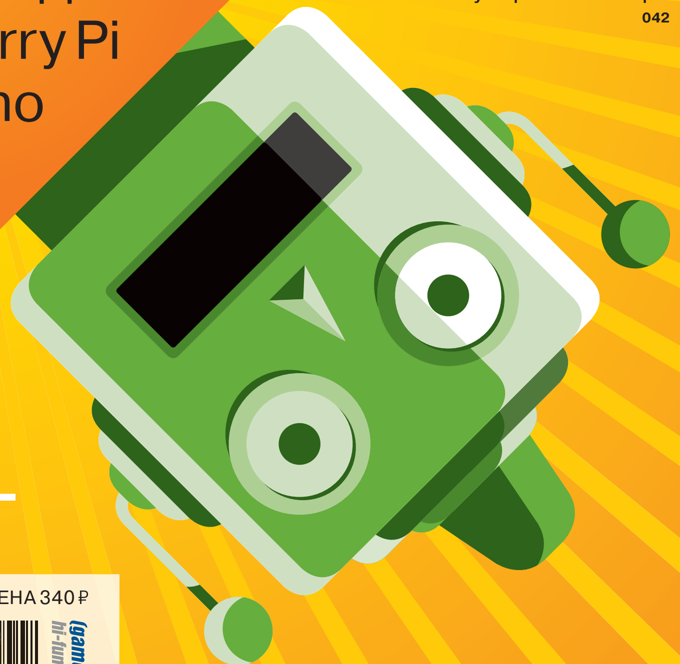
Будущее без паролей
Как будут развиваться методы авторизации
046

Наш гид по деанонимизации
Простые способы вычислить человека в Сети
082

ВСЯ ВЛАСТЬ РОБОТАМ!

Собираем модели на Raspberry Pi и Arduino

Можно ли еще заработать на Bitcoin?
Что делать, если у тебя нет под рукой суперкомпьютера
042



12+

РЕКОМЕНДОВАННАЯ ЦЕНА 340 ₽

Пришел, увидел, разобрал

Михаил Емельченков
soaw-security.is

kevinpoh@Flickr.com

Реверс-инжиниринг прошивки китайского Android-планшета

У китайцев особенное представление о копирайте — у них он просто не действует. В то же время свои наработки они защищают различными техническими средствами, почему-то «забывая» делиться ими со своими клиентами. Казалось бы, ситуация безвыходная: поступила партия китайских планшетов, и встала задача прошить их таким образом, чтобы контент заказчика не стирался при сбросе настроек, при этом имеется стоковая прошивка в неизвестном bin-формате, но отсутствует SDK. Что же делать, как собрать кастомную прошивку? Выход один — применить реверс-инжиниринг.

РАЗВЕДКА

Устройство, с которым предстояло поработать, было построено на базе Generalplus GP330xx SoC (bit.ly/1lnxm6L), а его системное ПО разработано с помощью Open Platform SDK, и, хотя китайцы заявляют о готовности (bit.ly/1dEuSNG) предоставить исходные коды, делать они этого не спешат. Несмотря на сложность поставленной задачи, оптимизма прибавлял включенный в устройстве по умолчанию рутовый доступ. Поэтому процесс изучения начался с запуска ADB Shell.

Все дисковое пространство планшета представляло собой одно большое блочное устройство NAND-флеш (/dev/block/nanda), побитое на разделы:

```
Disk /dev/block/nanda: 7457 MB, 7457472512 bytes
4 heads, 16 sectors/track, 227584 cylinders
Units = cylinders of 64 * 512 = 32768 bytes
Device Boot       Start    End      Blocks    Id System
/dev/block/nanda1 257      174335   5570528   b Win95 FAT32
/dev/block/nanda2 174336   207103   1048576   83 Linux
/dev/block/nanda3 207104   223487   524288    83 Linux
/dev/block/nanda4 223488   227583   131072    83 Linux
```

Часть памяти была выделена под так называемую Internal SD card. Нужно остановиться на этом термине подробнее. В Android каждая прикладная программа запускается в своей песочнице и использует для доступа к файлам системный API. Этот API позволяет обращаться к внутренней памяти (Internal Storage) и внешней памяти (External Storage). При этом внешняя память делится на removable storage media (SD-карта, которая вставляется в слот на торце устройства) и internal (non-removable) storage (раздел внутренней памяти, «мимикрирующий» под SD-карту). В данном планшете именно под внутреннюю SD-карту был отведен самый большой раздел — /dev/block/nanda1. Поэтому его решено было разбить на два раздела, выделив один из них под контент заказчика, а второй — под внутреннюю SD-карту. Устройство /dev/block/nanda размечено с помощью MBR, а не GPT, поэтому максимальное количество primary разделов равно четырем. С помощью fdisk был удален раздел /dev/block/nanda1, и на его месте создан extended-раздел с двумя подразделами /dev/block/nanda5 и /dev/block/nanda6.

