Securing on-the-instrument "plug and work" device drivers

J. Zedlitz, A. Kurland, N. Luttenberger
Communication System Research Group
Christian-Albrechts-University
D-24098 Kiel, Germany
{jze|afk|nl}@informatik.uni-kiel.de

Abstract *– In the MBARI "plug and work" approach, the use of different kinds of measurement instruments is facilitated by storing machine-readable device identification within the instrument. This identification can be used to retrieve an appropriate device driver – for example via a network connection. We present a solution for offline operation by automatically loading device driver code from the instrument to the host computer using the PUCK protocol. In this setting, it can obviously not be excluded that some person equips an instrument with malicious code and connects the spoiled instrument to the measurement host. Since retrieval and execution of the driver code is performed without human interaction, the host would immediately load the malicious code and start executing it—with unforeseen damages to the measurement process. In our contribution, we propose as countermeasure to protect the byte code of the device driver by a digital signature. A valid digital signature guarantees both origin and integrity of the device driver code. After loading the code from the instrument's storage, the host checks the validity of the signature and verifies that the origin of the signature (e.g. the device driver's author) is known and trusted. In a case study, we used the Java Distributed Data Acquisition and Control framework (JDDAC) that comes with minor overhead. However, an adaptation to other (especially Java based) frameworks should be very easy.*

Keywords *– PUCK, plug-and-work, device driver, security*

PROBLEM STATEMENT

Configuration of oceanographic observatories—especially when performed at sea or under water—is a difficult task. Therefore the number of steps needed to integrate a new instrument into an observatory should be reduced as much as possible. Every removed or automatically performed step reduces the possibility of errors. Even a minimal standard that contains a unique identifier for every compliant device enables automated integration of instruments if the device driver is present at the observatory or can be retrieved via network connection. However, up-to-date drivers for the newly connected instrument are not always present at the observatory. Also network connection to retrieve the device driver is often not available. We present a way to store the device driver within the instrument. That way the newest driver for the newly connected instrument is always available—event without any network connection.

One somewhat minimal protocol for "self-identifying" instruments that use serial interfaces is PUCK from the Monterey Aquarium Research Institute (MBARI) [1]. Serial interfaces are common for most today's oceanographic instruments [2]. PUCK accomplishes this by storing a 96-byte "instrument data sheet" containing machine-readable device information within the instrument. The data sheet contains information about manufacturer, model, version, the name of the instrument as well as a universally unique identification number (UUID). When the instrument is plugged in, the observation system can retrieve the data sheet from the instrument through the standard PUCK protocol. Instead of using the UUID contained in the data sheet to retrieve an appropriate device driver for the plugged-in instrument we make use of PUCK's additional "payload" storage. This storage that is able to hold an amount of arbitrary data that can also be retrieved using the standard PUCK protocol. Into this data storage we place the device driver for the instrument.

The general procedure is shown in fig. 1: The host system reads the byte code of the device driver using the PUCK protocol. Using this byte code an instance of the device driver will be created. Afterwards the device driver is able to answer requests from the host system by talking to the instrument in its device-specific protocol. Or it can forward data created by the instrument to the host system.

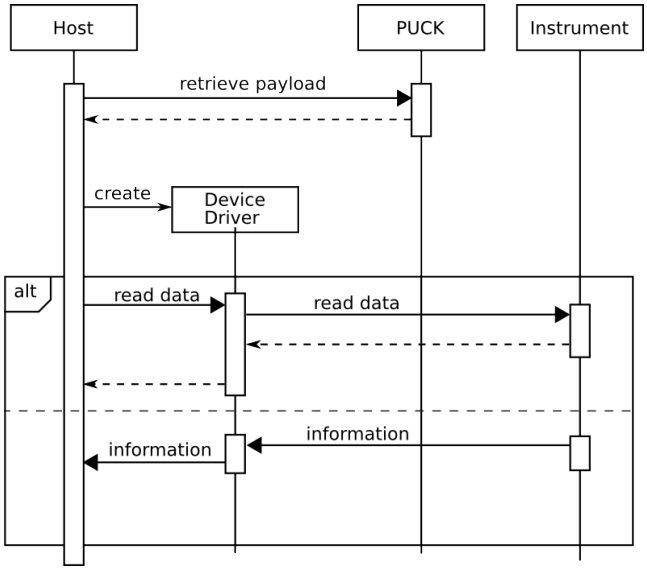

Fig. 1. Sequence diagram showing the general procedure.

INSTANTIATING THE DEVICE DRIVER

The device driver stored within the PUCK data storage consists of one or more Java classes that are not known to the Java runtime environment, yet. The classes are also not inside an area accessible to the Java runtime environment (i.e. the classpath). Therefore it is necessary to implement and use our own class loader. It reads in the byte code of the class using the standard PUCK protocol and creates a class object that is passed to the Java runtime environment.

