Viewing ELF binary signatures

By skape

Overview

The concept of binary security is most definitely not a new concept. In the age of viruses and mass anoniminity amongst internet goers, the requirement to verify not only the maker of a given binary but also that the code within it has been unmodified is of quite a great importance. In open source communities most would shrug off the need for such binary signing, but, as we see such open source communities progress towards centralized precompiled packages (redhat, debian, et. all), one should begin to realize that taking the word of another when it comes to binary security is not at all wise.

Although most unix users have yet to see many elf viruses in at all the same way as they see windows viruses, the potential for such things to occur is growing ever nearer. In order to quench this problem, to prevent it before it's even allowed to mature into actuality, we must come up with a way to fend off the possibility that a binary package will be installed and be able to run binaries which may potentially contain a virus.

Approaches to such a system range from the all-but-pointless md5sum technique whereby a user can verify that a given packages md5 matches that of the md5 the server told it to expect. Another approach would be much the same, but instead of listening to what the server thinks the md5 should be, you listen to a trusted third party server which contains a database of files and md5sums.

The second solution sounds strikingly similar to another form of authentication. What could it be? In this instance, it happens to be PKI. With the advent of PKI people were empowered with the ability to identify, encrypt and verify others in a de-centralized and secure fashion. One could send a message to another person and encrypt the message in such a format that only the destined person would be able to read it. One could also verify that a message sent to them was actually sent by the sender. If one were to take the above techniques and apply them instead to the field of binary security, the advantages could potentially be quite grand. In fact, the technique is already being applied. Microsoft has, with their patented Authenticode technology, built a mechanism by which windows binaries can be digitally signed and validated against trusted certificate authorities (CAs).

The details about Microsoft's Authenticode technology and how it actually works can be found in the “References” section.

In the following sections I plan to explain how in both general and specific the concept of how ELF binary signing and validation work.

Signing

I deliberated over how to start this section for some time before deciding that the most direct route would be that of symbolism. Let me present the following analogy that I will reference throughout this section.

In modern America fast food has all-but-replaced the dinner table for a means of consumption. As such, the need for common, standard brand names in the fast food industry is a must in order to have customers associate good food with a name. A few examples include Wendy's, McDonalds, and Taco Bell. For those of you who partake in fast food, your mind is instantly able to connect the words with the type and quality of food. The way such fast food companies work is by selling franchise.

When a person wishes to run their own franchise, they apply for a CERTIFICATE. This CERTIFICATE uniquely identifies them and gives them access to Wendy's menus, propoganda, and other internal information. Without a certificate that was ISSUED by Wendy's, franchise operators or common people have no access to these things because they are not TRUSTED. The reason they are not trusted is because Wendy's does not trust them.

With that said, it's not at all uncommon to encounter a franchise of a larger fast food restraunt that makes horrible food. When the Wendy's on 8th street begins to gain a reputation for having rats in place of meat paddies, word will spread that said Wendy's should no longer be TRUSTED.

The issuer may even REVOKE that franchises CERTIFICATE, the thing that uniquely identifies them. On the other hand, though, all other Wendy's franchises are TRUSTED by default due to the fact that people, by default, trust Wendy's, the ISSUER.

So to summarize, you have an entity (Wendy's) which issues certificates to lesser entities (Franchises). These certificates are trusted because people trust the issuer (Wendy's). If a restraunt opens up and is not a Wendy's restraunt, they are not trusted by default because they do not have a certificate. Now that you're hungry, I felt it might be best to hurtle you back into reality before your attention diverges away from this paper and instead to your stomache. The above analogy was meant to give a brief explanation of a real worth example of signing and validation. Hopefully it made sense.

The Concept

The following are the goals for signing a binary:

- Be able to tell that the binary has not been modified.
- Be able to tell who it was that created the binary.

Let's start with the first problem.

Calculating Checksums

In order to show that the binary has not been modified, we need to take all of the data in the binary and represent it in some fashion. A perfect example of a way to represent it would be a one-way hash, like MD5. If we were to take the contents of the ELF binary, such as the section headers, program headers, and the section header contents and accumlate them into an MD5 hash, we will have successfully come up with a representation for our binary in its current state.

Now that we have our MD5 hash of the binary, we've successfully solved the first problem. Partly solved, at least. We still need some way to prove that the binary has not been modified. If we were simply to put the hash into the ELF binary, all it would take is for someone to replace that MD5 with their own MD5 of the modified binary, thus allowing them to circumvent what we were trying to accomplish.

It would seem that we need some form of encryption, but that form must require no password for validation. How could we possibly do that? Well, this begins to tie in with the signing aspect. We've successfully come up with a hash that proves our binary has been unmodified, but we need some way to say that 'Entity X' claims that this binary is unmodified and here is its proof. The proof being the MD5 sum.

Signing the Binary

The signing of the binary begins to tie into the Wendy's analogy. We need to establish a trust heirarchy in order to make things work easily and securely. An example of a trust heirarchy as it relates to certificates is VeriSign. VeriSign is what's known as a Certificate Authority (CA). They're like Wendy's the corporation from the analogy above. They issue certificates to other entities and, in so doing, cause those entities to become 'trusted' due to the fact that VeriSign is trusted.

