加密固件之依据老固件进行解密

作者：OneShell@知道创宇 404 实验室 时间：2021 年 7 月 27 日

作者：OneShell@知道创宇 404 实验室
时间：2021 年 7 月 27 日

IoT 漏洞分析最为重要的环节之一就是获取固件以及固件中的文件系统。固件获取的方式也五花八门，硬核派有直接将 flash 拆下来到编程器读取，通过硬件调试器 UART/SPI、JTAG/SWD 获取到控制台访问；网络派有中间人攻击拦截 OTA 升级，从制造商的网页进行下载；社工派有假装研究者（学生）直接向客服索要，上某鱼进行 PY。有时候千辛万苦获取到固件了，开开心心地使用 `binwalk -Me` 一把梭哈，却发现，固件被加密了，惊不惊喜，刺不刺激。

如下就是针对如何对加密固件进行解密的其中一个方法：回溯未加密的老固件，从中找到负责对固件进行解密的程序，然后解密最新的加密固件。此处做示范使用的设备是前几天爆出存在漏洞的路由器 D-Link DIR 3040 US，固件使用的最新加密版本 1.13B03 (https://support.dlink.com/productinfo.aspx?m=DIR-3040-US)，老固件使用的是已经解密固件版本 1.13B02 (https://support.dlink.com/productinfo.aspx?m=DIR-3040-US)。

判断固件是否已经被加密

一般从官网下载到固件的时候，是先以 zip 等格式进行了一次压缩的，通常可以先正常解压一波。

```
$ tree -L 1
.
├── DIR3040A1_FW112B01_middle.bin
├── DIR3040A1_FW113B03.bin
└── DIR-3040_REVA_RELEASE_NOTES_v1.13B03.pdf
```

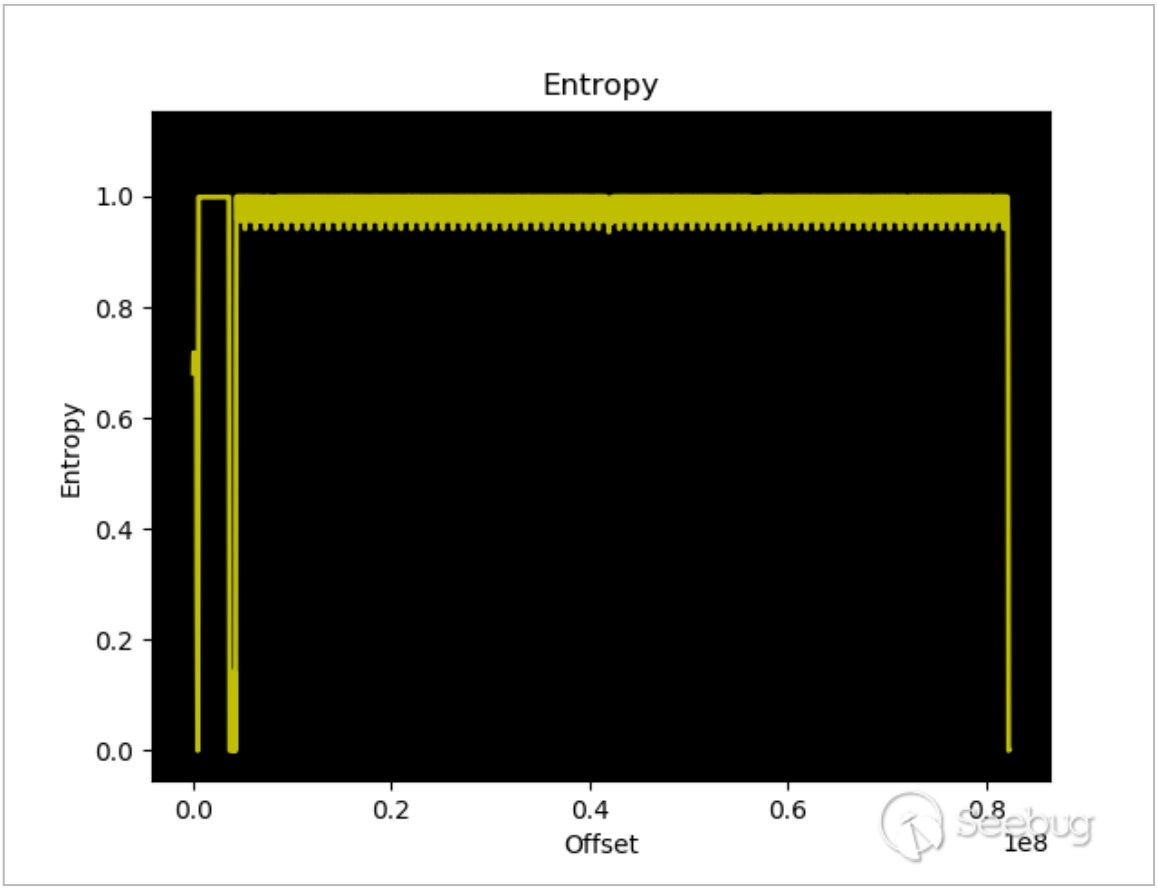
使用 binwalk 查看一下固件的信息，如果是未加密的固件，通常可以扫描出来使用了何种压缩算法，以常见的嵌入式文件系统 squashfs 为例，比较常见的有