КОЛДУЕМ НАД РАЗДЕЛАМИ

Просматривая список смонтированных устройств, видим, что раздел /dev/block/vold/253:97 смонтирован на /mnt/sdcard.

```
root@android:/etc # mount
...
/dev/block/vold/253:97 /mnt/sdcard vfat rw,dirsync,nosuid,
nodev,noexec,relatime,uid=1000,gid=1015,fmask=0602,
dmask=0602,allow_utime=0020,codepage=cp437,
iocharset=iso8859-1,shortname=mixed,utf8,errors=remount-ro 0 0

/dev/block/vold/253:97 /mnt/secure/asec vfat rw,dirsync,
nosuid,nodev,noexec,relatime,uid=1000,gid=1015,fmask=0602,
dmask=0602,allow_utime=0020,codepage=cp437,
iocharset=iso8859-1,shortname=mixed,utf8,errors=remount-ro 0 0
...
```

Какая связь между /dev/block/vold/253:97 и /dev/block/nanda1? Vold — это Volume Management daemon, демон монтирования внешних носителей. У него имеется конфигурационный файл, по синтаксису похожий на стандартный никсовый fstab, под названием vold.fstab:

```
## Vold 2.0 Generic fstab
...
dev_mount sdcard /mnt/sdcard auto /devices/virtual/block/↵
nanda /devices/virtual/block/nanda/nanda1 /devices/virtual/↵
block/nanda/nanda2 /devices/virtual/block/nanda/nanda3 ↵
/devices/virtual/block/nanda/nanda4
...
```

На первый взгляд все понятно: /mnt/sdcard — это путь монтирования, auto — автоматический выбор первого подходящего для монтирования раздела из списка разделов, указанных далее (/devices/virtual/…). Однако файл vold.fstab в данном устройстве был, по сути, «заглушкой». При внесении модификаций в строчку dev_mount sdcard... (например, подмонтировать свежесозданный раздел, отличный от /devices/virtual/block/nanda/nanda1), демон отказывался работать. Трудно сказать наверняка, связано ли это с кастомизированным ядром или же с кастомизированным демоном, но, как бы то ни было, мотивы разработчиков такого решения не ясны.

Таким образом, оказалось, что ни /dev/block/nanda5, ни /dev/block/nanda6 невозможно подмонтировать с помощью vold. Дальше можно было пойти двумя путями:

1. Запускать монтирование SD-карты из init-скриптов вручную. Правда, этот путь не мог гарантировать 100%-й совместимости со всеми Android internals, иными словами, нельзя было бы поручиться за стабильность работы системы, убрав из нее ключевой компонент «общения» с внешними накопителями — vold.
2. Взять открытые исходники vold и попробовать собрать его для данного устройства. Гарантий также никаких, кроме того, это могло бы потребовать изрядного количества времени, которого, как всегда, не хватало.

При этом пришлось бы пришлось бы писать shell-скрипты, вызываемые через ADB, и получить результирующий бинарник прошивки никак бы не вышло, а это, в свою очередь, удорожило бы работу технических специалистов заказчика, так что этот путь был оставлен про запас и исследование продолжилось в новом направлении.

РЕШЕНИЕ ПРИШЛО ВНЕЗАПНО

Столкнувшись с такой проблемой, я решил еще раз внимательно изучить то, что было у нас в руках. Особый интерес вызывал прошивальщик, который, помимо самого файла прошивки firmware.bin, содержал еще ряд вспомогательных ресурсов: bootheader.bin, bootpack.bin, bootresource.bin, scanram.bin, updater.bin. Они также необходимы, но нерелевантны для нашей задачи. Больший интерес представляют файлы, которые используются прошивальщиком для загрузки своего собственного кода на устройство: small_isp.bin, cmdline, initrd и kernel.

