


从0到TrustZone第一篇：探究高通的 SEE（安全可执行环境）

转载 sheji105 于 2018-09-12 15:11:07 发布 976 收藏 8

分类专栏：安全 文章标签：trustzone tee android

安全

专栏收录该内容

1 订阅 9 篇文章

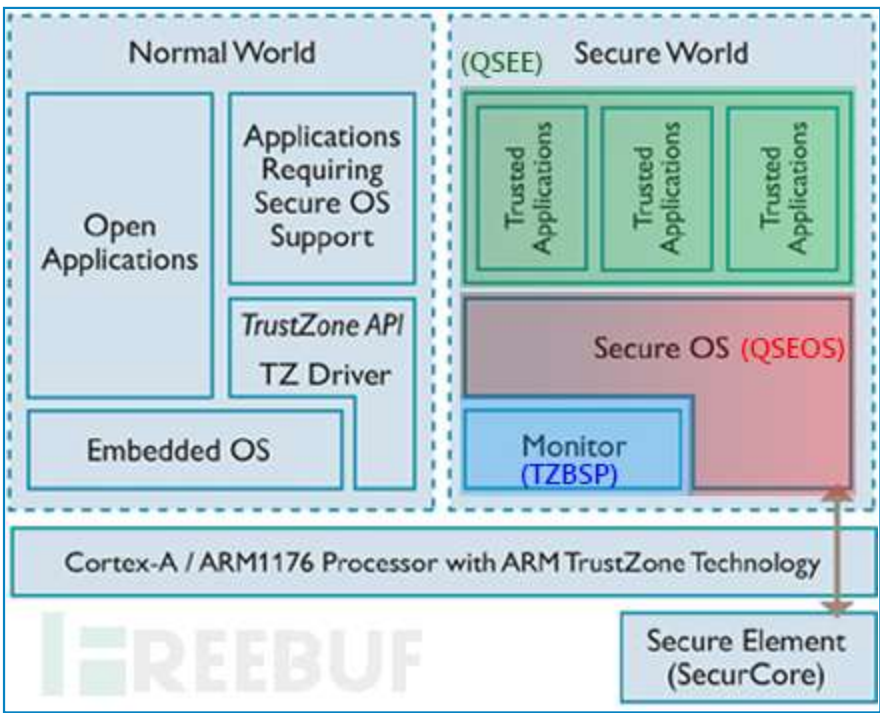
订阅专栏

转载：<http://www.freebuf.com/articles/system/103152.html>

在本篇文章中，我们将讨论高通安全执行环境（QSEE）。

之前讨论过，设备使用TrustZone的最主要的原因之一是它可以提供“可信执行环境（TEE）”，该环境可以保证不被常规操作系统干扰的计算，因此称为“可信”。

这是通过创建一个可以在TrustZone的“安全世界”中独立运行的小型操作系统实现的，该操作系统以系统调用（由TrustZone内核直接处理）的方式直接提供少数的服务。另外，TrustZone内核可以安全加载并执行小程序“Trustlets”，以便在扩展模型中添加“可信”功能。Trustlets程序可以为不安全（普通世界）的操作系统（本文指的是Android）提供安全的服务。



以下为设备上常用的Trustlets：

keymaster：实现由Android“keystore”守护进程提供的密钥管理API，它可以安全的生成和存储密钥，并运行用户使用这些密钥操作数据

widevine：实现Widevine DRM，提供安全的媒体播放

实际上，根据OEM和设备的不同有很多DRM-related trustlets，但是以上两种是通用的。

开始

下面通过对widevine模块的分析，来了解工作原理。

在设备固件中搜索widevine trustlet：

```
shell@shamu:/system/vendor/firmware $ find . | grep "widevine\"
./widevine.b00
./widevine.b01
./widevine.b02
./widevine.b03
./widevine.mdt
```

很显然，trustlet被分成多个文件..打开这些文件发现显示比较混乱..有些文件中包含代码、ELF头和元数据。在分解trustlet前，需要了解一下它的格式。我们可以挨个儿打开每个文件，尝试猜测内容的含义，或者查看用于加载该trustlet的代码路径。下面来都尝试一下。

加载TRUSTLET

为了从“普通世界”加载trustlet，应用程序可以使用libQSEECOM.so共享对象，输出函数“QSEECOM_start_app”：

```
* @param[in/out] handle The device handle
* @param[in] fname The directory and filename to load.
* @param[in] sb_size Size of the shared buffer memory for sending requests.
* @return Zero on success, negative on failure. errno will be set on
* error.
*/
int QSEECOM_start_app(struct QSEECOM_handle **clnt_handle, const char *path,
                     const char *fname, uint32_t sb_size);
```

