**Android boot image format analysis.**

The boot.img and recovery.img formats are google-customized.

1. The full composition of the image.

A complete bootimage consists of headerboot, kernel, ramdisk, second stage (optional), X509 certificate (optional), and signature information (optional). The original boot.img only includes boot, header, kernel disk, second stage (optional), and in general second staga is not.

2. The original boot.img format includes the file header boot\_img\_hdr, kernel.gz (Linux kernel zImage), and the ramdisk.cpio.gz (root file system) structure as follows: s. ----------------- . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

boot header . . . 1

page .

. . . . . . . . ----------------- . . . . . . . . . . . . . . . . . . kernels. n

pages .

. . . . . . . ----------------- . . . . . . . . . . . . . . . . . . . Ramdisk . . . m

pages -----------------

. . . . . . . . . . . . . . . . . . . . . . . . . . . Second stage . . . o pages.

\*\* +-----------------+

Each part of the original boot.img is page-aligned. Where the boot header is fixed to occupy the size of 1 page. The actual size of the kernel, ramdisk, and second stage is indicated in boot header.

3. boot header structure.

The boot header is actually a structure struct boot\_img\_hdr, which is defined as follows.

struct boot\_img\_hdr

{

unsigned char magic[BOOT\_MAGIC\_SIZE]; /\* 8 byte \*/

unsigned kernel\_size; /\* size in bytes \*/

unsigned kernel\_addr; /\* physical load addr \*/

unsigned ramdisk\_size; /\* size in bytes \*/

unsigned ramdisk\_addr; /\* physical load addr \*/

unsigned second\_size; /\* size in bytes \*/

unsigned second\_addr; /\* physical load addr \*/

unsigned tags\_addr; /\* physical addr for kernel tags \*/

unsigned page\_size; /\* flash page size we assume \*/

unsigned unused[2]; /\* future expansion: should be 0 \*/

unsigned char name[BOOT\_NAME\_SIZE]; /\* asciiz product name \*/

unsigned char cmdline[BOOT\_ARGS\_SIZE];

unsigned id[8]; /\* timestamp / checksum / sha1 / etc \*/

};

Here are a few more important values. One is page\_size, this is the page size. As mentioned earlier, each part of boot.img is page-aligned, and the size of that page is recorded in the page\_size, usually 2048, or 2K alignment. kernel\_size is the original size of kernel.gz, when the pages are aligned, the kernel takes up more space than kernel\_size, and the excess is filled with 0. ramdisk\_size is the original size of the ramdisk, when the pages are aligned, the ramdisk takes up more space than ramdisk\_size, the excess is filled with 0. 4. Signature .

If boot.img needs to be verified at android startup to ensure its legitimacy, then boot.img needs to be signed after compilation is complete. Google's signature of boot.img is to encrypt the original boot.img using verity.pk8. And at the end of boot.img to add the corresponding verity.x509.pem certificate, as well as a piece of encrypted data (i.e. signature information)

The additional certificate is not directly pasted into the source code,""v.i., but by the conversion of the code. Verity.x509.pem is base64 encoded, and the additional certificate is in binary ASN1 format. The additional certificates are also page-aligned. The ASN1 format indicates the actual size of the certificate. The certificate begins with the size of the entire certificate, generally starting with 0x30 0x82 0xaa 0xbb, 30 indicates SEQUENCE, 0x82 means that the length occupies two 2 bytes, then the length of the certificate is 0xaabb. Pages are also aligned when the certificate is appended. (Here android 7.0 and android 5.x seem to be slightly different, android 5.x does not do alignment processing)

In addition to certificates, there are signatures. The signature is the content obtained after the original boot.img content is calculated and the hash value is encrypted. The specific hash algorithm and the encryption algorithm are specified in the certificate. The resulting signature length is related to hash and the encryption algorithm. For example, the certificate stipulates that sha1RSA2048, the length of the sha1 summary is 160bit, then the length of the RSA redaction is 256 bytes, so the length of the final signature information is 256 bytes, and the signature information does not need to be page-aligned.

https://source.android.com/security/verifiedboot/verified-boot?hl=zh-cn.

5. Mirror size.

The total size of boot.img is.

total = align(sizeof(struct boot\_img\_hdr))

+ align(kernel\_size)

+ align(ramdisk\_size)

+ align(second\_size)

+ align(cert\_size)

+ signature\_size;

**#####boot.img details.**

First, cmdline packaging process.