Данное устройство использовало для прошивки так называемый ISP mode (это обозначение одного из режимов программирования флеш-памяти). Алгоритм работы прошивальщика можно разделить на четыре этапа:

1. Технический специалист перезагружает устройство в режиме прошивки, зажав при его включении кнопки <Home + Power>.
2. Прошивальщик опознает устройство по USB и перезагружает его в ISP mode.
3. Прошивальщик загружает на устройство Linux, передавая файлы cmdline, initrd и kernel. Файл kernel — это ядро ОС, initrd — раздел с ПО прошивальщика на стороне устройства, cmdline — параметры ядра, содержащие в себе размер файла initrd.
4. Загруженная на устройстве Linux начинает принимать от прошивальщика основные файлы прошивки, распаковывать их и записывать в соответствии с внутренними алгоритмами.

Что же представляли собой эти внутренние алгоритмы? Решение пришло внезапно. Оказалось, что initrd содержал в себе исходные коды прошивальщика на языке Lua, а также бинарники дополнительных Lua-модулей. Для распаковки initrd необходимо выполнить следующие команды:

```
# mkdir initrd-unpacked
# cd initrd-unpacked
# gunzip < ../initrd | cpio -i --make-directories
```

Для обратной упаковки (при необходимости; например, для тестирования модифицированных версий скриптов):

```
# find ./ | cpio -H newc -o > initrd.cpio
# gzip initrd.cpio
# mv initrd.cpio.gz initrd
```

Секция	Описание
Заголовок 1	Содержит в себе информацию о размерах firmwareImage, userImage, dataImage
Заголовок 2	Содержит в себе информацию о распакованных и сжатых kernel.bin и system.bin, их контрольные суммы, общую контрольную сумму, таблицу разделов и ряд других параметров
firmwareImage	
kernel.bin	LZO-packed kernel + initrd
system.bin	LZO-packed ext4 FS /system
userImage	GZ-packed Контент /mnt/sdcard, записываемый при первоначальной прошивке
dataImage	GZ-packed Контент /data, записываемый при первоначальной прошивке и сбросе настроек

Рис. 1. Формат прошивки планшета

Это может показаться странным, но действительно разработчики зачем-то придумали свой собственный формат прошивки, при этом оставив скрипты, оперирующие с этим форматом, в initrd в открытом виде.

PLUTO

Хидеры прошивки планшета были запакованы с помощью модуля Pluto (bit.ly/195V1Wj), который упаковывает Lua-таблицы в бинарный формат. Язык программирования Lua вообще активно использует подключаемые модули, представляющие собой so-библиотеки, которые добавляют те или иные API. Вдобавок ко всему, как следовало из документации, Pluto был платформо- и архитектурнозависим. Intel и ARM (на которой был построен планшет) существенно отличаются: Intel использует little-endian порядок байт в представлении чисел, а ARM — big-endian.

И здесь возникла серьезная проблема: стандартный модуль Pluto не распаковывал полученные данные. Были испробованы разные версии Lua и даже разные архитектуры CPU (x86, x86_64, ARM). Оказалось, что просто разработчики прошивки использовали свою, ни с чем ни совместимую версию Pluto.

Для того чтобы распаковать данные, пришлось воспользоваться эмулятором QEMU для архитектуры ARM и установить на него Debian Linux. А затем установить Lua и положить модуль pluto.so, извлеченный из initrd, в каталог модулей Lua.

LZO

Отдельную сложность преподнес также алгоритм сжатия LZO. Дело в том, что формат данных для этого алгоритма архивации не стандартизирован, поэтому сложно написать распаковщик, не зная, каким образом файл был запакован. Однако среди Lua-модулей initrd был модуль `lua_lzo.so`. На помощь пришел метод, описанный в предыдущем абзаце, правда, усложненный тем, что lua_lzo.so требовал в зависимости системную библиотеку `liblzo.so` (которая была взята из того же initrd) и нестандартное подключение модуля через `package.cpath`.

Распаковка выполняется в цикле, блоками данных. Для распаковки используются функции:

1. handle = `lzo.decompressInit(header)`, где
   - header — magic number + размер блока архивации;
   - handle — хендл, использующийся в двух других функциях.

2. `... = lzo.decompressPorcess(handle)`
3. `lzo.decompressFinish(handle)`

Примечательно то, что необходимо точно знать размер архива, чтобы распаковка выполнилась успешно. В противном случае распаковка зависает на статусе `DECOMPRESS_NEED_MORE_DATA`. Размер архива указан в заголовке 2 (см. рис. 1).