遗憾的是，该函数的源代码是不可用的，因此我们需要使用逆向工程获取其实现代码。我们发现它可以执行以下操作：

打开/dev/qseecom设备，并调用一些ioctl函数来配置它

打开与trustlet相关的.mdt文件，并读取前0×34字节

使用.mdt文件中的0×34字节计算.bXX文件的数量

分配一个连续的缓冲区（使用ion），并将.mdt和.bXX文件复制到其中

最后，调用ioctl函数加载trustlet，使用已分配的缓冲区

但是目前仍然不清楚镜像如何加载，但是我们也获得了一些信息：

首先，数字0x34可能看起来很熟悉——这是32比特ELF头的大小，打开.mdt文件文件发现，第一个0x34字节确实是一个有效的ELF头：

0000h: 7F 45 4C 46 01 01 01 00 00 00 00 00 00 00 00 00 .ELF.....
0010h: 02 00 28 00 01 00 00 00 00 00 00 00 34 00 00 00 ..(.4...
0020h: 00 00 00 00 02 00 00 05 34 00 20 00 04 00 28 004.
0030h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0040h: 00 00 00 00 B4 00 00 00 00 00 00 00 00 00 00 07
0050h: 00 00 00 00 00 00 00 00 00 10 00 00 00 50 04 00P..

Inspector

Name	Value	Start	Size	Color
► char e_ident[16]		0h	10h	Fg: Bg:
ushort e_type	2	10h	2h	Fg: Bg:
ushort e_machine	40	12h	2h	Fg: Bg:
uint e_version	1	14h	4h	Fg: Bg:
uint e_entry	0	18h	4h	Fg: Bg:
uint e_phoff	52	1Ch	4h	Fg: Bg:
uint e_shoff	0	20h	4h	Fg: Bg:
uint e_flags	83886082	24h	4h	Fg: Bg:
ushort e_ehsize	52	28h	2h	Fg: Bg:
ushort e_phentsize	32	2Ah	2h	Fg: Bg:
ushort e_phnum	4	2Ch	2h	Fg: Bg:
ushort e_shentsize	40	2Eh	2h	Fg: Bg:
ushort e_shnum	0	30h	2h	Fg: Bg:
ushort e_shstrndx	0	32h	2h	Fg: Bg:

另外，QSEECOM_start_app函数使用比特于0x2C偏移的字，以便于计算.bXX文件的数量，对应上图中的ELF头中的e_phnum字段。

e_phnum字段通常用来指定程序头文件的数量，这就表示可能每个.bXX文件都包含独立的trustlet段。打开每个文件发现它们确实看起来像一段加载的程序。但是保险起见，我们还是得找到程序的头部（并检查是否与.bXX文件匹配）。