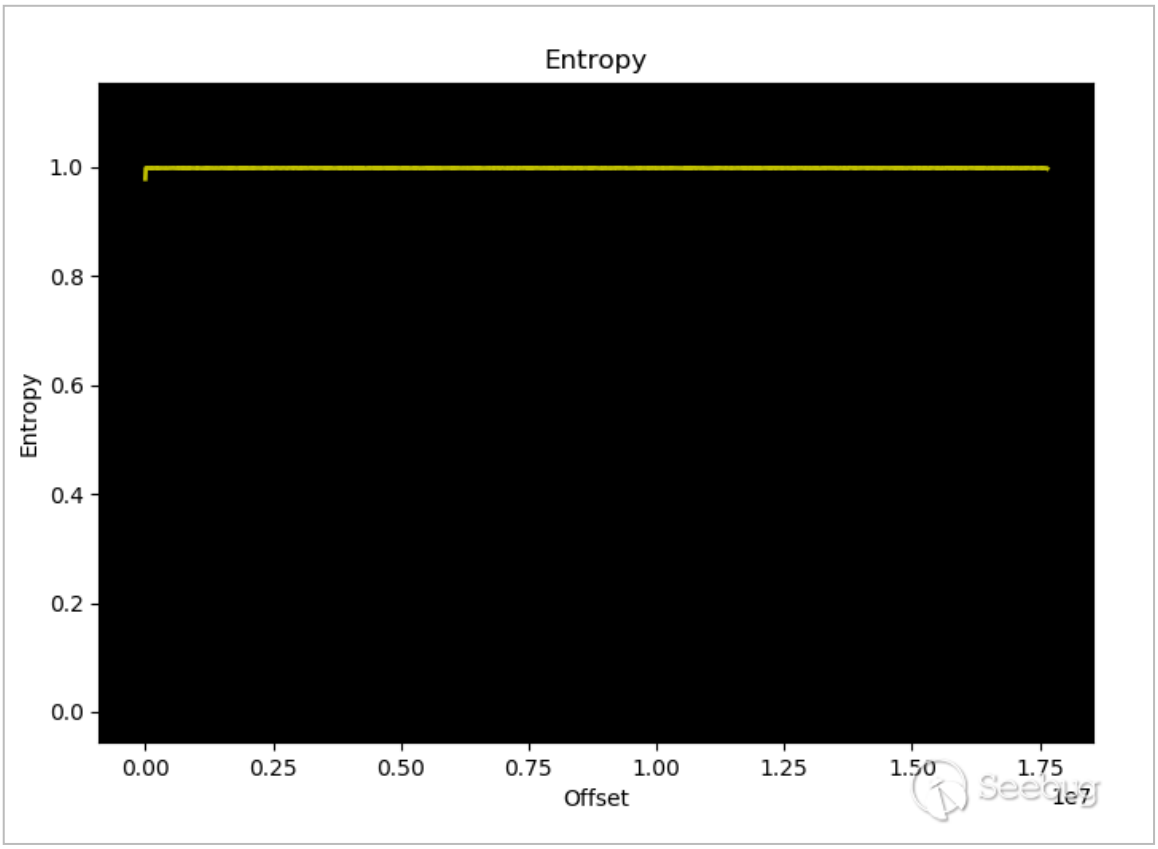
了何种压缩算法。以常见的嵌入式文件系统 squash-fs 为例，比较常见的有 LZMA、LZO、LAMA2 这些。如下是使用 binwalk 分别查看一个未加密固件（netgear）和加密固件（DIR 3040）信息。

```
$ binwalk GS108Tv3_GS110TPv3_GS110TPP_V7.0.6.3.bix

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
--
64            0x40            LZMA compressed data, properties: 0x5D, dictiona
ry size: 67108864 bytes, uncompressed size: -1 bytes
```

```
$ binwalk DIR3040A1_FW113B03.bin

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
--
```