To be able to invoke methods of device driver the device driver class has to implement a predefined interface (see fig. 2: interface `DeviceDriver`) that defines the methods the observation system needs to operate the instrument. However, it is no problem if the device driver class implements additional interfaces (see fig. 2: interface `OtherInterface`). That way the same device driver class can be used for different observation systems.

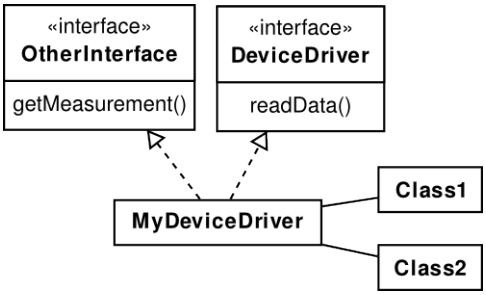

Fig. 2. Device drivers can implement multiple interfaces and consist of several classes.

More complex devices drivers typically consist of more than one Java class (e.g. multiple threads, action listener, anonymous classes). Therefore instead of putting a single class-file into the PUCK "payload" all Java classes belonging to the device driver are packed into a Java Archive (jar). The file format is explained in detail in [3]. Basically it is an archive format based on the *ZIP file* format and JAR files can indeed be built using common zip tools. The JAR file format allows optional meta information stored along with the class files. The JDK provides a special "jar" command which simplifies editing this meta information.

The operator of the instrument transfers the JAR file with the device driver prepared that way into the device's PUCK data storage before deployment. The PUCK specification does not define the format of the payload. However, a common way how to store multiple data items within the payload area has been developed: the files are stored in the payload's byte array sequentially. They can be identified by a name. A size information for each item indicates how many bytes belong to this entry. Transfer errors can be detected by a MD5-checksum.

As soon as a PUCK-enabled instrument is connected to the observation system, the host computer reads the PUCK datasheet containing some mandatory meta-data about the instrument. Using this data some decisions can already made at this point if and how the instrument can be put into operation automatically. It is possible that beside the device driver needed for this particular observation system the PUCK payload area contains additional files. Therefore, the host parses the payload-area until an item with a specified name (in our use case "JDDAC") appears or the end of the payload-area is reached—in the latter case the device cannot be put into operation automatically. If the payload-area contains an item with a matching name the further processing depends on the capability to create files on the host system:

Case A—No files can be created on the host system: If no caching is possible the complete JAR file has to be processed while data is read from the PUCK's payload-area. The Java platform provides a class `JarInputStream` to process a JAR file as sequential byte stream conveniently. It returns the entries of the JAR files one after the other while reading data. Our `JarStreamClassLoader` (fig. 3) iterates over all entries of the JAR file. As written above a JAR file might contain other files beside the binary code of Java classes. Therefore the information about filenames and paths contained in the JAR file is used to check if an entry is a class-file with the suffix ".class". In this case the byte code contained in the file is used to create a new Java class which is passed to the Java runtime environment. A disadvantage of this case is the fact that all classes inside the JAR file will be loaded even if it will never be used. Therefore care must be take while creating the JAR file to include only necessary classes.

Case B—The host system allows files to be created: If it is possible to create files, the relevant content of the PUCK payload-area is transferred into a file on the host system. This file with the cached data can used for random access later at any time. It is passed as a parameter to our `JarFileClassLoader` (fig. 3), an extension of a standard Java class loader. Each time a class is used, that is not yet present in the system, the Java runtime environment will invoke the method `findClass` of our class loader. Within that method—using information about filenames and paths contained in the JAR file—the corresponding entry within the JAR file is located. With its content—the byte code of the requested class—a new Java class object is created and passed to the Java runtime environment. An advantage of this case is the fact, that the JAR file might contain other unused classes that won't be loaded into the host's memory because the Java runtime environment will never request them.

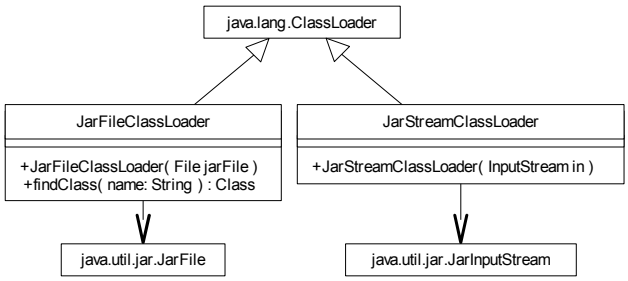

Fig. 3. Our two class loader classes.

