aicas GmbH

JamaicaAMS User Documentation

Version 1.3.1

21 January 2026

aicas GmbH
Emmy-Noether-Straße 9
76131 Karlsruhe
Germany

Email: `support@aicas.com`

Contents

Chapter 1

JamaicaAMS Framework

JamaicaAMS (Application Management System, AMS) is a modular and extensible application framework, especially designed and tailored for Industrial IoT use cases. It provides a powerful runtime environment for Java-based applications and components, thereby supporting not only static but also highly dynamic and distributed application scenarios.

JamaicaAMS targets in particular at heterogeneous embedded and mobile devices with sparse resources, providing performance guarantees for their applications during runtime. However, high-end servers and computational ecosystems are feasible target platforms as well, which also can take advantage of the integrated framework features and technology in a larger scaled fashion.

The strength of JamaicaAMS results from the combination of two solid open standards: OSGi™[1], that specifies a software architecture to create modular applications and services, called bundles, and the Real-Time Specification for Java (RTSJ) [8].

JamaicaAMS extends Apache Felix (currently version 7.0.5), which is an open source implementation of the OSGi specification, in its Core Release 8. The Apache framework implements standardized mechanisms for lifecycle and dependency management of components, service abstractions as well as standardized security and isolation mechanisms.

By implementing the OSGi open technology, JamaicaAMS allows for a seamless and automated integration, configuration and update of application components on a big amount of remote devices, at once and during runtime. By eliminating system downtime for maintenance and updates, it vastly improves the availability of services and avoids disruption, for instance in production, automation and consumer processes.

Moreover, JamaicaAMS relies on the robustness of JamaicaVM, aicas' hard realtime Java-based Virtual Machine. Based on the RTSJ specification, JamaicaVM implements mechanisms for realtime programming and execution. Therefore JamaicaAMS permits to explicitly configure and enforce resource budgets for applications and their components during runtime. Thus, even if the system might be overloaded and misbehaving, the framework and the resource-isolated applications are ensured to operate in a reliable fashion. This is especially useful for mixed critical- and control automation systems where, for example, the sampling of sensors and the control of actuators must be guaranteed in either case.

[1] OSGi™is a trademark, registered trademark or service mark of the OSGi Alliance.

Changes implemented in the Scheduler of JamaicaVM require setting certain capabilities, to allow JamaicaVM to access realtime priorities and provide realtime thread guarantees.

On Linux, for example, one must either run (effectively) as root or set the `CAP_SYS_NICE` capability on the executable, e.g., "`setcap cap_sys_nice=eip`" executable.

Please note that using "`setcap`" also requires root privileges. However, on a dedicated RTOS (Real-Time Operating Systems), such as QNX, this is generally not needed.

JamaicaAMS can be complemented, when used together with a set of specialized tools and features (see Figure 1.1) which support development, analysis, build, deployment, configuration and maintenance of Java-based applications and components. Integration with well established development environments, like the Eclipse IDE, speeds up the implementation process, shortening the time-to-market of future IoT applications.

Cloud connectivity features can easily be integrated into JamaicaAMS to allow for a central platform and device management by remote operators.

Industrial IoT use cases are especially supported by automation bus interfaces, like Modbus. Finally, a custom component repository can be easily deployed and integrated, providing a common codebase for a multitude of devices in an app-store fashion. Comprising those features and functionalities based on open standards, JamaicaAMS provides a powerful and portable platform for a variety of innovative, performance critical, highly distributed and dynamic industrial and IoT scenarios.

This document provides information about particular technologies and components included in the JamaicaAMS. It assumes from its readers a certain knowledge about the OSGi standard. However, for those who are not familiar with the specification, Section 5 offers a brief introduction to the OSGi layers and the core concept of bundle lifecycle, which are central to the development and management of modular applications.

For additional information or conceptual clarification, please see the OSGi Core Release 8 Specification [6].