1, INTERNAL\_KERNEL\_CMDLINE is a string, from the BOARD\_KERNEL\_CMDLINE variables, and then stitched buildvariant s$ (VERITY\_KEYID) from the results.

Among them, BOARD\_KERNEL\_CMDLINE variable sdefined

vim build/core/Makefile in device/asr/aquilac\_phone/BoardConfig.

INTERNAL\_KERNEL\_CMDLINE := $(strip $(BOARD\_KERNEL\_CMDLINE) buildvariant=$(TARGET\_BUILD\_VARIANT) $(VERITY\_KEYID))

1

INTERNAL\_KERNEL\_CMDLINE added to the INTERNAL\_BOOTIMAGE\_ARGS:

INTERNAL\_BOOTIMAGE\_ARGS += --cmdline "$(INTERNAL\_KERNEL\_CMDLINE)"

1

2, then use the mkbootimg tool, packaged to make boot.img.

$(hide) $(MKBOOTIMG) $(INTERNAL\_BOOTIMAGE\_ARGS) $(INTERNAL\_MKBOOTIMG\_VERSION\_ARGS) $(BOARD\_MKBOOTIMG\_ARGS) --output $(INSTALLED\_BOOTIMAGE\_TARGET)

1

Among

them: (1), MKBOOTIMG is mkbootimg, is

the android boot.img tool (2), INTERNAL\_BOOTIMAGE\_ARGS, is the parameters required to make boot.img, including ramdisk, cmdline and so on.

/\* the boot image, which is a collection of other images. \*/

INTERNAL\_BOOTIMAGE\_ARGS := \

$(addprefix --second ,$(INSTALLED\_2NDBOOTLOADER\_TARGET)) \

--kernel $(INSTALLED\_KERNEL\_TARGET)

INTERNAL\_BOOTIMAGE\_ARGS += --ramdisk $(INSTALLED\_RAMDISK\_TARGET)

INTERNAL\_BOOTIMAGE\_ARGS += --base $(BOARD\_KERNEL\_BASE)

INTERNAL\_BOOTIMAGE\_ARGS += --cmdline "$(INTERNAL\_KERNEL\_CMDLINE)"

(3), BOARD\_MKBOOTIMG\_ARGS is the boot.img header\_version version number, on android 9.0, mandatory boot.img, recovery.img header\_version is 1, otherwise vts can not measure.

vim device/asr/aquilac\_phone/BoardConfig.mk

/\* The header\_version of boot.img and recovery.img must be 1 \*/

BOARD\_BOOTIMG\_HEADER\_VERSION:=1

BOARD\_MKBOOTIMG\_ARGS:=--header\_version $(BOARD\_BOOTIMG\_HEADER\_VERSION)

, INTERNAL\_MKBOOTIMG\_VERSION\_ARGS is not currently required, empty.

Second, boot.img format

below is the boot.imgheader format, can be seen in kernel/uboot/include/android\_image.h.

Among them, kernel\_addr, ramdisk\_addr, second\_addr, all indicate the position of the image in boot.img.

#define ANDR\_BOOT\_MAGIC "ANDROID!"

#define ANDR\_BOOT\_MAGIC\_SIZE 8

#define ANDR\_BOOT\_NAME\_SIZE 16

#define ANDR\_BOOT\_ARGS\_SIZE 512

struct andr\_img\_hdr {

char magic[ANDR\_BOOT\_MAGIC\_SIZE];

u32 kernel\_size; /\* size in bytes \*/

u32 kernel\_addr; /\* physical load addr \*/

u32 ramdisk\_size; /\* size in bytes \*/

u32 ramdisk\_addr; /\* physical load addr \*/

u32 second\_size; /\* size in bytes \*/

u32 second\_addr; /\* physical load addr \*/

u32 tags\_addr; /\* physical addr for kernel tags \*/

u32 page\_size; /\* flash page size we assume \*/

char name[ANDR\_BOOT\_NAME\_SIZE]; /\* asciiz product name \*/

char cmdline[ANDR\_BOOT\_ARGS\_SIZE];

u32 id[8]; /\* timestamp / checksum / sha1 / etc \*/

};

So where is the kernel or Image?