实际上，.mdt文件中接下来的几个数据库确实是程序头，对应每一个.bXX文件。

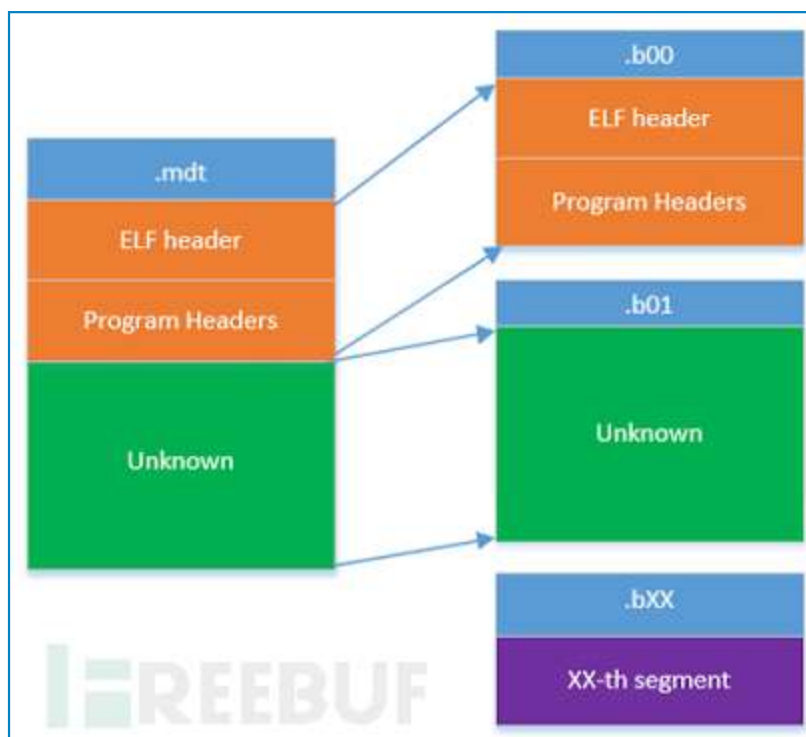
0000h:	7F 45 4C 46 01 01 01 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00	.ELF.....
0010h:	02 00 28 00 01 00 00 00 00 00 00 00 34 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 34 00 00 00	.. (.....4..
0020h:	00 00 00 00 02 00 00 05 34 00 20 00 04 00 28 00	00 00 00 00 00 00 00 00 00 00 00 00 04 00 28 004. ... (.
0030h:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0040h:	00 00 00 00 B4 00 00 00 00 00 00 00 00 00 00 07	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 07
0050h:	00 00 00 00 00 00 00 00 00 10 00 00 00 50 04 00	00 00 00 00 00 10 00 00 00 50 04 00 00 50 04 00P..
0060h:	00 50 04 00 A8 19 00 00 00 20 00 00 00 00 20 02	00 50 04 00 A8 19 00 00 00 20 00 00 00 00 20 02	.P.
0070h:	00 10 00 00 01 00 00 00 00 30 00 00 00 00 00 00	00 10 00 00 01 00 00 00 00 30 00 00 00 00 00 000.....
0080h:	00 00 00 00 30 C2 02 00 30 C2 02 00 05 00 00 A0	00 00 00 00 30 C2 02 00 30 C2 02 00 05 00 00 A00Ã..0Ã.....
0090h:	00 01 00 00 01 00 00 00 CC 0F 03 00 00 D0 02 00	00 01 00 00 01 00 00 00 CC 0F 03 00 00 D0 02 00î....Đ..
00A0h:	00 D0 02 00 A8 03 00 00 B8 77 01 00 06 00 00 30	00 D0 02 00 A8 03 00 00 B8 77 01 00 06 00 00 30	.Đ.w....0
00B0h:	00 10 00 00 00 00 00 00 03 00 00 00 00 00 00 00	00 10 00 00 00 00 00 00 03 00 00 00 00 00 00 00
00C0h:	28 50 04 00 80 19 00 00 80 00 00 00 A8 50 04 00	28 50 04 00 80 19 00 00 80 00 00 00 A8 50 04 00	(P..ë...ë...P..

Name	Value	Start	Size	Color
► char ELF header[52]		0h	34h	Fg: Bg:
► char phdr0[32]		34h	20h	Fg: Bg:
► char phdr1[32]		54h	20h	Fg: Bg:
► char phdr2[32]		74h	20h	Fg: Bg:
► char phdr3[32]		94h	20h	Fg: Bg:

正如之前猜想，它们的大小与.bXX文件文件完全匹配。

注意，上图中的前两个程序头看起来有些奇怪——它们都是NULL类型，表示它们是“保留的”，不会被加载到最终的ELF镜像中。更加奇怪的是，打开相应的.bXX文件发现第一个数据块包含与.mdt中相同的ELF头和程序头，第二个数据块中包含剩余的.mdt文件。

下图为根据目前所知的简单示意图：



需要注意的是，由于ELF头和程序头都包含在.mdt文件中，我们可以使用readelf命令快速转储trustlet中与程序头相关的信息：

```
>head CAChain
-----BEGIN CERTIFICATE-----
MIID5DCCAasygAwIBAgIBBTANBgkqhkiG9w0BAQUFADB9MQswCQYDVQQGEwJVUzET
MBEGA1UECBMKQ2FsaWZvcn5pYTESMBAGA1UEBxMjU2FuIERpZWdvMR0wGAYDVQQL
ExFDRE1BIFRlY2hub2xvZ2llczERMA8GA1UEChMIUVVBVENPTU0xFTjAUBgNVBAMT
DVFDVCBSb290IENBIDewHhcNMjQwNTIwMDA0NTIyWhcNMjQwODE5MTgzMDQ0WjCB
hZELMAkGA1UEBhMCVVMxEzARBgNVBAGTCkNhbgGlb3JuaWExEjAQBgNVBAcTCVNH
biBEaWVnbzEaMBGGA1UECxMRQ0RNQSBUZWNobm9sb2dpZXNxEtAPBgNVBAoTCFFV
QUxDT01NMSAwHgYDVQQDEXdRVUFRMQ09NTSBBdHRlc3RhZGlvb3B1BDQTCASAwDQYJ
KoZIhvcNAQEBBQADggENADCCAQgCggEBAMKqODEF76p9Ft8ZEHdn1gwJlUYl6Avp
QLbJJ18SbL4sbGc2E1jRRaiXwaQyLn+SGR3F0mgqL9S3rz5KtbTnnTGWfQ1jtRDRv
>openssl verify -CAfile CAChain cert1_PEM.cert
cert1_PEM.cert: OK
>openssl verify -CAfile CAChain cert2_PEM.cert
cert2_PEM.cert: OK
```