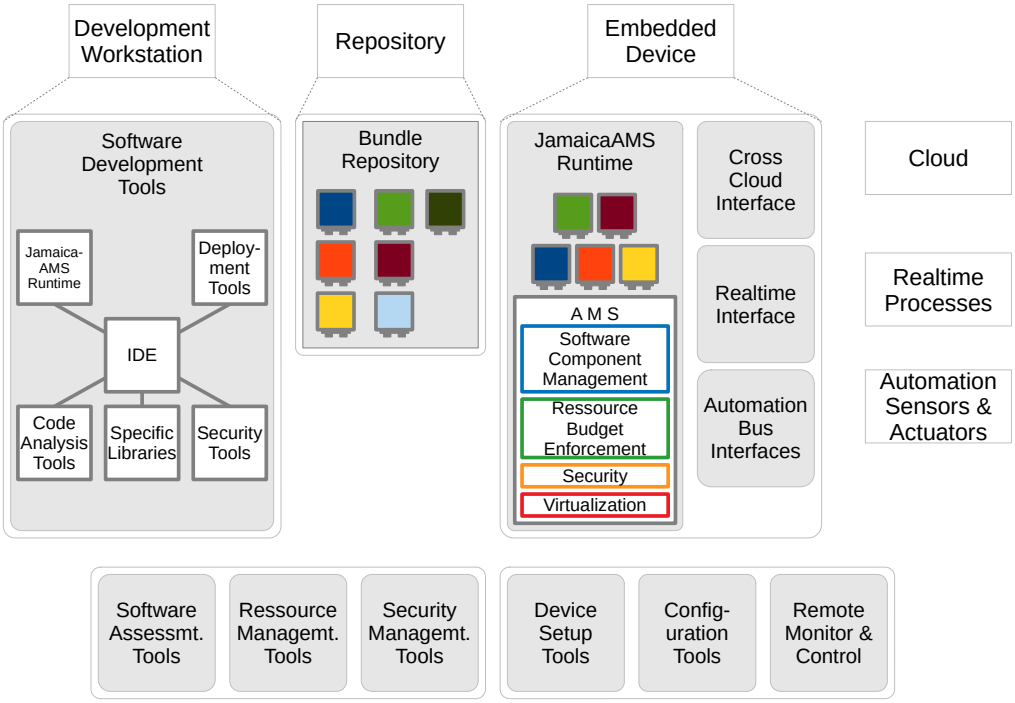

Figure 1.1: JamaicaAMS Framework and its components

Typographic Conventions

Throughout this manual the following typographic conventions were applied.

For items related to the file system, like paths and files:

`path`

`filename`

For references in code, like classes and methods:

`class`

`method`

Commands are given like:

“`command`”

Names of components, modules or bundles appear like:

“component”

“module”

“bundle”

Properties, Parameters, Environment Variables, and other items that require definition are written like:

property

VARIABLE

Output in terminal sessions is reproduced in `monospaced`. User inputs are designated by a prompt:

```
> input
```

Please note that the placeholder "< >" is to be replaced by information pertinent to the particular deployment in question, e.g., `<path to JamaicaAMS>` should be replaced with the current path to the JamaicaAMS directory in the user's specific filesystem.

Chapter 2

Getting Started

2.1 System Requirements

Figure 2.1 shows the combination of operating systems and processor architectures currently supported by JamaicaAMS.

Operating Systems	Processor Architectures
Ubuntu 24.04 LTS	x86_64
Linux	RISC-V
Pi OS	ARMv7-A
Pi OS	ARMv8-A
QNX 7.1	x86_64
QNX 7.1	ARMv8-A
QNX 8.0	x86_64
QNX 8.0	ARMv8-A
VxWorks 24.03	x86_64

Figure 2.1: JamaicaAMS supported platforms.

JamaicaAMS runs on top of JamaicaVM, therefore it supports all platforms that JamaicaVM supports. If you are interested in a specific OS/processor architecture combination not mentioned here, please contact aicas.

Without loss of generality, the following example describes one of the typical configurations for running JamaicaAMS on an embedded target device (e.g., Raspberry Pi 4 Model B in this example).