In fact, android requirements, boot.img offset s0x800, storage kernel. You can also see it by the macro definition below.

/\* Kernel loader address is where the kernel binary to be loaded. Before that, we have \*/

/\* Android header (0x800 long) and AIMG header (0x20 long). Be aware to adjust these \*/

/\* offsets when the header size is changed. \*/

#define CONFIG\_ASR\_BOOT 1

#define BOOTIMG\_HEAD\_SIZE 0x800

#define AIMG\_HEAD\_SIZE 0x20

#define AIMG\_TAIL\_SIZE 0x100

#define SIGNED\_BOOTIMG\_LOADADDR (KERNEL\_LOADADDR - BOOTIMG\_HEAD\_SIZE - AIMG\_HEAD\_SIZE)

#define BOOTIMG\_LOADADDR (KERNEL\_LOADADDR - BOOTIMG\_HEAD\_SIZE)

**android boot.img structure**

Android's boot.img includes boot header，kernel， ramdisk

First look at how Makefile produces our boot.img:

Boot mirror is not a normal file system, but a special Android custom format, consisting of file header information boot header, compressed

kernel, file system data ramdisk and second stage (optional), between them are filled with 0, can be seen from the mkbootimg.h file.

The specific structure of the file header information can be seen in system/core/mkbootimg/bootimg.h:

struct boot\_img\_hdr

{

unsigned char magic[BOOT\_MAGIC\_SIZE];

unsigned kernel\_size;

unsigned kernel\_addr;

unsigned ramdisk\_size;

unsigned ramdisk\_addr;

unsigned second\_size;

unsigned second\_addr;

unsigned tags\_addr;

unsigned page\_size;

unsigned unused[2];

unsigned char name[BOOT\_NAME\_SIZE]

unsigned char cmdline[BOOT\_ARGS\_SIZE]

unsigned id[8]; //Store timestamp, checksum, SHA encryption, etc.}

Boot, img file after skipping 4k file heads, including two gz packets, one boot.img-kernel.gz: Linux kernel, one boot.img-ramdisk.cpio.gz.

The approximate composition structure is as follows.

\*

\*\* +-----------------+

\*\* | boot header | 1 page

\*\* +-----------------+

\*\* | kernel | n pages

\*\* +-----------------+

\*\* | ramdisk | m pages

\*\* +-----------------+

\*\* | second stage | o pages

\*\* +-----------------+

Boot header s, including command line arguments, and so on, has an address of 000 -----0xFFF.

The ramdisk starts with 1F8B080000000.

Kernel is 0000A0E1 repeat s8 times at the beginning.

We need to pay attention to the boot header data structure, where we look at a few of the more important values, which are defined in boot/boardconfig.h, where different chips correspond to different boardconfigs under the vendor, where our values are (the physical address of kernel/ramdis/tags loaded- ram:

#define PHYSICAL\_DRAM\_BASE 0x00200000

#define KERNEL\_ADDR (PHYSICAL\_DRAM\_BASE + 0x00008000)

#define RAMDISK\_ADDR (PHYSICAL\_DRAM\_BASE + 0x01000000)

#define TAGS\_ADDR (PHYSICAL\_DRAM\_BASE + 0x00000100)

#define NEWTAGS\_ADDR (PHYSICAL\_DRAM\_BASE + 0x00004000)

These values correspond to the terms we mentioned at the beginning, such as kernel\_addr zTEXTADDR, RAMDISK\_ADDR is INITRD\_PHYS, and TAGS\_ADDR is PARAMS\_PHYS. Bootloader reads and kernelramdisk into the address defined above RAM from the boot.img partition, and then jumps to ZTEXTADDR to start executing.

The ramdisk image is the most basic small file system, which includes all the core files needed to initialize the system, such as the initialization init process and init.rc (can be used to set many system parameters) and other files. The following is a list of files contained in a typical ramdisk:

. / init.trout.rc

. /default.prop

. / proc

. / dev

. / init.rc

. / init

. / sys

. / init.goldfish.rc

. / sbin

. /sbin/adbd

. / system

. /data

If you want to separate you can use winhex to turn the boot. img is on.

Find the block in front of 0000A0E1 to 1F8B08000000tokeeping to keep as kernel.