SECURITY

With the observation system automatically retrieving and executing device driver code, it can obviously not be excluded that some person equips an instrument with malicious code and connects the spoiled instrument to the measurement host. Since retrieval and execution of the driver code is performed without human interaction, the host would immediately load the malicious code and start executing it—with unforeseen damages to the measurement process or other actions performed based on these measurement. Consider the chaos an infiltrated tsunami warning system could cause when untruly reporting a large scale flood wave.

Therefore two things have to be validated before the device driver is executed: *origin* and *integrity* of its byte code. The *origin* of the byte code gives information about the person or institution responsible for the device driver. Often it identifies the person who wrote the source code, compiled it into byte code and/or created the final software bundle. Validated *integrity* ensures that nobody modified the byte code on the way from its origin to the host com-

puter where it will be executed. Even a small manipulation of only one byte that e.g. could invert the logic of a comparison. That change—made on purpose or as a result of a transmission error—would break the integrity of the software.

A simple approach is to XOR the device driver's byte code with a secret key. This calculation can be performed very quickly and without further memory consumption while loading the binary data from the PUCK payload. Only if the byte code has been encrypted with the same secret key as it is stored on the observation system the result will be a valid Java class. However, Java classes start with a number of known and predicable bytes (file identifier CA FE BA BE, version information and class name). That means that the secret key has to be quite long. The well-known structure of Java class files makes this simple approach an easy target for cryptographic attacks, too.

DIGITAL SIGNATURES AND CERTIFICATES

Digital signatures based on asymmetric cryptography are a common way to guarantee both origin and integrity of computer programs. They are for example well known in update managers for operation systems and in Java applets loaded from the internet. We use the same technique to sign a file that contains the device driver code. After loading the file from the instrument's storage as described above the host checks the validity of the signature and verifies that the origin of the signature is known and trusted.

Digital signatures make use of public and private keys. A signer uses it's private key to encrypt a checksum of the file he wants to sign. This encrypted checksum is called the signature. To validate the signature a receiver decrypts the signature with the sender's public key. He also calculates the checksum of the transmitted file. If checksum and decrypted signature are equal the file has not been modified—its integrity has been verified.

If the public key is sent along with the signed files—which is typical for signed code when you don't known in advance the exact set of users who are trusted and their public keys—some effort has to be taken to verify the origin of the signature. Certificates are a common way to achieve that. X.509 is the most important standard for digital certificates [4]. An X.509 certificate contains at least a public key and a subject which describes the owner of the key. It also has an issuer, the authority which signed the certificate. Both subject and issuer are implemented as distinguished names. The distinguished name identifies an entity, e.g. a person or organization. Every certificate must be signed by an issuer to attest the identity of the subject and the integrity of the entire certificate. Signing a certificate means to create a signature with the issuer's private key that contains information about the certificate content. There are two possibilities:

1. A certificate can be be self-signed. In that case owner and issuer of the certificate are the same person.
2. It can be signed using a different private key.

In that second case, the integrity of the certificate can be verified by using the issuer's public key contained in another certificate. The connection between these different certificates is called *certificate chain*. Figure 4 shows such a three-certificate chain and its chain of trust.

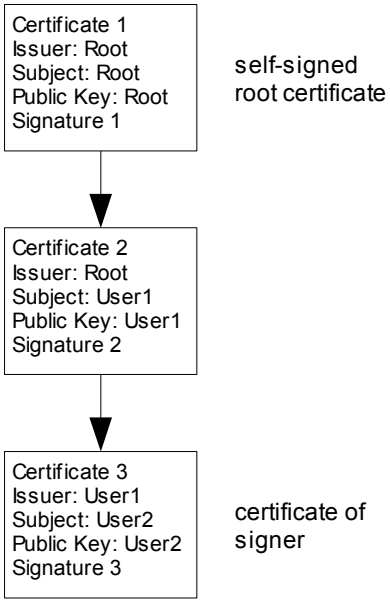

Fig. 4. Example of a certificate chain

There are no possibilities to verify the identity of a self-signed certificate. Therefore it has to be provided by a trustworthy authority and stored at a trustworthy location.

Validating and verifying a certificate means to verify every certificate of its certification chain. So, if the host checks the validity of a given certificate of the device drivers signer (e.g. Certificate 3 from fig. 4), he also has to check every certificate in its chain of trust. Therefore the certificate's signature has to be verified using the public key of its issuer—in this example the subject of the intermediate certificate (Certificate 2). Before this certificate can be used for verifying, it is again important to verify its signature using the root certificate and its public key. Because the root certificate is self-signed only its own public key can be used to verify the root certificate. If and only if every certificate of the certificate chain can be verified and the root certificate is trustworthy the last certificate (Certificate 3) can also be called trustworthy.