This chapter offers information for a straight start, on how to install and execute JamaicaAMS on the target platform.

- **Operating System:** Pi OS (Kernel version: 5.15)
- **Processor Architectures:** ARMv8-A (ARM-Cortex-A72, 1.5 GHz CPU clock, 4GB SDRAM)

Note that...

Running on a target with much less computing power is possible or even more common in a typical JamaicaAMS use case. For more detailed information about the supported targets, please contact support@aicas.com.

2.2 Installation

JamaicaAMS is distributed in the form of a ZIP file. In order to install JamaicaAMS simply unzip the file and copy the created directory structure to the target platform. In case of the example target:

```
> unzip JamaicaAMS-<version>-linux-aarch64-raspberrypi-full.zip
```

the file structure of the JamaicaAMS distribution is depicted below.

```
jamaica-ams
|-- bundles.optional
|   |-- jline-<version>.jar
|   `-- org.apache.felix.gogo.jline-<version>.jar
|-- doc
|   |-- build.info
|   |-- documentation.pdf
|   |-- KNOWN_ISSUES
|   `-- Release_Notes
|-- example
|   |-- infinite-loop-example-<version>.jar
|   |-- infinite-loop-example-<version>-sources.zip
|   |-- io-blocked-thread-example-<version>.jar
|   |-- io-blocked-thread-example-<version>-sources.zip
|   |-- memory-consumption-budget-example-<version>.jar
|   |-- memory-consumption-budget-example-<version>-sources.zip
|   |-- memory-consumption-example-<version>.jar
|   |-- memory-consumption-example-<version>-sources.zip
|   |-- primes-budget-example-<version>.jar
|   |-- primes-budget-example-<version>-sources.zip
|   |-- primes-example-<version>.jar
|   |-- primes-example-<version>-sources.zip
|   |-- primes-lower-priority-example-<version>.jar
|   |-- primes-lower-priority-example-<version>-sources.zip
|   |-- thread-spawning-budget-example-<version>.jar
|   |-- thread-spawning-budget-example-<version>-sources.zip
|   |-- thread-spawning-example-<version>.jar
|   `-- thread-spawning-example-<version>-sources.zip
|-- license
|   |-- AICAS_EVALUATION_LICENSE
```

```
|   |-- ApacheLicense2
|   `-- NOTICE
`-- setup
    |-- bin
    |   |-- jams
    |   |-- jamsi
    |   |-- jamsp
    |   `-- jamst
    |-- bundle.1
    |   |-- org.apache.felix.log-<version>.jar
    |   |-- osgi-log-writer-<version>.jar
    |   `-- security-<version>.jar
    |-- bundle.2
    |   |-- configuration-admin-<version>.jar
    |   |-- jakarta.json-<version>.jar
    |   |-- org.apache.felix.cm.json-<version>.jar
    |   |-- org.apache.felix.configurator-<version>.jar
    |   |-- org.apache.felix.scr-<version>.jar
    |   |-- org.osgi.service.component-<version>.jar
    |   |-- org.osgi.util.converter-<version>.jar
    |   |-- org.osgi.util.function-<version>.jar
    |   |-- org.osgi.util.promise-<version>.jar
    |   `-- policy-file-reader-<version>.jar
    |-- bundle.3
    |   |-- org.apache.felix.gogo.command-<version>.jar
    |   |-- org.apache.felix.gogo.runtime-<version>.jar
    |   `-- org.apache.felix.gogo.shell-<version>.jar
    |-- conf
    |   |-- all.policy
    |   |-- config.properties
    |   |-- logging.properties
    |   |-- osgi.all.policy
    |   `-- system.properties
    `-- lib
        |-- aarch64
        |   `-- lib<identity>.so
        `-- fonts
            `-- Vera<identity>.ttf

13 directories, 63 files
```

The JamaicaAMS directory structure is explained below. Please note that the placeholder between "< >" needs to be replaced by information pertinent to the particular deployment in question, e.g., in Figure 2.2, the current path to the JamaicaAMS directory.