Find 1F8B08000000 to the end of the file block to keep as ramdisk.img.

out/host/linux-x86/bin/mkbootimg --kernel out/target/product/msm7630\_surf/kernel --ramdisk out/target/product/msm7630\_surf/ramdisk.img --cmdline "console=ttyMSM1,115200n8 androidboot.hardware=qcom" --base 0x00200000 --pagesize 4096 --output out/target/product/msm7630\_surf/boot.img

According to the above command we can first look at the source file of the mkbootimg tool: system/core/mkbootimg.c. After reading it, we can see clearly the internal structure of boot.img, which is made up of boot -kernel/ramdisk/second stage, of which the first 3 items are required and the last one is optional. After mkbootimg analyzes the parameters, write to header, kernel, ramdisk .

header + padding + kernel + padding + ramdisk + padding + ...

4 \* 2, magic，Fixed as "ANDROID!"

4 \* 1, kernel length, little endian unsigned

4 \* 1, kernel Address, should be base + 0x00008000 (base for 0x200000)

4 \* 1, ramdisk Length, little end unsigned

4 \* 1, ramdisk Address, should be base + 0x01000000

4 \* 1, second stage Length, little end unsigned，for 0

4 \* 1, second stage Address, should bebase + 0x00f00000

4 \* 1, tags Address, should be base + 0x00000100

4 \* 1, page Size, little endian unsigned, for 2048 or 4096

4 x 2, unused, fixed

0x00 4 x 4,

board name, generally empty 4 x 128, Kernel command parameters, for mem?211M console-ttyMSM2,115200n8 androidboot.hardware-qcom console?ttyUSB-set0

androidboot.console?ttyUSB CONSOLE0 4 s 8, id, for sha and so on, actually write 0x00 padding,

the above part is 608 bytes, this part is added to

kernel\_size

the page\_size Make the kernel\_size to

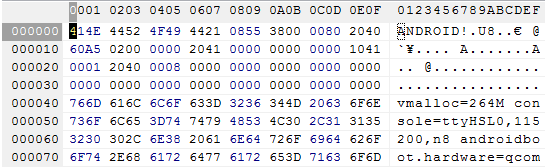
page\_size s 2

ramdisk\_size, ramdisk content padding,

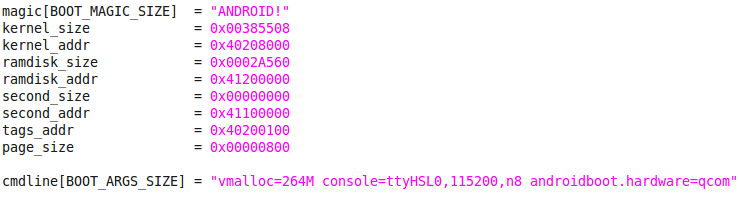
ramdisk to page\_size s

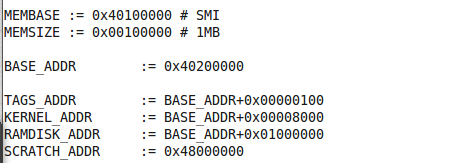
2 second\_size, secondadd content, generally 0 padd, second\_sise for page\_size, generally 0.

With boot.img look stoic.



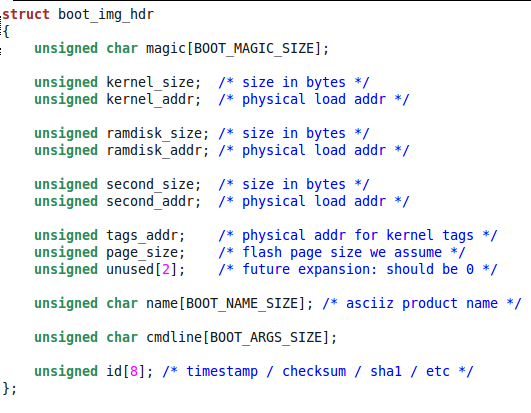
As a *result,*the value of the members in the boot\_img\_hdr is : .





TAGS\_ADDR as defined by target/?lt;you-platform/rules.mk above: 0x40200100, so boot\_linux(), is the incoming TAGS\_ADDR,

Then write the data to the tag, the structure of the tag is shown below.