此时，我们从.mdt和.bXX文件中获得了所有创建完整和有效的ELF文件所需的信息；我们有ELF头和程序头，已经每个片段。我们只需要使用这些数据写一个小脚本就可以创建ELF文件。

可信TRUSTLETS的反射

现在，我们对trustlets如何组装成一个可执行文件已经有了基本的了解，但是还不清楚它怎么样验证。我们知道，.bXX文件中只是包含加载的片段，这就意味着，该数据也必须存在于.mdt文件中。

假设，要创建一个可信加载器，我们要怎么做？

一个通用的方法是使用hash-and-sign（基于CRHF和数字签名）。本质上，我们计算用于身份验证的数据的哈希值，并使用私钥（其对应的公钥加载器已知）对其进行签名。

因此，我们需要在.mdt文件中找到两个信息：

证书链

签名的二进制对象

接下来，让我们通过查找证书链开始。证书的格式有很多种，但是由于.mdt文件只包含二进制数据，我们可以猜想它可能是二进制格式，最常见的是DER。

这里有一个快速找到DER编码的证书的破解方法——它们通常以“ASN.1 SEQUENCE”（编码后为0x30 0x82）开始。那么我们需要做的就是，在.mdt文件中查找这两个字节，并将每个结果都保存在文件中。然后，只需要检查这些数据是否为使用“openssl”的结构良好的证书：


```
>head CACchain
-----BEGIN CERTIFICATE-----
MIID5DCCAsygAwIBAgIBBTANBgkqhkiG9w0BAQUFADB9MQswCQYDVQQGEwJVUzET
MBEGA1UECBMKQ2FsaWZvcn5pYTESMBAGA1UEBxMJU2FuIERpZWdvMR0wGAYDVQQL
ExFDRE1BIFRlY2hub2xvZ2llczERMA8GA1UEChMIUVVBVENPTU0xFTjAUBgNVBAMT
DVFDVCBSb290IENBIDewHhcNMDQwNTIwMDA0NTIyWhcNMjQwODE5MTgzMDQ0WjCB
hzELMAkGA1UEBhMCVVMxEzARBgNVBAgTCkNhbmGlb3JuaWExEjAQBgNVBAcTCVNH
biBEaWVnbzEaMBGGA1UECxMRQ0RNQSBUZWNobm9sb2dpZXMxETAPBgNVBAoTCFFV
QUxDT01NMSAwHgYDVQQDEXdRVUFGMQ09NTSBBdHRlc3RhZGlvb3RlY2VhZDQYJ
KoZIhvcNAQEBBQADggENADCCAQgCggEBAMKqODEF76p9Ft8ZEHdn1gwJlUYL6Avp
QLbJ18SbL4sbGc2E1jRRaiXwaQyLn+SGR3F0mgqL9S3rz5KtbTnnTGWfQ1jtRDRv
>openssl verify -CAfile CACchain cert1_PEM.cert
cert1_PEM.cert: OK
>openssl verify -CAfile CACchain cert2_PEM.cert
cert2_PEM.cert: OK
```

对于该链的可信根，通过查看该链的根证书发现，与验证高通的“安全引导”进程的引导链的其他部分的证书相同。目前已经有一些关于这种机制的研究，该验证通过匹配根证书的SHA256和一个特殊的值“OEM_PK_HASH”，该值在生产过程中被混淆到设备的QFuses中，并且在理论上是不可修改的，这就意味着建立一个这样的根证书需要对SHA256实施二次原像攻击。

现在，让我们回到.mdt文件——我们已经找到了证书链，现在需要的是签名。通常，私钥用于生成签名，公钥用于恢复签名数据。由于我们拥有该链最顶层证书的公钥，我们可以使用该值重新检查文件，尝试恢复每一个二进制对象。