[1] Here, the OSGi bundles from `bundle.`i are assigned start level i.

Directory	Contents
<path to JamaicaAMS>/doc	documentation
<path to JamaicaAMS>/example	examples
<path to JamaicaAMS>/bundles.optional	optional bundles not in default setup
<path to JamaicaAMS>/setup/bundle.i [1]	auto-deploy OSGi bundles
<path to JamaicaAMS>/setup/conf	configuration files
<path to JamaicaAMS>/setup/bin	executable binaries
<path to JamaicaAMS>/setup/lib	Bitstream Vera fonts and dynamically linked shared libraries

Figure 2.2: JamaicaAMS directory structure.

2.3 Execution

The JamaicaAMS distribution contains four executable binaries in the `bin` directory: `jamsi` (interpreted), `jams` (default), `jamst` (tool interface) and `jamsp` (profiling).

The `jamsi` executable should be used when debugging problems or issues, because it provides more detailed output in error cases; the `jams` executable contains ahead-of-time compiled classes which may lead to better performance. The `jamst` executable provides remote debugging functionality and the `jamsp` executable should be used to collect information on the amount of runtime spent for the execution of individual methods.

2.3.1 Launching

The binaries can be launched as follows, where `jams` is used for concreteness and `jamsi` can be launched the same way:

```
cd <path-to-root-directory-of-JamaicaAMS>/setup
./bin/jams [-c <config-properties-file>] \
           [-s <system-properties-file>] \
           [-b <bundle-cache-dir>] \
           [-r <bundle-readonly-cache-dir>]
```

Here, the parameters have the following meaning:

-c <config-properties-file>

points to a framework configuration file. That file contains a description of the most important framework configuration properties and can be adapted if needed. If not given, it defaults to `./conf/config.properties`.

-s <system-properties-file>

points to a system configuration file. This makes it possible to set system properties when running JamaicaAMS. The JamaicaAMS distribution contains a sample system configuration file that shows how to enable security. If not given, it defaults to `./conf/system.properties`.

-b <bundle-cache-dir>

specifies the cache directory for the OSGi bundles. If not given, it defaults to `./jamaica-ams-cache`.

-r <bundle-readonly-cache-dir>

specifies the read-only cache directory. If the specified directory does not exist, the framework will fail to start. There is no default value, if it is not explicitly set, the read-only cache is disabled.

Additional parameters

Apart from those, the following parameters are also available:

-version/--version prints the version of JamaicaAMS and exits.

-help/--help prints usage information of JamaicaAMS and exits.

2.3.2 Environment Variables

The following environment variables are recognized by JamaicaAMS:

- **JAMAICA_AMS_HEAP_SIZE** specifies the Java heap size to be used by JamaicaAMS. The Java heap is allocated at startup and stays at the fixed size specified by this environment variable. Default: 64M
- **JAMAICA_AMS_JAVA_STACK_SIZE** specifies the Java stack size to be used by Java threads. Values that are too small can cause a Java `StackOverflowError` to be thrown. Default: 18K
- **JAMAICA_AMS_NATIVE_STACK_SIZE** specifies the C stack size to be used by Java threads. Values that are too small can cause a Java `StackOverflowError` to be thrown or the system to crash. Default: 256K
- **JAMAICA_AMS_NUM_THREADS** specifies the number of Java threads that JamaicaAMS should create on startup. No further Java threads will be created once this number of active threads has been reached. Default: 80

