恶意文件分析系统中的数字签名验证

安全研究部 李志昕

关键词：恶意文件分析 数字签名 Openssl

摘要：本文主要介绍了 Linux 环境下的 PE 文件数字签名验证的相关技术，着重介绍了从 PE 文件中解析数字签名，并应用 Openssl 库进行证书信任链的验证。最后，对恶意文件分析系统中对数字签名的信任策略问题，提出一些讨论和思考。

一．前言

本文是基于一个真实项目中所涉及到的数字签名验证问题的相关实践所进行的总结，并不希望详细介绍数字签名技术的方方面面，而是仅就项目中遇到的问题和解决问题的思路方法进行介绍。

本文主要介绍的内容有：在 Linux 环境下如何验证 PE 文件的数字签名，如何从 Windows 系统中导出信任证书并在 Linux 应用，以及数字签名验证通过情况下的文件信任策略。这当中所涉及到的数字签名的概念和技术，为了保证文章的完整可读，也将作以简要的介绍。

二．PE 文件数字签名格式

这里假定读者已经具备数字签名的基本理论知识，如果不了解可以先阅读一下 What is a Digital Signature[1] 这篇文章，其中用比较形象的手法解释了相关概念。

接下来，首先看一下我们遇到的问题。我们的恶意文件分析系

统（以下简称分析系统）接收到用户提交的样本文件后，在进行动态分析之前会先做一些静态分析。而在对 PE 文件的静态分析中，如果 PE 文件有数字签名，则要对签名进行验证。若数字签名验证通过，则不再对其进行后续分析。这样做主要考虑的是降低误报，以及减少服务器资源消耗。但是由于分析系统是运行在 Linux 平台上，所以无法直接利用 Windows 系统的工具来完成验证。当然，另外提供一个基于 Windows 的验证服务器也是一个很好的思路，那么在我们的项目中，采用的是在 Linux 环境下直接提取 PE 文件的数字签名并验证的方法。要做到这一点，就必须先了解 PE 文件数字签名的格式。

图 2.1 引自于微软的官方文档 [2]，该图比较清晰地展现了 PE 文件数字证书的结构，下面作以简要的描述：

(1)PE 文件头部的可选头部中，数据目录（Data Directories）的第五项，为安全目录（Security Directory），指定了证书表项（Certificate Table）在 PE 文件中的位置和大小。

(2) 证书表项，每项包含一个 8 字节的头部以及符合 PKCS#7[3] 定义的签名数据。

头部定义如下：

1 ~ 4	5 ~ 6	7 ~ 8
表项长度	证书版本	证书类型

- 表项长度：头部和签名数据的总长度
- 证书版本：常见为 0x0200 (WIN_CERT_REVISION_2)
- 证书类型：常见为 0x0002 (WIN_CERT_TYPE_PKCS_SIGNED_DATA)

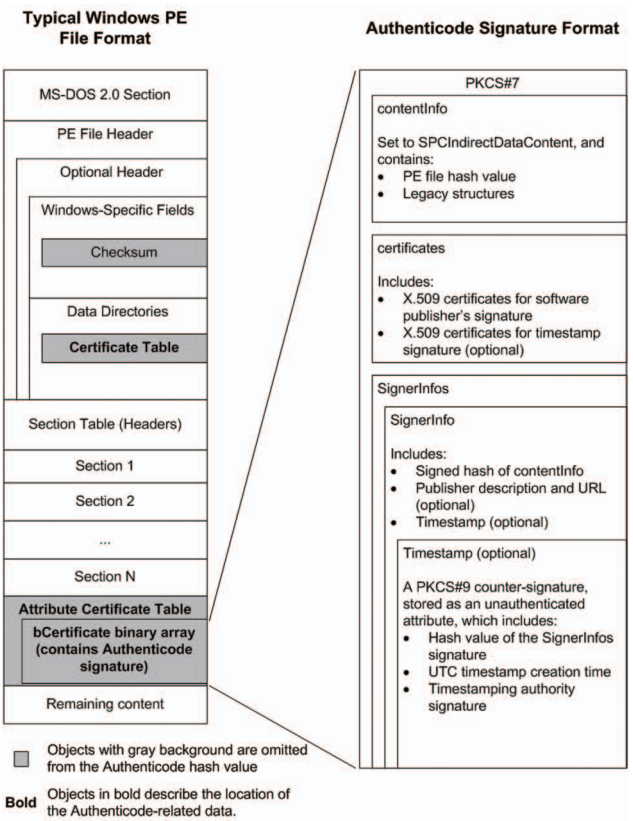

图 2.1 Windows PE 文件格式和可信代码签名格式

(3) 签名数据中主要包含了 PE 文件的 Hash 值，使用软件发布者私钥创建的签名，以及 X.509 v3 格式的证书。PE 文件 Hash 的计算不包含图 2.1 中灰色背景的部分，目的是避免绑定证书文件影响到 Hash 值。