Компрессия данных выполняется сложнее, так как функции компрессии не документированы и их работоспособность выявлялась пробным путем. Функции аналогичны:

1. handle, header `= lzo.compressInit(blockSize)`
2. `... = lzo.compressProcess(handle, data)`
3. `lzo.compressFinish(handle)`

Отличительный момент компрессии от декомпрессии в том, что перед записью блока данных, полученных в результате выполнения функции `lzo.compressProcess`, необходимо записать размер упакованного блока данных. Это следует из общей документации на алгоритм сжатия LZO и из анализа архива, полученного при разборе оригинальной прошивки.

В итоге, исследуя исходный код скриптов, пытаясь понять их логику работы, форматы данных, а также проведя множество экспериментов, прошивку удалось распаковать.

РЕСАЙЗ СИСТЕМНОГО РАЗДЕЛА

Распакованный файл системного раздела (`system.bin`) представлял собой образ файловой системы ext4. Для того чтобы записать данные заказчика, его необходимо было расширить на 1 Гб. Для этого нужно сделать следующее:

1. Расширить саму файловую систему.
2. В заголовке 2, в таблице разделов, уменьшить на 1 Гб раздел `nanda1` и увеличить на столько же раздел `nanda2`.
3. Снова заархивировать `system.bin`, пересчитать контрольные суммы и записать их в заголовки.

Сам же ресайз системного раздела выполняется следующими командами:

```
# mkdir system_new
# losetup /dev/loop0 system.bin
# e2fsck -f /dev/loop0
# resize2fs /dev/loop0 2G
# mount /dev/loop0 system_new
...
# umount system_new
# losetup -d /dev/loop0
```

РАБОТА С DATA-РАЗДЕЛОМ

В рамках решения данной задачи часть изменений в системе производилась не только в `/system`, но и в `/data`. Для этого необходимо было распаковать `dataImage.tar.gz`, сделать необходимые изменения и запаковать обратно. Подобным образом следует поступить и с `userImage.tar.gz`, если требуется внести изменения в контент SD-карты.

Для упаковки с сохранением прав доступа используем команды:

```
Mikhail — ssh — 120×59
root@debian:~/original# lua printHeaders.lua
----- Header 1 -----
        table
        [userImageSize] number  6961484
        [preloadOffset] number  716142616
        [dataPreloadHeader]     string
        [data]  string  MRIF
        [firmwareImageSize]     number  716142584
        [dataImageSize] number  27333689
        [animationSize] number  0
        [animationType] number  0
        [preloadImage]  boolean true
        [file]  userdata        nixio file 3
        [dataPtr]       number  0
        [header1]       table
        [header1]       [tag]   string  FIRC
        [tag]   string  FIRC

----- Header 1 + 2 ------
        table
        [upgradeSectionOffset]  number  3840
        [data]  string  PMOC
        [firmwareImageSize]     number  716142584
        [animationType] number  0
        [preloadImage]  boolean true
        [dataPreloadHeader]     string
        [header1]       table
        [header1]       [tag]   string  FIRC
        [userImageSize] number  6961484
        [preloadOffset] number  716142616
        [upgradeFlag]   string
        [dataImageSize] number  27333689
        [upgradeFileHeader]     table
        [upgradeFileHeader]     [nandVersion]   string  V 0.25.0
        [upgradeFileHeader]     [VENDOR_VERSION]        string  0.01
        [upgradeFileHeader]     [sectionTable]  table
        [upgradeFileHeader]     [sectionTable]  [1]     table
        [upgradeFileHeader]     [sectionTable]  [1]     [offset]        number  0
        [upgradeFileHeader]     [sectionTable]  [1]     [flag]  table
        [upgradeFileHeader]     [sectionTable]  [1]     [flag]  [compression]   boolean true
        [upgradeFileHeader]     [sectionTable]  [1]     [flag]  [compression_size]      number  131072
        [upgradeFileHeader]     [sectionTable]  [1]     [partitoin]     string  nanda_data0
        [upgradeFileHeader]     [sectionTable]  [1]     [crc]   string ????Y????%??Qh
        [upgradeFileHeader]     [sectionTable]  [1]     [name]  string  kernel
        [upgradeFileHeader]     [sectionTable]  [1]     [padding]       number  2
        [upgradeFileHeader]     [sectionTable]  [1]     [file]  string  kernel.bin
        [upgradeFileHeader]     [sectionTable]  [1]     [compressSize]  number  5020014
        [upgradeFileHeader]     [sectionTable]  [1]     [size]  number  5295108
        [upgradeFileHeader]     [sectionTable]  [2]     table
        [upgradeFileHeader]     [sectionTable]  [2]     [offset]        number  5020016
        [upgradeFileHeader]     [sectionTable]  [2]     [flag]  table
        [upgradeFileHeader]     [sectionTable]  [2]     [flag]  [compression]   boolean true
        [upgradeFileHeader]     [sectionTable]  [2]     [flag]  [compression_size]      number  131072
        [upgradeFileHeader]     [sectionTable]  [2]     [partitoin]     string  nanda2
        [upgradeFileHeader]     [sectionTable]  [2]     [crc]   string  ?MY?/XY2>???(
        [upgradeFileHeader]     [sectionTable]  [2]     [name]  string  system
        [upgradeFileHeader]     [sectionTable]  [2]     [padding]       number  1
        [upgradeFileHeader]     [sectionTable]  [2]     [file]  string  system.bin
        [upgradeFileHeader]     [sectionTable]  [2]     [compressSize]  number  711118727
```