  Please note that JamaicaAMS inherits from JamaicaVM the limitation of 511 as maximum number of Java threads.
- **JAMAICA_AMS_JARACCELERATOR_LOAD** enables or disables loading the compiled code of an accelerated bundle. It can be set to "`true`" or "`false`". More information about acceleration can be found in Section 3.3.1. Default: "`true`".
- **JAMAICA_AMS_JARACCELERATOR_VERBOSE** enables or disables displaying the steps performed for loading the compiled code of an accelerated bundle. It can be set to "`true`" or "`false`". Default: "`false`".
- **JAMAICA_AMS_JARACCELERATOR_EXTRACTION_DIR** specifies where the shared library containing compiled code should be extracted from a bundle. The value may be an absolute or relative path, ending in the system-specific separator ('/' on Unix-Systems, '\' on Windows). The empty path and the symbolic values "`JAR`" and "`TMP`" (case insensitive) are also accepted. A relative or empty path tells the system to extract the shared library in the same runtime location as the JAR containing the accelerated code, i.e., the bundle.. The value "`JAR`" is equivalent to the empty path. The value "`TMP`" denotes a system-dependent default temporary file directory. If the specified extraction directory is not writable, the default temporary file directory is used instead. If the default temporary file directory is the extraction

directory and it does not exist or it is not writable, then the library can not be extracted and the accelerated code is not loaded. Libraries extracted to a specified directory keep their original name and are never deleted, rather they are reused in later executions[2]. Libraries extracted to the default temporary file directory receive a unique name in each extraction and are deleted when JamaicaAMS terminates. Default: "`JAR`".

- **JAMAICA_AMS_NUM_JNI_ATTACHABLE_THREADS** specifies the initial number of Java thread structures that will be allocated and reserved for calls to the JNI funtions JNI AttachCurrentThread or JNI AttachCurrentThreadAsDaemon. Even if this number set to zero it still possible to call these functions. In this case, the Java threads used for these calls are allocated dynamically when needed. However, dynamic allocation will fail once the maximum number of threads has been reached. Default: 0.

[2] An extracted library is reused only if it has the same name as the library in the bundle and, if the library entry in the bundle is signed, if their contents are the same. If the extracted library can not be reused, it is overwritten by the library in the bundle.

2.3.3 Interaction

To interact with JamaicaAMS, Apache Felix Gogo shell is supplied as auto-deploy bundle. After launching JamaicaAMS, type "`help`" into the shell to see the list of the available commands and "`help <command-name>`", to get specific additional information. Figure 2.3 shows a list of the commands in the default setup.

felix:bundlelevel	gogo:cat	cm:createFactoryConfiguration
felix:cd	gogo:each	cm:getConfiguration
felix:frameworklevel	gogo:echo	cm:getFactoryConfiguration
felix:headers	gogo:format	cm:listConfigurations
felix:help	gogo:getopt	
felix:inspect	gogo:gosh	
felix:install	gogo:grep	
felix:lb	gogo:history	
felix:log	gogo:not	
felix:ls	gogo:set	
felix:refresh	gogo:sh	
felix:resolve	gogo:source	
felix:start	gogo:tac	
felix:stop	gogo:telnetd	
felix:uninstall	gogo:type	
felix:update	gogo:until	
felix:which		

Figure 2.3: Apache Felix Gogo shell commands

For further details, please refer to the Apache Felix Gogo Shell documentation [1].

Note that...

GoGo shell will immediately terminate JamaicaAMS running in the background, since the inputs expected by GoGo shell are not provided in this case. This can be avoided by adding the property "`gosh.args=-noninteractive`" in `conf/system.properties`, or using a terminal multiplexer to invoke JamaicaAMS, e.g., "`screen -m -d ./bin/jams &`".

2.3.3.1 Enhanced Interaction with the Advanced GoGo Shell (JLine)

JLine, an advanced GoGo shell, providing that distinct colorizations and rich commands for user interactions, is located in the `bundles.optional` directory. The default GoGo shell, located in `setup/bundle.3/org.apache.felix.gogo.shell-<version>.jar`, can be replaced with the JLine variant to harness advanced capabilities. Ensuring you substitute <version> with the appropriate version number for the path referencing. To have a safe switching, ensure JamaicaAMS is gracefully shut down to prevent any data loss or missfunction during the shell replacement, and create a backup of the existing GoGo shell to facilitate a smooth rollback if necessary. You can then replace the existing GoGo shell with these

bundles `bundles.optional/org.apache.felix.gogo.jline-<version>.jar` and `bundles.optional/jline-<version>.jar` to initiate the switch to JLine.