三．应用 Openssl 验证数字签名

从 PE 文件中提取得到签名数据后，接下来就需要对其进行验证。读者可能发现，在上一节中我们并没有对签名数据进行深入的解析。主要原因：一是 PKCS#7、X.509 等等这些标准并不是本文要着重介绍的，此外我们实际可以应用 Openssl 来完成解析和验证的工作，从而绕过对这些复杂细节的纠缠。

下面简要介绍一下所使用到的主要 Openssl 库函数和验证过程：

(1)PKCS7 *d2i_PKCS7(PKCS7 **a, unsigned char **pp, unsigned int length)

该函数的作用是加载 PKCS7 SignedData 结构数据。*a 为结果写入的对象，可以传入 NULL，函数将自动创建新对象；*pp 为指向签名数据的指针；length 为签名数据长度，这两个变量的值按上一节的方法解析 PE 文件即可获得。

(2)X509_STORE *X509_STORE_new(void)

该函数用于创建 X509_STORE 对象，结合下列函数使用。

(3)int X509_STORE_load_locations(X509_STORE *store,
const char *file, const char *dir)

该函数的作用为加载可信证书，用来验证签名证书。store 即 X509_STORE_new 函数创建的对象；file 为加载的文件名；dir 为加载的目录名。

(4)int PKCS7_verify(PKCS7 *p7, STACK_OF(X509) *certs,
X509_STORE *store, BIO *indata, BIO *out,
int flags)

这个函数就是验证签名的关键函数了，下面详细介绍一下相关参数：

•p7 即为使用 d2i_PKCS7 加载的数据结构

•certs 为一套用来查找签名者证书的证书集，可传入 p7->d.sign->cert

•store 为可信证书仓库对象

•indata 为已签名的数据；这里稍微有点复杂，需要计算一下 buffer 的偏移，实际 Openssl 已经解析了结构，只要用以下代码处理一下即可：

```
/* 引自osslsigncode.c */
unsigned int asn1_simple_hdr_len(const unsigned char *p,
unsigned int len) {
    if (len <= 2 || p[0] > 0x31)
        return 0;
    return (p[1]&0x80) ? (2 + (p[1]&0x7f)) : 2;
}

int seqhdrlen = asn1_simple_hdr_len(
p7->d.sign->contents->d.other->value.sequence->data,
p7->d.sign->contents->d.other->value.sequence->length);

BIO *indata = BIO_new_mem_buf(
p7->d.sign->contents->d.other->value.sequence->data + seqhdrlen,
p7->d.sign->contents->d.other->value.sequence->length - seqhdrlen);
```

•out 为输出内容写入的 buffer

•flags 是用来修改验证操作的可选标志。主要介绍两个 flag：

1)PKCS7_NOVERIFY，表示签名者证书将不进行信任链验证。在无可信证书仓库的情况下，可以设置这个标志。

2)PKCS7_NOCHAIN，表示除了签名者证书，其他所包含的证书将不被作为 untrusted CAs 使用，也就意味着整个信任链必须包含在可信证书仓库中。在实际项目中应用了该标志，原因是某些 PE

文件中所带的其他证书可能不作为验证目的，如果不设置此标志，则会导致验证异常。我们在IE自带的某些PE文件上观察到了这种现象。按理说，不作为验证目的的其他证书不应该出现在信任链检查过程中，可能是 Openssl 的 BUG，也可能是我们哪里用法不对。理论上，设置该标志，对数字签名的验证是强验证，很多二级或多级签发的证书不能通过验证。总之，设置该标志，纯粹是为了临时解决问题。