2

Рис. 2. Заголовки бинарника прошивки в консоли ARM-эмулятора

ПОДКЛЮЧЕНИЕ LUA-МОДУЛЕЙ

Язык программирования Lua расширяется за счет внешних подключаемых модулей, которые могут быть написаны как на Lua, так и на C. В последнем случае это обычные so-библиотеки, экспортирующие ряд API-функций.

Их подключение производится с помощью функции `require`, а за путь поиска бинарных модулей отвечает переменная `package.cpath`. У проприетарного LZO-модуля есть своя особенность подключения, которая кроется в его наименовании — `lua_lzo.so`. При этом сам модуль называется lzo, из-за чего его подключение вместо привычного:

```
package.cpath = package.cpath .. "/home/mikhail/lua_so/?.so"
require "lzo"
```

следует производить так:

```
package.cpath = package.cpath .. "/home/mikhail/lua_so/lua_?.so"
require "lzo"
```

Также стоит обратить внимание на пакетный менеджер LuaRocks, который позволяет устанавливать модули из единого репозитория и удобно их подключать. Например, в рамках данного исследования модули nixio и MD5 были подключены именно через LuaRocks.

```
# tar cvf - . | gzip -9 - > ../user.tar.gz
# tar cvfp - . | gzip -9 - > ../data.tar.gz
```

ЗАМЕНА ПРИЛОЖЕНИЙ ПО УМОЛЧАНИЮ

Заказчику было нужно не только записать свой контент в постоянную память устройства, но и заменить стандартный launcher своим собственным приложением, обеспечив требуемый User Experience.

Замена launcher'а (и других приложений по умолчанию) производилась путем редактирования файлов `/data/system/packages.list` и `/data/system/packages.xml`. Сначала дефолтные настройки выполнялись на устройстве, затем содержание файлов частично переносилось в прошивку.

Файл `packages.list` представляет собой список установленных в системе пакетов. Нужный пакет launcher'а называется com.soaw.launcher и добавляется строчкой:

```
com.soaw.launcher 10068 1 /data/data/com.soaw.launcher
```

А `packages.xml` — это база данных установленных в системе пакетов и их метаданных, таких как сертификаты, права доступа, приложения по умолчанию и прочее. За настройку программ по умолчанию отвечают две записи. Первая запись — это метаданные launcher'а. Обрати внимание на атрибут `index` в теге `<cert>`, его значение должно быть на единицу больше уже существующего в файле, чтобы не случилось путаницы сертификатов.

```
<package name="com.soaw.launcher" codePath="/system/app/↵
 SOAWLauncher.apk" nativeLibraryPath="/data/data/com.soaw.↵
 launcher/lib" flags="1" ft="141c2c2bbe0" it="141c2c2bbe0" ↵
 ut="141c2c2bbe0" version="1" userId="10068">
    <sigs count="1">
        <cert index="20" key="..." />
    </sigs>
</package>
```