但是，我们要怎样知道是否成功呢？

RSA是一个陷门置换家族——每一个相同比特的二进制对象都通过一个公共模N映射到相同大小的二进制对象中。然而，当RSA的公共模长度为2048比特时，多数哈希值会比这更短（SHA1为160比特，SHA256为265比特）。也就是当我们尝试使用该公钥解密二进制对象时，会出现很多“空”空间（例如，零字节）。但是我们有很大可能得到想要的签名（对于一个完全随机的排列，连续n个零比特的可能性为 2^{-n} ——几率非常小）。

我写了一个小程序（使用带有PKCS #1 v1.5填充的rsa_public_decrypt函数），用于加载证书链最顶层证书的公钥，并尝试恢复文件中的二进制对象。如果恢复的二进制对象以一串零字节结尾，那么输出结果。以下为运行结果：

```
luginimaine@luginimainebox:~/Desktop/TrustletSignatureVerification/TryDecrypt$
./trydec ../CERTS/pubkey1.pem ../widevine.mdt
Loaded key: 0x6fe910
i=348
4730B61D5C2D12FD729782DDC58C6E1257F8321719DBD265486FC9045838B86900000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
```

我们成功获取了一个签名。

更重要的是，该签名长度为265比特——这说明它可能是一个SHA256哈希。如果.mdt中存在一个SHA256，就说明可能有更多的SHA256：

00D0h: 00 01 00 00 A8 51 04 00 00 18 00 00 F8 38 92 8EQ.....E8'Z

00E0h: 68 A4 B4 44 56 63 BF 51 1C 63 AD AE A1 1E E7 19 k' DVczQ.c-01.C

00F0h: 18 F6 37 12 F8 99 6B 76 F9 45 6C CE 00 00 00 00 .07.f"kvuE1f....

0100h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

0110h: 00 00 00 00 00 00 00 00 00 00 00 00 F2 D3 09 6000.

0120h: 40 C4 DD C8 05 77 27 87 A8 B6 9E 2D 68 9A 29 59 @AYE.w'+`qz-hs)Y

0130h: 45 E0 BF F3 AD 2A D6 65 93 D9 28 C0 FE F6 0E 03 Ea;0-*0e"U(Ap0..

0140h: B8 E1 2D D8 6D 18 FB 16 46 EF 0F 4B 34 ED 21 4C .á-0m.û.Fi.K4i!L

0150h: B7 89 18 F8 B3 2F C9 A6 35 F6 03 DF EE EE EE EE .%.0³/É|50.Biiii

.....

Inspector

Name	Value	Start	Size	Color
► char b00_sha256[32]	É8'Zk'DVcz...	DCh	20h	Fg: Bg:
► char b01_sha256[32]		FCh	20h	Fg: Bg:
► char b02_sha256[32]		11Ch	20h	Fg: Bg:
► char b03_sha256[32]		13Ch	20h	Fg: Bg:

```
lmy48m/SYSTEM/vendor/firmware$ sha256sum widevine.b00 widevine.b01 widevine.b02 widevine.b03
c838928e6ba4b4445663bf511c63adaea11ee7191bf63712fe996b76f9456cce widevine.b00
d94e6f0779031b1ba892ef23ad20db21c45b0f872d9764f90ac378434e439838 widevine.b01
f2d3096040c4ddc805772787a8b69e2d689a295945e0bff3ad2ad66593d928c0 widevine.b02
fef60e03b8e12dd86d18fb1646ef0f4b34ed214cb78918f8b32fc9a635f603df widevine.b03
```

如上图，.bXX文件中的SHA256哈希值同样存储在.mdt文件中，并且是连续的。我们可能做一个可靠的猜测，这可能是用于生成之前的签名的数据。

注意，.b01文件的哈希值丢失了——为什么？记住.b01文件中包含处理ELF头和程序头之外的.mdt文件中的所有数据。由于该数据中包含以上签名，并且该数据又用于生成数据块文件的哈希值，这就会形成一个依赖循环。因此，说该数据块的哈希值不存在是有道理的。