还有一种方式就是查看固件的熵值。熵值是用来衡量不确定性，熵值越大则说明固件越有可能被加密或者压缩了。这个地方说的是被加密或者压缩了，被压缩的情况也是会让熵值变高或者接近 1 的，如下是使用 `binwalk -E` 查看一个未加密固件（RAX200）和加密固件（DIR 3040）。可以看到，RAX200 和 DIR 3040 相对比，不像后者那样直接全部是接近 1 了。





找到负责解密的可执行文件

接下来是进入正轨了。首先是寻找到老固件中负责解密的可执行文件。基本逻辑是先从 HTML 文件中找到显示升级的页面，然后在服务器程序例如此处使用的是 lighttpd 中去找到何处进行了调用可执行文件下载新固件、解密新固件，这一步也可能是发生在调用的 CGI 中。

使用 find 命令定位和升级相关的页面。

```
$ find . -name "*htm*" | grep -i "firmware"
./etc_ro/lighttpd/www/web/MobileUpdateFirmware.html
./etc_ro/lighttpd/www/web/UpdateFirmware.html
./etc_ro/lighttpd/www/web/UpdateFirmware_e.html
./etc_ro/lighttpd/www/web/UpdateFirmware_Multi.html
```

```
./etc_ro/lighttpd/www/web/UpdateFirmware_Simple.html
```

然后现在后端 lighttpd 中去找相关字符串，似乎没有结果呢，那么猜测可能发生在 CGI 中。

```
$ find . -name "*httpd*" | xargs strings | grep "firm"
strings: Warning: './etc_ro/lighttpd' is a directory
```

从 CGI 程序中查找，似乎运气不错，，，直接就定位到了，结果过多就只展示了最有可能的结果。Bingo！似乎已经得到了解密固件的程序，img、decrypt。

```
$ find . -name "*cgi*" | xargs strings | grep -i "firm"
/bin/imgdecrypt /tmp/firmware.img
```

仿真并解密固件

拿到了解密程序，也知道解密程序是怎么输入参数运行的，这个时候可以尝试对直接使用 qemu 模拟解密程序跑起来，直接对固件进行解密。最好保持解密可执行文件在老版本固件文件系统的位置不变，因为不确定是否使用相对或者绝对路径引用了什么文件，例如解密公私钥。

先查看可执行文件的运行架构，然后选择对应 qemu 进行模拟。

```
$ file bin/imgdecrypt
bin/imgdecrypt: ELF 32-bit LSB executable, MIPS, MIPS32 rel2 version 1 (SYSV),
dynamically linked, interpreter /lib/ld-uClibc.so.0, stripped
$ cp $(which qemu-mipsel-static) ./usr/bin
$ sudo mount -t proc /proc proc/
$ sudo mount --rbind /sys sys/
$ sudo mount --rbind /dev/ dev/
```

```
$ sudo chroot . qemu-mipsel-static /bin/sh


BusyBox v1.22.1 (2020-05-09 10:44:01 CST) built-in shell (ash)
Enter 'help' for a list of built-in commands.

/ # /bin/imgdecrypt tmp/DIR3040A1_FW113B03.bin
key:C05FBF1936C99429CE2A0781F08D6AD8
/ # ls -a tmp/
..                      .firmware.orig          .                       DIR304
0A1_FW113B03.bin
/ #
```

那么就解压出来了，解压到了 tmp 文件夹中，.firmware.orig 文件。这个时候使用 binwalk 再次进行查看，可以看到已经被成功解密了。

```
$ binwalk .firmware.orig

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
--
0             0x0             uImage header, header size: 64 bytes, header CR
C: 0x7EA490A0, created: 2020-08-14 10:42:39, image size: 17648005 bytes, Data
Address: 0x81001000, Entry Point: 0x81637600, data CRC: 0xAEF2B79F, OS: Linux,
CPU: MIPS, image type: OS Kernel Image, compression type: lzma, image name: "L
inux Kernel Image"
160           0xA0            LZMA compressed data, properties: 0x5D, dictiona
ry size: 33554432 bytes, uncompressed size: 23083456 bytes
1810550       0x1BA076        PGP RSA encrypted session key - keyid: 12A6E329
67B9887A RSA (Encrypt or Sign) 1024b
14275307      0xD9D2EB        Cisco IOS microcode, for "z"
```