Then enter the kernel's entry function: entry (0, machtype, tags)

**Android-boot.img partition details.**

* [Introduction to Android boot.img.](https://blog.csdn.net/linux_embedded/article/details/82494399#android-bootimg%E7%AE%80%E4%BB%8B)
* [boot.img file format.](https://blog.csdn.net/linux_embedded/article/details/82494399#bootimg%E6%96%87%E4%BB%B6%E6%A0%BC%E5%BC%8F)
* [boot.img custom.](https://blog.csdn.net/linux_embedded/article/details/82494399#bootimg%E5%AE%9A%E5%88%B6)
  + [boot.img unpack.](https://blog.csdn.net/linux_embedded/article/details/82494399#bootimg%E8%A7%A3%E5%8C%85)
  + [boot.img envelope.](https://blog.csdn.net/linux_embedded/article/details/82494399#bootimg%E5%B0%81%E5%8C%85)
  + [Resources.](https://blog.csdn.net/linux_embedded/article/details/82494399#%E8%B5%84%E6%BA%90)

Introduction to Android boot.img.

Android is different from the general embedded Linux system environment firmware composition (booloader s kernel s rootfs), it will be kernel, ramdisk (rootfs), second stage (dtb, kernel.logd, etc.) as a whole packaged into a boot.img file, and the boot.img file in the boot boot partition, boot Loader start-up, It reads boot.img, places information such as kernel, ramdisk, dtb, and so on in fixed memory, and then boots kernel. Visible, boot.img is quite important, if the file is corrupted, then the Android system will not start properly (can enter recovery mode for repair). For developers who need Android, Linux-driven, ramdisk customization, boot.img must master it, and here's a look at the boot.img file format.

boot.img file format.

| entity\_name. | entity\_size. |
| --- | --- |
| boot header. | 1 page. |
| kernel. | n pages. |
| Ramdisk. | m pages. |
| second stage. | o pages. |

The entity\_size is calculated as follows:

* n s (kernel\_size s page\_size - 1) / page\_size.
* m s (ramdisk\_size s page\_size - 1) / page\_size.
* o s (second\_size s page\_size - 1) / page\_size.

Note:  
1. All entities are page\_size  
aligned in flash 2 and kernel ramdisk are awyd (size! s 0)  
3. second is optional (second\_size s 0 -  
no second) 4. Load all element (kernel, ramdisk, second) at  
the specified physical address (kernel\_addr, etc)  
5. tags at tag\_addr. kernel\_args the  
sappend to the kernel the commandline in the tags.  
6. r0 , r1 , MACHINE\_TYPE,  
r2 , tags\_addr 7. if second\_size! . .  
0: jump to second\_addr else: jump to kernel\_addr.

boot.img custom.

For Android firmware developers, there  
are two development modes: 1. Based on Android source development.  
2. Based on Android existing firmware, customize boot.img, and then package boot.img into Android firmware.

The first is for people who are custom-developed at the Android framework layer, and the second is for linux kernel, ramdisk custom development. This section is intended for the second development model.

boot.img unpack.

First, customizing a boot.img needs to unpack it, the unpacking tool itself is using split\_boot, which is based on perl development, very simple to use.

split\_boot boot.img

The unpacked boot is then generated, as follows:

boot.img-kernel boot.img-ramdisk.cpio.gz boot.img-second.gz ramdisk

The boot.img in this example is taken from the firefly-rk3288 development board Android firmware, which includes the boot.img-second.gz file, which includes the dtb, logo.bmp file.

boot.img envelope.

When the files contained in boot.img are complete (for example, kernel, ramdisk, or dtb) are made, the individual files need to be repackaged as boot.img.

Two tools are needed here, one to package ramdisk into boot.img-ramdisk.cpio.gz, and one to package boot.img-kernel boot.img-ramdisk.cpio.gz.boot.boot.second.gz as boot.img.

First, show you how to package the ramdisk, and the package dinghofer is as follows:

mkbootfs ramdisk | gzip -n -f -c > boot.img-ramdisk.cpio.gz

Where ramdisk is the ramdisk directory that will be packaged.

After that, use the mkbooting tool to package all files into boot.img.

mkbooting \

--kernel boot.img-kernel \

--ramdisk boot.img-ramdisk.cpio.gz \

--second boot.img-second.gz \

--base 0x60400000 \

--pagesize 16384\

-o ../../boot.img

For boot.img packaging process, it can be implemented through a single script.

Resources.

download the tools used above including: split\_tool, mkbootfs, mkbooting, and so on.

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++