到现在为止我们已经解码了.mdt文件中的所有数据，处理位于程序头后的小结构体。经观察，该结构体中只是简单地包含了“.mdt”的各个部分的指针和长度：

0000h:	7F 45 4C 46 01 01 01 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.ELF.....
0010h:	02 00 28 00 01 00 00 00 00 00 00 00 34 00 00 00	00 00 00 00 00 00 00 00 34 00 00 00 00 00 00 00	.. (.....4..
0020h:	00 00 00 00 02 00 00 05 34 00 20 00 04 00 28 00	00 00 00 00 00 00 00 00 00 00 00 00 04 00 28 004.
0030h:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0040h:	00 00 00 00 B4 00 00 00 00 00 00 00 00 00 00 07	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 07
0050h:	00 00 00 00 00 00 00 00 00 10 00 00 00 50 04 00	00 10 00 00 00 50 04 00 00 00 00 00 00 00 00 00P..
0060h:	00 50 04 00 A8 19 00 00 00 20 00 00 00 00 20 02	00 20 00 00 00 00 00 00 00 00 20 02 00 00 00 00	.P.
0070h:	00 10 00 00 01 00 00 00 00 30 00 00 00 00 00 00	00 30 00 00 00 00 00 00 00 00 00 00 00 00 00 000.....
0080h:	00 00 00 00 30 C2 02 00 30 C2 02 00 05 00 00 A0	30 C2 02 00 05 00 00 A0 00 00 00 00 00 00 00 000Ã..0Ã....
0090h:	00 01 00 00 01 00 00 00 CC 0F 03 00 00 D0 02 00	CC 0F 03 00 00 D0 02 00 00 00 00 00 00 00 00 00Ï....Đ..
00A0h:	00 D0 02 00 A8 03 00 00 B8 77 01 00 06 00 00 30	B8 77 01 00 06 00 00 30 00 00 00 00 00 00 00 00	.Đ.w....0
00B0h:	00 10 00 00 00 00 00 00 03 00 00 00 00 00 00 00	03 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00C0h:	28 50 04 00 80 19 00 00 80 00 00 00 A8 50 04 00	80 00 00 00 A8 50 04 00 00 00 00 00 00 00 00 00	(P...€...€...P..

Inspector

Name	Value	Start	Size	Color
► char ELF header[52]		0h	34h	Fg: Bg:
► char phdr0[32]		34h	20h	Fg: Bg:
► char phdr1[32]		54h	20h	Fg: Bg:
► char phdr2[32]		74h	20h	Fg: Bg:
► char phdr3[32]		94h	20h	Fg: Bg:

因此，我们已经解码了.mdt文件中的所有数据：



摩托罗拉HAB (High Assurance Boot)

尽管.mdt文件的格式对所有原始设备供应商通用，但是摩托罗拉则有些不同。

与我们之前看到的提供RSA签名不同，摩托罗拉设备中签名字段为空（以上展示的签名来自Nexus 5设备），签名如下：

B7	89	18	F8	B3	2F	C9	A6	35	F6	03	DF	EE	EE	EE	EE	·%·ø³/É!5ø.ßi	iiii
EE	EE	EE	EE	EE	EE	EE	EE	EE	EE	EE	EE	FF	FF	FF	FF	iiiiiiiiiiiiii	yyyy
FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	yyyyyyyyyyyy	yyyy
FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	yyyyyyyyyyyy	yyyy
FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	yyyyyyyyyyyy	yyyy
FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	yyyyyyyyyyyy	yyyy
FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	yyyyyyyyyyyy	yyyy
FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	yyyyyyyyyyyy	yyyy
FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	yyyyyyyyyyyy	yyyy
FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	yyyyyyyyyyyy	yyyy
FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	yyyyyyyyyyyy	yyyy
FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	yyyyyyyyyyyy	yyyy
FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	yyyyyyyyyyyy	yyyy
FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	yyyyyyyyyyyy	yyyy
FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	yyyyyyyyyyyy	yyyy
FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	yyyyyyyyyyyy	yyyy
FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	yyyyyyyyyyyy	yyyy
FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	yyyyyyyyyyyy	yyyy
FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	yyyyyyyyyyyy	yyyy
FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	yyyyyyyyyyyy	yyyy
FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	yyyyyyyyyyyy	iiii
EE	EE	EE	EE	EE	EE	EE	EE	EE	EE	EE	EE	30	82	04	2A	iiiiiiiiiiiiii	0,.*

那么这样的镜像怎样验证呢？

这是通过摩托罗拉调用HAB（High Assurance Boot）的机制完成的。该机制允许将整个.mdt文件的证书链和签名附加到文件的末尾，并使用HAB的专有格式编码：