For those of you who aren't familiar with certificates, certificates hold information about a given entity, like their company name, group name, email address, expiration date, and a few other things. Let's say for the sake of simplicity that my company is Bob, Inc. I've been given a certificate that has been issued by VeriSign and now I want to use my certificate to sign my binary applications. I've already calculated my checksums, all that's left for me to do is to do the actual signing.

Viewing ELF binary signatures

By skape

Now this part that I'm about to describe is quite technical so in order to keep this document in scope, I've voted to give a general overview of the signing process. The basic jist of signing is this:

1. Calculate the MD5 checksum for a given binary.

2. Take your certificate and private key that VeriSign (or any other CA) issued you and 'sign' the checksum.

3. Take the signed data and write it to the binary in its own elf section.

I realize that the second step describes quite a complex operation, but I like to consider that operation a sort of black box. OpenSSL provides an interface to such a black box via the X509_sign function. This function allows one to sign arbitrary data with a given certificate and private key combination. The data signed can be written out in PEM format and then written to the elf section created for the signature in the binary.

A signed binary might look something like this:

```
# readelf -S /root/ownme | grep sig [27] .sig PROGBITS 00000000 001004 00077f
# readelf -x 27 /root/ownme

Hex dump of section '.sig':
0x00000000 3753434b 50204e49 4745422d 2d2d2d2d -----BEGIN PKCS7
0x00000010 6f4b4a59 515a4649 494d0a2d 2d2d2d2d ------.MIIFZQYJKo
0x00000020 6a564649 496f4363 51414e63 7668495a ZIhvcNAQcCoIIFVj
0x00000030 67424a41 7a437845 51414349 56424343 CCBVICAQExCzAJBg
0x00000040 53434773 444d4155 6747434d 67447255 UrDgMCGgUAMDsGCS
0x00000050 42754161 41484551 440a3362 49534771 qGSIb3.DQEHAaAuB
0x00000060 63355256 4c303557 5a303532 62447843 CxDb250ZW50LVR5c
0x00000070 61687847 63765148 656c5248 49365547 GU6IHRleHQvcGxha
0x00000080 764c7868 3769486c 554b3067 434e3457 W4NCg0KUlHi7hxLv
0x00000090 4343716d 456b7769 0a61474f 7033564d MV3pOGa.iwkEmqCC
0x000000a0 4441616d 4349494d 774d6767 77517a41 AzQwggMwMIICmaAD
...
```

Verifying

Verification of a signed binary is a less daunting task than the signing part of it. The procedure entails the following:

1. Calculate the MD5 checksum of the binary, skipping over the signature section of the elf binary.

2. Extract the signature stored in the signature section and determine if the binary was signed and issued by a trusted authority.

3. Decrypt the signed checksum in the binary and compare it with your calculated checksum. If they match the binary has not been modified.

Once again I've left a kind of black box for steps 2 and 3. In OpenSSL there is a function called X509_verify which will verify that a given signature (extracted from the elf section) was signed by a given certificate authority as well as decrypt the signed data and give you the buffer to work with in plaintext. So, simply enough with one function call you're able to take care of both steps 2 and 3.

Using elfsign and elfverify

Using elfsign to sign a binary image is quite easy. Let's say that I wish to sign the binary /root/ownme with my certificate. My certificate is for my company, 'Test Company', and my certificate was issued by 'Uninformed Research'. In order to sign /root/ownme, I would do the following:

```
# ./elfsign -f /root/ownme -c tester.cert -p tester.privkey
```

```
Key Password: <My certificates private key passphrase>
# readelf -S /root/ownme | grep sig [27] .sig PROGBITS 00000000 001004 00077f
```

Well, it looks like we've successfully signed /root/ownme, but let's make sure.

```
# ./elfverify -f /root/ownme -c ca.crt
OK
#
```

Sweet, it appears everything is kosher. elfverify has validated the fact that /root/ownme is signed and the certificate that signed the binary was issued the ca.crt.

What happens if I don't tell elfverify what root CA was used?

```
# ./elfverify -f /root/ownme
Issuer: C=US, ST=Kansas, L=Overland Park, O=Uninformed Research, OU=R&D
Signer: C=US, ST=NY, O=Test Company, OU=Test Group, CN=Tester
Issuer is not trusted, would you like to trust them? [y/N] Y
OK
```

It presents me with a prompt letting me know that the certificate was not trusted by default and asks me if I would like to trust it which, in this instance, I said yes to. What happens if we modify /root/ownme?

```
# diff /root/bob /root/ownme
Binary files /root/bob and /root/ownme differ
# ./elfverify -f /root/bob -c ca.crt
FAIL (The binary digest did not match the signed digest.)
```

What happens if we try to verify a binary that hasn't even been signed?

```
# gcc /root/a.c -o /root/a -ldl -lpthread
# ./elfverify -f /root/a -c ca.crt
FAIL (The binary is not signed.)
```

Conclusion

Although this document doesn't go indepth into the concepts of PKI, certificate authorities and the actual process of signing and verifying data, it was my hope that one would be able to walk away with an understanding of how binary signatures are possible and why it is that they are useful in both opened and closed source communities.

In the future I would hope to see an introduction of a root certificate authority for open source projects, or, at least for operating system distributions that distribute binary applications. I would also like to see an adoption of an automated technique for authenticating linux binaries at execution time via a kernel module. I think that this support (optional of course), would lend great amounts of security to local unix installations and will help to protect against malicious users overwriting system binaries.

References

Authenticode Concept http://www.tutorialbox.com/tutors/J++/ch23.htm