Note on Usage: While both the JLine and default GoGo shells are useful tools for experimentation or testing phases, they may not be optimized for production environments. It is not recommended to deploy any of them in production setups to avoid unintended disruptions or vulnerabilities.

2.3.4 Exit Codes

Figure 2.4 shows the exit codes that are currently defined and used by JamaicaAMS, followed by a complete list of exit codes which are specific to the Jamaica Virtual Machine.

JamaicaAMS Exit Codes	
0	Normal termination
2	Error while parsing arguments
3	Framework implementation was not found
4	Error while initializing the framework
5	Evaluation timeout has expired
JamaicaVM Standard Exit Codes	
0	Normal termination
1	Exception or error in Java program
2..63	Application specific exit code from `System.exit()`
JamaicaVM Error Codes	
64	JamaicaVM failure
65	VM not initialized
66	Insufficient memory
67	Stack overflow
68	Initialization error
69	Setup failure
70	Clean-up failure
71	Invalid command line arguments
72	No main class
73	`Exec()` failure
74	Lock memory failed
JamaicaVM Internal Errors	
100	Serious error: `HALT` called
101	Internal error
102	Internal test error
103	Function or feature not implemented
104	Exit by signal
105	Unreachable code executed
130	POSIX signal SigInt
143	POSIX signal SigTerm
255	Unexpected termination

Figure 2.4: Summary of the exit codes.

2.3.5 Example Bundles

The JamaicaAMS distribution contains several examples, e.g.,“primes-example”, “primes-budget-example”, and “primes-lower-priority-example”. Each of these 3 bundles does exactly the same: it calculates prime numbers endlessly, and for each 1000 numbers calculated, it prints in the console the time spent on the calculation (see Figure 2.7). However, the “primes-budget-example” bundle has a CPU budget [3] set to 5%. The “primes-example” and “primes-lower-priority-example” do not have CPU budgets, but have priorities NORM_PRIORITY and (NORM_PRIORITY - 1), respectively. The bundles used in these examples can be found in /example/primes-example-<version>.jar.

2.3.5.1 How to install a bundle

After starting JamaicaAMS as shown in Section 2.3.1, bundles can be installed. Only installed bundles are available for execution. A bundle is installed using the GoGo Shell “`install`” command.

```
> install ../example/primes-example-<version>.jar
```

It shows the bundle ID after the bundle has been successfully installed, for example as shown in Figure 2.5. Note: The command’s outputs are truncated to avoid overflowing text.

```
jamaica-ams/setup\$ ./bin/jams
...
____________________________________
Welcome to Apache Felix Gogo

g! install
 ↪  ../example/primes-example-1.1.0.jar
Bundle ID: 23
g!
```

Figure 2.5: Install the primes bundle

2.3.5.2 How to query the bundles

The list of currently installed bundles in JamaicaAMS can be seen using the “`lb`” command. Depending to the OSGi bundle lifecycle, bundles could be in different states, e.g., installed, resolved, or active. For example, Figure 2.6 shows that the “primes” bundle’s state is `Installed`. Note: The command’s outputs are truncated to avoid overflowing text.

[3] For further information about budgets in JamaicaAMS, please refer to Section 8.2.

```
g! lb
START LEVEL 3
   ID|State      |Level|Name
    0|Active     |    0|System Bundle (0.0.0)|0.0.0
   ...
   21|Active     |    3|slf4j-api (1.7.36)|1.7.36
   22|Active     |    3|slf4j-nop (1.7.36)|1.7.36
   23|Installed  |    3|JamaicaAMS Primes Example (1.1.0)|1.1.0
g!
```