1200h:	1D 4C 96 34 EF 44 80 49 C1 D0 01 0A 01 00 04 2E	.L-4IDEIÁD.....
1210h:	4F 3D 4D 6F 74 6F 72 6F 6C 61 20 49 6E 63 2C 20	O=Motorola Inc,
1220h:	4F 55 3D 4D 6F 74 6F 72 6F 6C 61 20 50 4B 49 2C	OU=Motorola PKI,
1230h:	20 43 4E 3D 48 41 42 20 43 41 20 36 33 37 52 3B	CN=HAB CA 637R;
1240h:	81 0F 6E 72 CB 8F 01 00 04 39 4F 3D 4D 6F 74 6F	..nrE....9O=Moto
1250h:	72 6F 6C 61 20 49 6E 63 2C 20 4F 55 3D 4D 6F 74	rola Inc, OU=Mot
1260h:	6F 72 6F 6C 61 20 50 4B 49 2C 20 43 4E 3D 43 53	orola PKI, CN=CS
1270h:	46 20 43 41 20 36 33 37 2D 31 3B 20 53 4E 3D 36	F CA 637-1; SN=6
1280h:	33 32 33 02 00 00 03 01 00 01 01 00 C6 13 64 D7	323.....E.d×
1290h:	1C 02 73 F4 7D 68 F2 FC 9A E0 47 08 5B BB 3D 0F	..sô}hõušaG.[>=.
12A0h:	A3 C7 C1 55 D5 B3 7A 7D 59 11 2B 8B FA AE 84 02	fÇAUÔ³z}Y.+<úö,,.
12B0h:	D1 61 57 58 D4 81 B0 9C 89 E7 EB 6E 99 DA 5F B7	ÑaWXÔ.°æñçén™Ú .
12C0h:	36 53 8A 1A B7 2F 33 E0 63 7A D8 DB 22 8E CC AA	6SŠ.·/3áčzØŮ"Žì*
12D0h:	7F 2A 45 C7 DA 54 8F 47 5B 0E D6 E0 1F 9F A8 36	.*EÇŮT.G[.Öä.ÿ"6
12E0h:	64 F7 DA CA 3F 9B F5 65 E8 44 7F 5B 56 18 0E 3F	d÷ŮÊ?>öëèD.[V..?

Inspector

Name	Value	Start	Size	Color
▶ char string_identifier[3]		120Ch	3h	Fg: Bg:
char cert_name_len	46 '.'	120Fh	1h	Fg: Bg:
▶ char cert_name[46]	O=Motorola I...	1210h	2Eh	Fg: Bg:
▶ char unk1[8]		123Eh	8h	Fg: Bg:
▶ char string_identifier[3]		1246h	3h	Fg: Bg:
char issuer_name_len	57 '9'	1249h	1h	Fg: Bg:
▶ char issuer_name[57]	O=Motorola I...	124Ah	39h	Fg: Bg:
▶ char byte_field_identifier[2]		1283h	2h	Fg: Bg:
▶ char exponent_length[2]		1285h	2h	Fg: Bg:
▶ char exponent[3]		1287h	3h	Fg: Bg:
▶ char modulus_length[2]		128Ah	2h	Fg: Bg:
▶ char modulus[256]		128Ch	100h	Fg: Bg:

有关该机制的详细信息，可参看[Tal Aloni调研](#)。简而言之，就是.mdt文件使用证书链顶端的密钥哈希编码并签名，该链的根证书使用在引导程序阶段被硬编码的Super Root Key进行验证。

TRUSTLET的生命周期

在以上的验证程序之后，TrustZone内核将trustlet程序段加载到“普通世界”无法访问的安全存储区域（secapp-region），并为其分配ID。

随后，内核切换到“安全世界”用户模式，并运行trustlet的入口函数：

```
void __fastcall entry_func(int a1, int handled_func)
{
    char v2; // zf@0
    int handler_func; // r0@5

    if ( !v2 || handled_func != 1 )
        register_app(255, handled_func);
    get_stack_size();
    get_stack_location();
    get_app_name();
    handler_func = (int)get_handler_function();
    register_app(0, handler_func);
}
```