四．创建可信证书仓库

从上一节的内容可知，如果没有可信任证书仓库，也就无法进行信任链的验证。当然，还有使用 CA 服务器等其它方式，但如果考虑在本地验证的情况下，是必须要有可信任证书仓库的。接下来我们要做的就是“乾坤大挪移”，将 Windows 系统中的可信证书导出到 Linux 系统中来，需要以下几个步骤：

(1) 导出 Windows 系统中的可信证书

“Win + R” 调出 Run 对话框，输入“certmgr.msc” 运行。在证书管理窗口（见图 4.1 ）的左侧栏中选中“Trusted Root Certification Authorities” – “Certificates”，然后在右侧栏中“Ctrl + A”全选，单击右键选择“All Tasks” – “Export…”。

在向导（见图 4.2 ）中选择“Personnal Information Exchange - PKCS#12”，完成向导导出后得到后缀为”.pfx”的文件，假定为 CAs.pfx。

(2) 转换格式

Windows 导出的证书无法直接应用，需要转换成 pem 格式，使用如下命令：

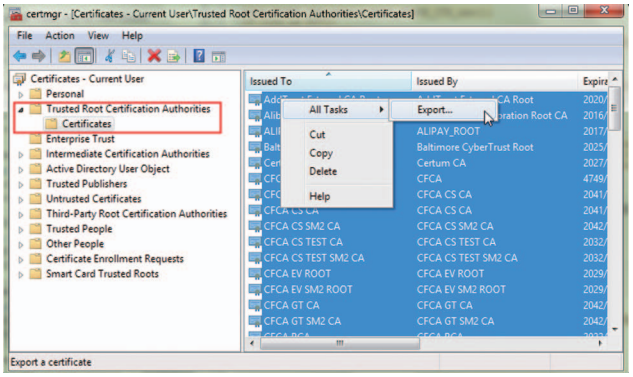

图 4.1 Windows 证书管理程序

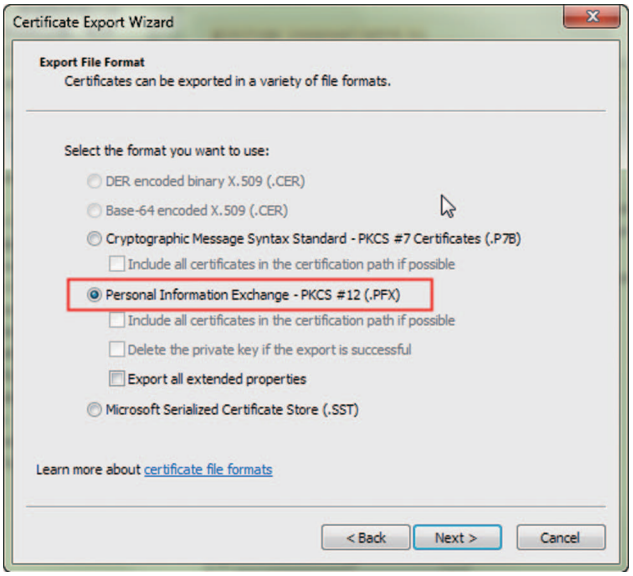

图 4.2 证书导出向导

```
$> openssl pkcs12 –in CAs.pfx –out CAs.pem
```

此时 CAs.pem 中包含了所有导出的证书，还需要将它们拆分成单独的 pem 文件。这里可以用一个简单的 Python 脚本来实现：

```
pem_bundle = file('CAs.pem', 'r')
ENDING = "-----END CERTIFICATE-----"
pem_id = 1
buf = []

for line in pem_bundle:
    buf.append(line)
    if line.strip() == ENDING:
        with file('cert%d.pem' % pem_id, 'w') as pem:
            pem.writelines(buf)
        buf = []
        pem_id += 1
```

可能有现成的 Openssl 命令行参数实现这种拆分，没有细究。

(3) 创建 hash link

最后，需要对证书文件创建 hash link，不同 Linux 发行版下用的命令可能会有所不同，例如：Ubuntu 14.04 使用 c_rehash，而 Fedora 20 使用 cacertdir_rehash。

```
$> c_rehash <directory>
```