Figure 2.6: Query bundles

2.3.5.3 How to start and stop a bundle

A bundle can be started and stopped using the commands "`start #`" and "`stop #`", respectively, where "#" is a placeholder for the Bundle ID. E.g.:

```
> start #
```

and

```
> stop #
```

Figure 2.7 shows the outputs after issuing the commands.

```
g! start 23
Computed 1000 primes in 626ms!
Computed 1000 primes in 903ms!
Computed 1000 primes in 1194ms!
Computed 1000 primes in 1508ms!
Computed 1000 primes in 1796ms!
Computed 1000 primes in 2121ms!
Computed 1000 primes in 2423ms!
Computed 1000 primes in 2725ms!
Computed 1000 primes in 3044ms!
stop 23
Stopping the worker thread
g!
```

Figure 2.7: Example of a bundle that calculates prime numbers

2.3.5.4 How to uninstall a bundle

After stopping a bundle, its state becomes `Resolved` as shown in Figure 2.8. Note: The command's outputs are truncated to avoid overflowing text.

```
g! lb
START LEVEL 3
   ID|State      |Level|Name
    0|Active     |    0|System Bundle (0.0.0)|0.0.0
   ...
   21|Active     |    3|slf4j-api (1.7.36)|1.7.36
   22|Active     |    3|slf4j-nop (1.7.36)|1.7.36
   23|Resolved   |    3|JamaicaAMS Primes Example (1.1.0)|1.1.0
g!
```

Figure 2.8: Stop the primes bundle

The bundle is stopped but it is still existing in the cache of JamaicaAMS. A bundle can be completely removed using the command "`uninstall #`", as shown in Figure 2.9.

```
g! uninstall 23

g! lb
START LEVEL 3
   ID|State      |Level|Name
    0|Active     |    0|System Bundle (0.0.0)|0.0.0
   ...
   21|Active     |    3|slf4j-api (1.7.36)|1.7.36
   22|Active     |    3|slf4j-nop (1.7.36)|1.7.36
g!
```

Figure 2.9: Uninstall the primes bundle

Chapter 3

Tools and Components

In order to interact with JamaicaAMS, and to profit from its functional extensions, it is our recommendation that users employ tools such as the ones listed in this section. Either provided by third parties, part of aicas' portfolio or contained in the JDK, those tools are vital to fully optimize the development and couple the framework to the Industrial IoT (IIoT) environment.

3.1 Development

- **Integrated Development Environment**—The IDE of choice to write the bundles is Eclipse. For a practical "step-by-step" example, please refer to Section 6.
- **JVMTI Interface**—JamaicaVM implements the Java Virtual Machine Tool Interface to support introspection and debugging of a local or remote Java Virtual Machine. This interface can also be used from inside common Java IDEs.
- **Code Analyzers**—aicas recommends as tools for static source analysis PMD and SpotBugs (see Section 3.1.1).
- **Bndtools**—Open source software, that uses bytecode analysis to accurately calculate the dependencies of OSGi bundles. It features a repository model for bundles, that may be referenced at build-time and can be used to satisfy runtime dependencies. For an introduction to the BND set of tools, please refer to the Bndtools website [2].

3.1.1 PMD and SpotBugs

These open source tools are two different static source analyzers that are used for the same purpose: to analyze the code that is produced by a developer.

They detect problematic issues in the code, like missing synchronization, empty catch blocks or possible null pointer dereferences, classifying major or minor bugs according to potential severity. Note that code analyzers do not actively change anything in the code.

PMD as well as SpotBugs follow predefined rulesets, that are built in and compared with the code in a project. These rulesets contemplate aspects like bad practices and rules concerning for instance performance, security and correctness.

A brief introduction on how to use those tools follows.

SpotBugs

SpotBugs can easily be installed through the Eclipse Marketplace. It already contains several hundred rules that can also be extended.

After installation, SpotBugs can be configured to fit a specific category, e.g. performance- or bad practice oriented. Severity levels can be set from *20* (least) to *1* (most) severe, with a further separation on how a bug should be marked (error, info or warning). Single rules can be enabled or disabled for a further, more focused, analysis (see Figure 3.1).