加解密逻辑分析（重点）

关于固件安全开发到发布的一般流程

如果要考虑到固件的安全性，需要解决的一些痛点基本上是：

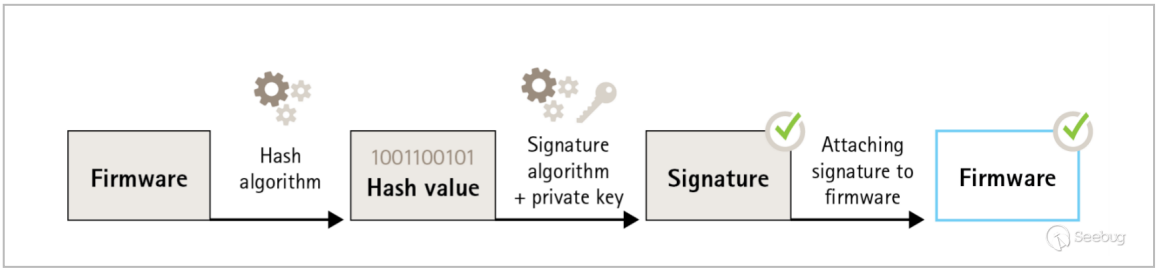
- 机密性：通过类似官网的公开渠道获取到解密后的固件
- 完整性：攻击者劫持升级渠道，或者直接将修改后的固件上传到设备，使固件升级

对于机密性，从固件的源头、传输渠道到设备三个点来分析。首先在源头，官网上或者官方 TFP 可以提供已经被加密的固件，设备自动或手动检查更新并从源头下载，下载到设备上后进行解密。其次是渠道，可以采用类似于 HTTPS 的加密传输方式来对固件进行传输。但是前面两种方式终归是要将固件下载到设备中。

如果是进行简单的加密，很常见的一种方式，尤其是对于一些低端嵌入式固件，通

常使用了硬编码的对称加密方式，例如 AES、DES 之类的，还可以基于硬编码的字符串进行一些数据计算，然后作为解密密钥。这次分析的 DIR 3040 就是采用的这种方式。

对于完整性，开发者在一开始可以通过基于自签名证书来实现对固件完整性的校验。开发者使用私钥对固件进行签名，并把签名附加到固件中。设备在接受安装时使用提前预装的公钥进行验证，如果检测到设备完整性受损，那么就拒绝固件升级。签名的流程一般不直接对固件本身的内容进行签名，首先计算固件的 HASH 值，然后开发者使用私钥对固件 HASH 进行签名，将签名附加到固件中。设备在出厂时文件系统中就被预装了公钥，升级通过公钥验证签名是否正确。



加解密逻辑分析

既然到这个地方了，那么顺便进去看一看解密程序是如何进行运作的。从 IDA 的符号表中可以看到，使用到了对称加密 AES、非对称加密 RSA 和哈希 SHA512，是不是对比上面提到的固件安全开发到发布的流程，心中大概有个数了。

首先我们进入 main 函数，可以知道，这个解密程序 imgdecrypt 实际上也是具有加密功能的。这里提一下，因为想要把整个解密固件的逻辑都撸一撸，可能会在文章里面贴出很多的具体函数分析，那么文章篇幅就会有点长，不过最后会进行一个流程的小总结，希望看的师傅不用觉得啰嗦。

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  int result; // $v0

  if ( strstr(*argv, "decrypt", envp) )
    result = decrypt_firmare(argc, (int)argv);
  else
    result = encrypt_firmare(argc, argv);
  return result;
}
```

下一步继续进入到函数 decrypt_firmare 中，这个地方结合之前仿真可以知道：

argc=2，argv = 参数字符串地址。首先是进行一些参数的初始化，例如 aes_key、公钥的存储地址 pubkey_loc。

接下来是对输入参数数量和参数字符串的判定，输入参数数量从 2 开始判定，结合之前的仿真，那么 argc=2，第一个是程序名，第二个是已加密固件地址。

然后在 004021AC 地址处的函数 check_rsa_cert，该函数内部逻辑也非常简单，基本就是调用 RSA 相关的库函数，读取公钥并判定公钥是否有效，有效则将读取到的 RSA 对象保存在 dword_413220。检查成功后，就进入到 004025A4 地址

处的函数 aes_cbc_crypt 中。这个函数的主要作用就是根据一个固定字符串 0123456789ABCDEF 生成密钥，是根据硬编码生成的解密密钥，因此每次生成并打印出来的密钥是相同的，此处密钥用变量 aes_key 表示。