使用上一节介绍的 X509_STORE_load_locations 函数导入该目录的可信证书就可以支持信任链的验证了。

五．签名验证通过文件的信任策略

签名验证的技术问题解决之后，是否就大功告成了呢？是不是只要签名验证通过，就可以认为这个 PE 文件一定是正常程序，换言之是一个非恶意程序？

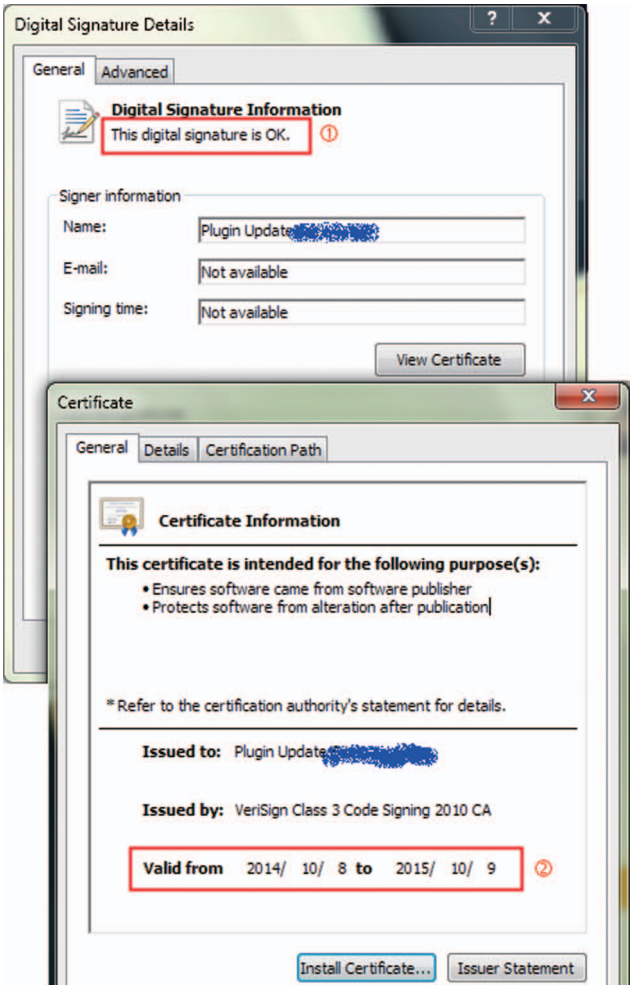

图 5.1 Windows 数字签名证书内容

图 5.2 virustotal 检测结果

起初我们的分析系统的确是这样设计的，完全信任数字签名。因为实际样本中具有数字签名的只是极少数，而签名验证通过的就更少了，所以并没有仔细思考这个问题。然而，后来我们注意到在某个较短时间内，比如某一天，会出现比平时多很多的签名验证通过的样本，感觉有一点不太正常。

为了避免是签名验证程序实现上的 bug 所致，转到 Windows 系统再来查看这些文件的证书（如图 5.1 所示），发现数字签名确实是有效的（见 1 处），只是证书的有效期通常只有一年（见 2 处）。把样本上传 virustotal 检查，结果发现确是恶意程序（见图 5.2 ）。

如此便说明了一个事实，并非能够通过数字签名验证的文件都是正常文件，很有可能是恶意软件的开发者申请合法证书并签名。所以对于数字签名，不应该直接信任，而是建立软件发布者的黑白名单。当样本的签名验证通过，但软件发布者是分析系统未知的时，则强制进行完整的样本分析过程。经过几轮相同软件发布者的样本分析后，根据分析结果将软件发布者加入分析系统的黑白名单，为后续样本数字签名的验证提供依据。

这样数字签名就成为样本分析的一个维度，通过大量样本数据的积累后，则可以以此维度进行统计分析，作为安全态势可视化的数据源。

六 . 结束语

本文中介绍的方法并不是唯一的，更不可能是最好的，只是在我们项目当中的一个技术选择。或在资源受限，或在网络受限，或考虑性能因素等情况下，可以作为一个可行方案备用。此外对于可信证书的更新、证书吊销等问题，也还需要进一步完善解决方案。

最后提出的以数字签名作为一个维度进行安全态势可视化，已经见到有安全公司做类似的在线应用，应该是值得进一步研究的。

参考文献

[1].What is a Digital Signature?

http://www.youdzone.com/signature.html

[2].Windows Authenticode Portable Executable

http://download.microsoft.com/download/9/c/5/9c5b2167-8017-4bae-9fde-d599bac8184a/Authenticode_PE.docx

[3].http://en.wikipedia.org/wiki/PKCS

[4].OpenSSL-based signcode utility

http://sourceforge.net/projects/osslsigncode/