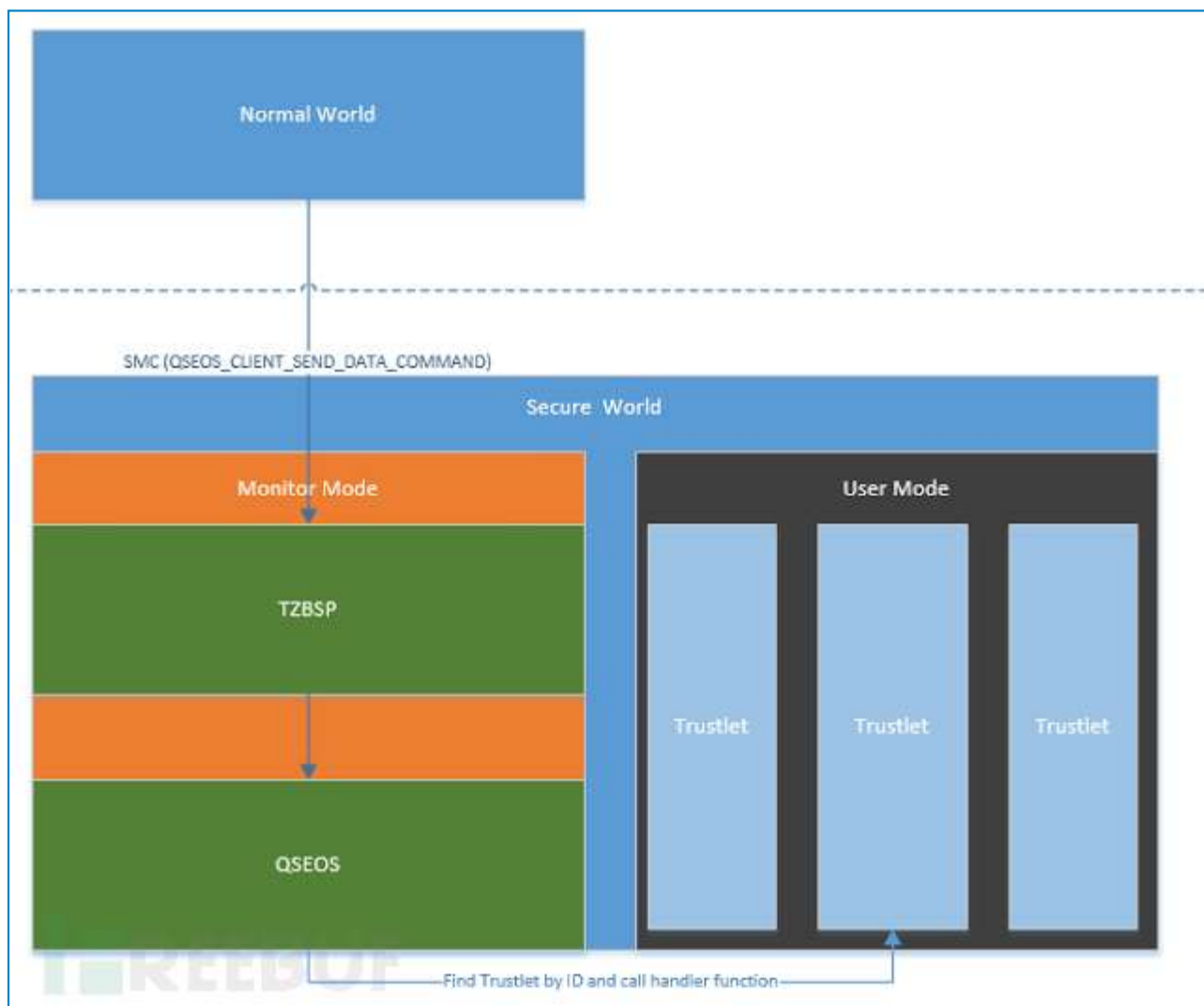
如上图，trustlet会使用“handler”函数在TrustZone内核中自动完成注册，注册完成后，trustlet会将控制权重新交给TrustZone内核，完成加载进程。

一旦trustlet加载完成，“普通世界”就可以通过调用SCM

（QSEOS_CLIENT_SEND_DATA_COMMAND，其中包含已加载trustlet ID和请求响应缓冲区）向trustlet发送命令，如下：

```
__packed struct qseecom_client_send_data_ireq {
    uint32_t qsee_cmd_id;
    uint32_t app_id;
    void *req_ptr;
    uint32_t req_len;
    void *rsp_ptr; /* First 4 bytes should always be the return status */
    uint32_t rsp_len;
};
```

TrustZone内核（TZBSP）接收到SCM调用后，将其映射到QSEOS，查找给定ID的应用程序，调用“handler”函数，处理请求。



后续

现在我们对trustlets及其加载有了一定的了解，接下来我们就可以发起攻击。在下一篇文章中，我们将在一个热门的trustlet中挖漏洞，并利用该漏洞在QSEE中执行代码。