```
int __fastcall decrypt_firmare(int argc, int argv)
{
  int result; // $v0
  const char *pubkey_loc; // [sp+18h] [-1Ch]
  int i; // [sp+1Ch] [-18h]
  int aes_key[5]; // [sp+20h] [-14h] BYREF

  qmemcpy(aes_key, "0123456789ABCDEF", 16);
  pubkey_loc = "/etc_ro/public.pem";
  i = -1;
  if ( argc >= 2 )
  {
    if ( argc >= 3 )
      pubkey_loc = *(const char **)(argv + 8);
    if ( check_rsa_cert((int)pubkey_loc, 0) )   // 读取公钥并进行保存RSA对象到dwor
d_413220中
    {
      result = -1;
    }
    else
    {
      aes_cbc_crypt((int)aes_key);              // 生成aes_key
      printf("key:");
      for ( i = 0; i < 16; ++i )
        printf("%02X", *((unsigned __int8 *)aes_key + i));// 打印出key
      puts("\r");
      i = actual_decrypt(*(_DWORD *)(argv + 4), (int)"/tmp/.firmware.orig", (i
nt)aes_key);
      if ( !i )
      {
        unlink(*(_DWORD *)(argv + 4));
        rename("/tmp/.firmware.orig", *(_DWORD *)(argv + 4));
      }
      RSA_free(dword_413220);
```

```
      result = i;
    }
  }
  else
  {
    printf("%s <sourceFile>\r\n", *(const char **)argv);
    result = -1;
  }
  return result;
}
```

接下来就是真正的负责解密和验证固件的函数 actual_decrypt，位于地址 00401770 处。在分析这个函数的时候，我发现 IDA 的 MIPS32 在反编译处理函数的输入参数的时候，似乎会把数值给弄错了，，，比如 fun(a + 10)，可能会反编译成 fun(a + 12)。已经修正过函数参数数值的反编译代码就放在下面，代码分析也全部直接放在注释中了。

```
int __fastcall actual_decrypt(int img_loc, int out_image_loc, int aes_key)
{
  int image_fp; // [sp+20h] [-108h]
  int v5; // [sp+24h] [-104h]
  _DWORD *MEM; // [sp+28h] [-100h]
  int OUT_MEM; // [sp+2Ch] [-FCh]
  int file_blocks; // [sp+30h] [-F8h]
  int v9; // [sp+34h] [-F4h]
  int i; // [sp+38h] [-F0h]
  int out_image_fp; // [sp+3Ch] [-ECh]
  int data1_len; // [sp+40h] [-E8h]
  int data2_len; // [sp+44h] [-E4h]
  _DWORD *IN_MEM; // [sp+48h] [-E0h]
  char hash_buf[68]; // [sp+4Ch] [-DCh] BYREF
  int image_info[38]; // [sp+90h] [-98h] BYREF

  image_fp = -1;
  out_image_fp = -1;
  v5 = -1;
  MEM = 0;
  OUT_MEM = 0;
  file_blocks = -1;
  v9 = -1;
  // 这个hashbuf用于存储SHA512的计算结果，在后面比较会一直被使用到
  memset(hash_buf, 0, 64);
  data1_len = 0;
  data2_len = 0;
  memset(image_info, 0, sizeof(image_info));
  IN_MEM = 0;
  // 通过stat函数读取加密固件的相关信息写入结构体到image_info，最重要的是文件大小
  if ( !stat(img_loc, image_info) )
  {
    // 获取文件大小
    file_blocks = image_info[13];
    // 以只读打开加密固件
    image_fp = open(img_loc, 0);
    if ( image_fp >= 0 )
    {
```