Следующая запись — это настройки программ по умолчанию. Здесь задается выбор launcher'а и программы-медиаплеера.

```
<preferred-activities>
  <item name="com.soaw.launcher/.activity.HomeActivity" ↵
   match="100000" set="2">
    <set name="com.android.launcher/com.android.launcher2.↵
     Launcher"/>
    <set name="com.soaw.launcher/.activity.HomeActivity"/>
    <filter>
      <action name="android.intent.action.MAIN"/>
      <cat name="android.intent.category.HOME"/>
      <cat name="android.intent.category.DEFAULT"/>
    </filter>
  </item>
  <item name="com.android.gallery3d/.app.MovieActivity" ↵
   match="600000" set="2">
    <set name="com.generalplus.GaGaPlayer/.↵
     MoviePlayerActivity"/>
    <set name="com.android.gallery3d/.app.MovieActivity"/>
    <filter>
      <action name="android.intent.action.VIEW"/>
      <cat name="android.intent.category.DEFAULT"/>
      <type name="video/mp4"/>
    </filter>
  </item>
</preferred-activities>
```

СИСТЕМНЫЕ НАСТРОЙКИ

Как известно, Android имеет SQLite базу данных системных настроек, которую возможно модифицировать на этапе подготовки образа прошивки. Файл базы данных находится в `/data/data/com.android.providers.settings/databases/settings.db`.

Сокрытие нижней панели выполняется в таблице `system` следующими записями:

```
navigation_bar_mode = 4
navigation_bar_buttons_show = 0
navigation_bar_buttons_need_show = 0
```

Отключение экрана блокировки выполняется в таблице `secure`:

```
lockscreen.disabled = 1
```

АЛГОРИТМ СЖАТИЯ LZO

LZO — семейство блочных алгоритмов сжатия, обладающих важными для портативных компьютеров характеристиками:

- очень высокой скоростью распаковки;
- малым потреблением памяти;
- поблочной распаковкой данных, порциями небольшого размера.

С точки зрения реверс-инжиниринга он имеет два недостатка:

- LZO включает в себя девять алгоритмов сжатия, и к каждому из них идет свой распаковщик;
- структура файлов LZO-архивов не стандартизирована, разные библиотеки генерируют разную структуру.

В нашем случае архивные данные имели следующий формат:

- Magic-последовательность («PMOC»);
- размер блока данных, используемый при упаковке (131072). Напомню, что в ARM используется система little-endian, а значит, что это число соответствует hex-значению 0x00000200 (см. рис. 3);
- блоки данных, содержащие:
  - размер блока (например, 1816);
  - запакованные данные обозначенного выше размера.

Это означает, что блок запакованных данных размером 1816 байт распакуется в 128 килобайт информации.

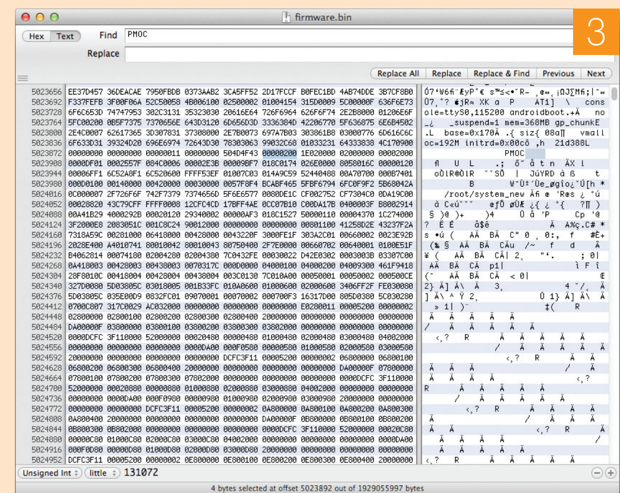


Рис. 3. Размер блока данных LZO-архива

INIT-СКРИПТЫ

Init-скрипты Android записываются на / в момент загрузки устройства и поэтому, хотя они и могут быть отредактированы непосредственно на устройстве, при следующей перезагрузке будут перезаписаны оригинальными файлами. Вероятнее всего, они располагаются в initrd, но исследования на эту тему не проводились.

ЗАКЛЮЧЕНИЕ

Наша жизнь — процесс. Закрытые программные системы — потемки. Процесс познания потемок и есть реверс-инжиниринг. Этот подход помог не только решить основную бизнес-задачу — выпустить кастомную прошивку, но и узнать больше о внутреннем устройстве Android в целом, что, несомненно, весьма интересно для настоящего хакера. Важно помнить, что реверс-инжиниринг — инструмент легальный и универсальный. Не будь его, мир никогда бы не узнал об опаснейших бэкдорах в прошивках ведущих производителей сетевого оборудования, об аппаратных «закладках» в микропроцессорах, об утечках данных в популярных интернет-приложениях. Если кто-то изобрел «черный ящик», то всегда найдется тот, кто сможет понять, как он работает. ⌶