SIGNED JAR FILES

As described above the device driver binaries are packaged in a JAR file. This JAR file is signed with a digital signatures to ensure authenticity and integrity. It is possible that a JAR file contains signatures from more than one signer. In that case it is up to the observation system if one valid and trusted signature is sufficient or if all signatures have to be valid. A signed JAR file contains the certificate chain of every signer and a special file called *manifest*. The manifest file contains security and configuration information which will be described below. It also lists checksums for every signed file inside the archive. The algorithm used to calculate the checksum is strong in the sense that changing the content of the file without changing the original checksum is computationally difficult.

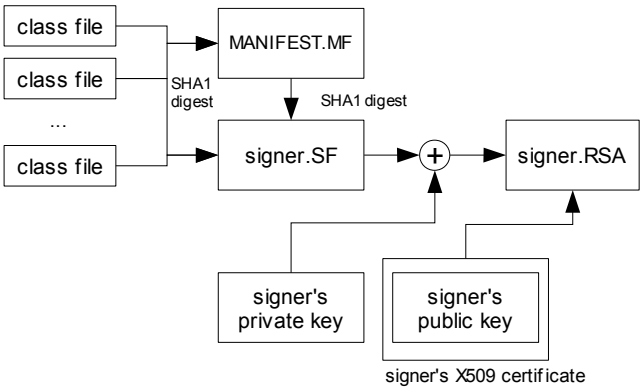

Fig. 5. Structure of a signed JAR file

For every signer the signed JAR file contains a *signature file* (file extension .SF). Additional to the checksums in the manifest this file contains checksums for each file of the archive as well as a checksum for the manifest file itself. To assure the origin of the files the signature files is digitally signed with the signer's private key. Together with the signer's certificate chain this signature is stored in a binary *signature block file* (file extension depends on the signature algorithm used—e.g. RSA in fig. 5).

To verify a JAR file basically the following three steps have to be performed:

1. Check the certificate chain of the signer's certificate found in the signature block file in the way described above.
2. Use the public key of the signer to check the signature of the signature file containing the list of checksums.
3. Recalculate the checksum for every file in the archive and compare it to the listed value.

VERIFYING A DEVICE DRIVER

Depending on the capability of the host system the driver will be downloaded from PUCK memory as a byte stream or into a file (see above). It is then passed to the verifier. The verifier checks the JAR file basically in the manner as described above. For checking the integrity of the JAR file it uses methods provided by the Java runtime environment. These methods operate on byte streams. Therefore no special care has to be taken whether it is possible to store files on the host system or not.

The existing methods from the Java runtime environment for checking JAR files do not check files that are not listed in the manifest. This would enable an attacker to include a new Java class file. If this class contains malicious code in a static block that malicious code would be executed as soon as the class is loaded. Therefore our verifier also assures that the archive does not contain a file not noted in the manifest.

If the verifier detects an manipulated or unexpected file inside the archive it will stop loading the JAR file. Since the integrity and origin of unsigned JAR files cannot be checked they will be rejected as well.

The owner of the observation system has the power to decide which device drivers will be accepted by selecting the trusted root certificate(s) stored inside the observation system. In combination with a policy which certificates will be signed with this root certificate a precise set of people, allowed to create valid device drivers, can be defined. Signed device drivers with a certificate chain starting with a different root certificate will be rejected by the verifier because the trust chain cannot be traced to a trusted root certificate stored inside the observation system.

SUMMARY AND FURTHER WORK

We have presented an approach for Java based observation systems how to store complex device drivers within an instrument, retrieve them using the PUCK protocol, and executing them on the host system. To prevent the execution of malicious code we propose to use digital signatures. The JDK already contains good support for creating and checking these signatures.

It should also be possible to use this approach with the PUCK payload containing machine code rather than Java classes. This code might be used as a dynamic library or executed by setting the processor's instruction pointer to the begin of the loaded data. Unlike in a Java environment no out-of-the-box solutions exist for check integrity and origin of the code. However, similar check can be implemented in other languages, too.

To disable certain certificates (e.g. from employees that left the institute) newly connected instruments could be used to transfer certification revocation lists to the observatory. Care must be taken to only accept revocation lists from trusted sources.

REFERENCES

[1] T. O'Reilly, K. Headley *et al.*, "MBARI technology for self-configuring interoperable ocean observatories", *OCEANS 2006*, pp. 1–6, Sept. 2006

[2] T. O'Reilly, K. Headley *et al.*, "Instrument Interface Standards for Interoperable Ocean Sensor Networks ", *OCEANS 2009*, pp. 1-9, May 2009

[3] Sun Microsystems Inc. "JAR File Specification", 2003

[4] D. Cooper *et al.*, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", *IETF RFC* 5280, May 2008