```
// 将加密固件映射到内存中
MEM = (_DWORD *)mmap(0, file_blocks, 1, 1, image_fp, 0);
if ( MEM )
{
  // 以O_RDWR | O_NOCTTY获得解密后固件应该存放的文件描述符
  out_image_fp = open(out_image_loc, 258);
  if ( out_image_fp >= 0 )
  {
    v9 = file_blocks;
    // 比较写入到内存的大小和固件的真实大小是否相同
    if ( file_blocks - 1 == lseek(out_image_fp, file_blocks - 1, 0) )
    {

      write(out_image_fp, &unk_402EDC, 1);
      close(out_image_fp);
      out_image_fp = open(out_image_loc, 258);
      // 以加密固件的文件大小，将待解密的固件映射到内存中，返回内存地址OUT_MEM
      OUT_MEM = mmap(0, v9, 3, 1, out_image_fp, 0);
      if ( OUT_MEM )
      {
        IN_MEM = MEM;                       // 重新赋值指针
        // 检查固件的Magic，通过查看HEX可以看到加密固件的开头有SHRS魔数
        if ( check_magic((int)MEM) )        // 比较读取到的固件信息中含有SHRS
        {
          // 获得解密后固件的大小
          data1_len = htonl(IN_MEM[2]);
          data2_len = htonl(IN_MEM[1]);
          // 从加密固件的1756地址起，计算data1_len个字节的SHA512，也就是解密后固件大小的消息摘要，并保存到hash_buf
          sub_400C84((int)(IN_MEM + 0x6dc), data1_len, (int)hash_buf);
          // 比较原始固件从156地址起，64个字节大小，和hash_buf中的值进行比较，也就是和加密固件头中预保存的真实加密固件大小的消息摘要比较
          if ( !memcmp(hash_buf, IN_MEM + 0x9c, 64) )
          {
            // AES对加密固件进行解密，并输出到OUT_MEM中
            // 这个地方也可以看出从加密固件的1756地址起就是真正被加密的固件数据，前面都是一些头部信息
            // 函数逻辑比较简单，就是AES加解密相关，从保存在固件头IN_MEM + 0xc获取解密密钥
            sub_40107C((int)(IN_MEM + 0x6dc), data1_len, aes_key, IN_MEM + 0xc, OUT_MEM);
            // 计算解密后固件的SHA_512消息摘要
            sub_400C84(OUT_MEM, data2_len, (int)hash_buf);
            // 和存储在原始加密固件头，从92地址开始、64字节的SHA512进行比较
            if ( !memcmp(hash_buf, IN_MEM + 0x5c, 64) )
            {
              // 获取解密固件+aes_key的SHA512
              sub_400D24(OUT_MEM, data2_len, aes_key, (int)hash_buf);
              // 和存储在原始固件头，从28地址开始、64字节的SHA512进行比较
              if ( !memcmp(hash_buf, IN_MEM + 0x1c, 64) )
              {
                // 使用当前文件系统内的公钥，通过RSA验证消息摘要和签名是否匹配
```

```
                    if ( sub_400E78((int)(IN_MEM + 0x5c), 64, (int)(IN_MEM +
0x2dc), 0x200) == 1 )
                    {
                      if ( sub_400E78((int)(IN_MEM + 0x9c), 64, (int)(IN_MEM
+ 0x4dc), 0x200) == 1 )
                        v5 = 0;
                      else
                        v5 = -1;
                    }
                    else
                    {
                      v5 = -1;
                    }
                  }
                  else
                  {
                    puts("check sha512 vendor failed\r");
                  }
                }
                else
                {
                  printf("check sha512 before failed %d %d\r\n", data2_len,
 data1_len);
                  for ( i = 0; i < 64; ++i )
                    printf("%02X", (unsigned __int8)hash_buf[i]);
                  puts("\r");
                  for ( i = 0; i < 64; ++i )
                    printf("%02X", *((unsigned __int8 *)IN_MEM + i + 92));
                  puts("\r");
                }
              }
              else
              {
                puts("check sha512 post failed\r");
              }
            }
            else
            {
              puts("no image matic found\r");
            }
          }
        }
      }
    }
  }
  if ( MEM )
    munmap(MEM, file_blocks);
  if ( OUT_MEM )
    munmap(OUT_MEM, v9);
  if ( image_fp >= 0 )
    close(image_fp);
  if ( image_fp >= 0 )
```

```
    close(image_fp);
  return v5;
}
```

概述 DIR 3040 的固件组成以及解密验证逻辑

从上面最关键的解密函数逻辑分析中，可以知道如果仅仅是解密相关，实际上只用到了 AES 解密，而且还是使用的硬编码密钥（通过了一些计算）。只是看上面的解密 + 验证逻辑分析，对整个流程可能还是会有点混乱，下面就说一下加密固件的文件结构和总结一下上面的解密 + 验证逻辑。

先直接给出加密固件文件结构的结论，只展现出重要的 Header 内容，大小 1756 字节，其后全部是真正的被加密固件数据。

起始地址	长度（Bytes）	作用
0：0x00	4	魔数：SHRS
4：0x4	4	解密固件的大小，带填充
8：0x8	4	解密固件的大小，不带填充
12：0xC	16	AES_128_CBC 解密密钥
28：0x1C	64	解密后固件 + KEY 的 SHA512 消息摘要
92：0x5C	64	解密后固件的 SHA512 消息摘要
156：0x9C	64	加密固件的 SHA512 消息摘要
220：0xDC	512	未使用
732：0x2DC	512	解密后固件消息摘要的数字签名
1244：0x4DC	512	加密后固件消息摘要的数字签名

结合上面的加密固件文件结构，再次概述一下解密逻辑：

1. 判断加密固件是否以 Magic Number：SHRS 开始。
2. 判断（加密固件中存放的，真正被加密的固件数据大小的 SHA512 消息摘要），和，（去除 Header 之后，数据的 SHA512 消息摘要）。

这一步是通过验证固件的文件大小，判定是否有人篡改过固件，如果被篡改，解密失败。

1. 读取保存在 Header 中的 AES 解密密钥，对加密固件数据进行解密
2. 计算（解密后固件数据的 SHA512 消息摘要），和（预先保存在 Header 中的，解密后固件 SHA512 消息摘要）进行对比

中的、解密后固件 SHA512 消息摘要）进行对比

3. 计算（解密固件数据 + 解密密钥的、SHA512 消息摘要），和（预先保存在 Header 中的、解密后固件数据 + 解密密钥的、SHA512 消息摘要）进行对比

4. 使用保存在当前文件系统中的 RSA 公钥，验证解密后固件的消息摘要和其签名是否匹配

5. 使用保存在当前文件系统中的 RSA 公钥，验证加密后固件的消息摘要和其签名是否匹配

小结

这篇文章主要是以 DIR 3040 固件为例，说明如何从未加密的老固件中去寻找负责解密的可执行文件，用于解密新版的加密固件。先说明拿到一个固件后如何判断已经被加密，然后说明如何去找到负责解密的可执行文件，再通过 qemu 仿真去执行解密程序，将固件解密，最后简单说了下固件完整性相关的知识，并重点分析了解密程序的解密 + 验证逻辑。

这次对于 DIR 3040 的漏洞分析和固件解密验证过程分析还是花费了不少的时间。首先是固件的获取，从官网下载到的固件是加密的，然后看到一篇文章简单说了下基于未加密固件版本对加密固件进行解密，也是 DIR 3040 相关的。但是我在官网上没有找到未加密的固件，全部是被加密的固件。又在信息搜集的过程中，发现了原来在 Github 上有一个比较通用的、针对 D-Link 系列的 固件解密脚本（https://github.com/0xricksanchez/dlink-decrypt）。原来，Dlink 近两年使用的加密、验证程序 imgdecrypt 基本上都是一个套路，于是我参考了解密脚本开发者在 2020 年的分析思路，结合之前看过的关于可信计算相关的一些知识点，简单叙述了固件安全性，然后重点分析了解密验证逻辑如上。

关于漏洞分析，感兴趣的师傅可以看一下我的这篇 分析文章（https://genteeldevil.github.io/2021/07/23/D-Link%20DIR%203040%E4%BB%8E%E4%BF%A1%E6%81%AF%E6%B3%84%E9%9C%B2%E5%88%B0RCE/）。

参考链接

- Breaking the D-Link DIR3060 Firmware Encryption (https://0x00sec.org/t/breaking-the-d-link-dir3060-firmware

(https://0x00sec.org/t/breaking-the-d-link-dir3060-firmware-encryption-recon-part-1/21943)

- D-Link DIR 3040 从信息泄露到 RCE (https://genteeldevil.github.io/2021/07/23/D-Link%20DIR%203040%E4%BB%8E%E4%BF%A1%E6%81%AF%E6%B3%84%E9%9C%B2%E5%88%B0RCE/)