This can be configured under **Window** → **Preferences** → **Java** → **SpotBugs**.

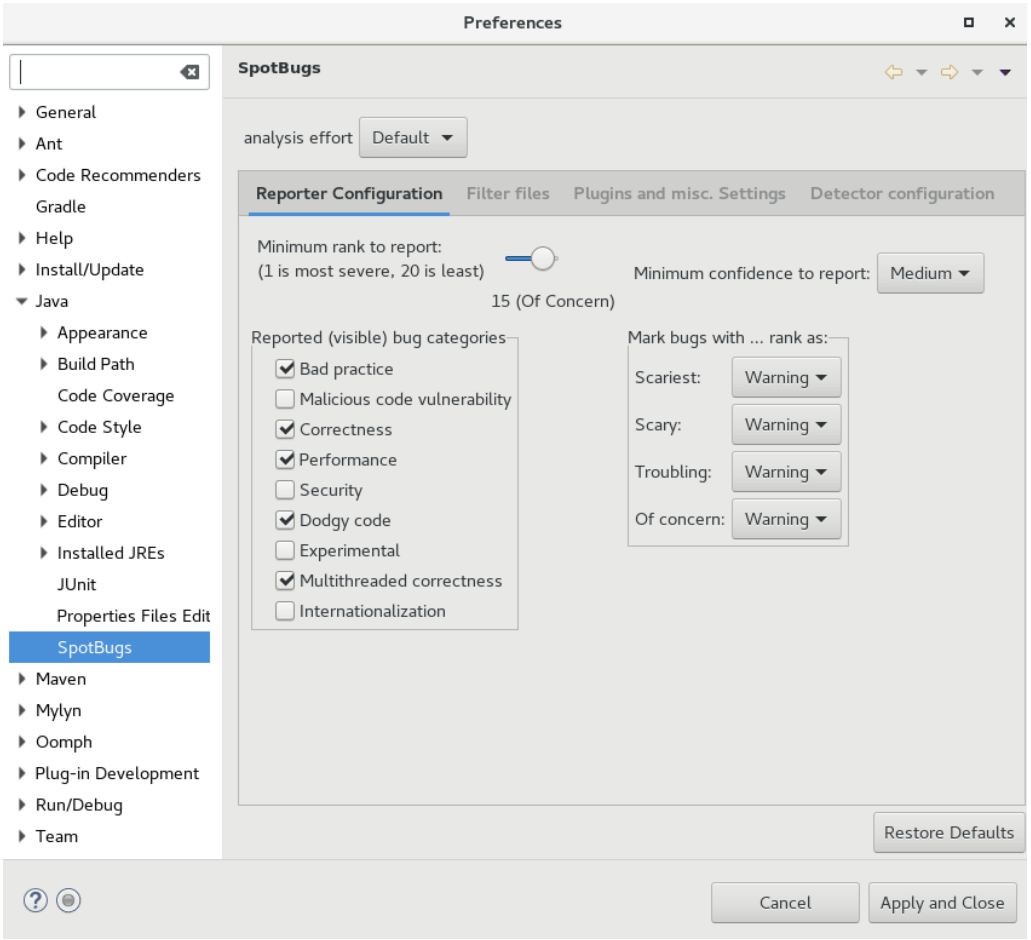

Figure 3.1: SpotBugs can be easily configured in Eclipse

The rulesets in SpotBugs can also be extended under the **Plugins and misc. Setting** tab. These extensions are available online.

To use SpotBugs, right click on the preferred project and select **SpotBugs** → **Find bugs**.

For further details on SpotBugs, please refer to its documentation [9].

PMD

PMD is as user-friendly as SpotBugs. The plugin we recommend can be downloaded from `https://sourceforge.net/projects/pmd/`. It can also be directly downloaded in Eclipse, from `https://dl.bintray.com/pmd/pmd-eclipse-plugin/updates/`, through **Window** → **Install new Software** → **add**. The tool can be configured in a similar way: **Window** → **Preferences** → **PMD** (see Figure 3.2).

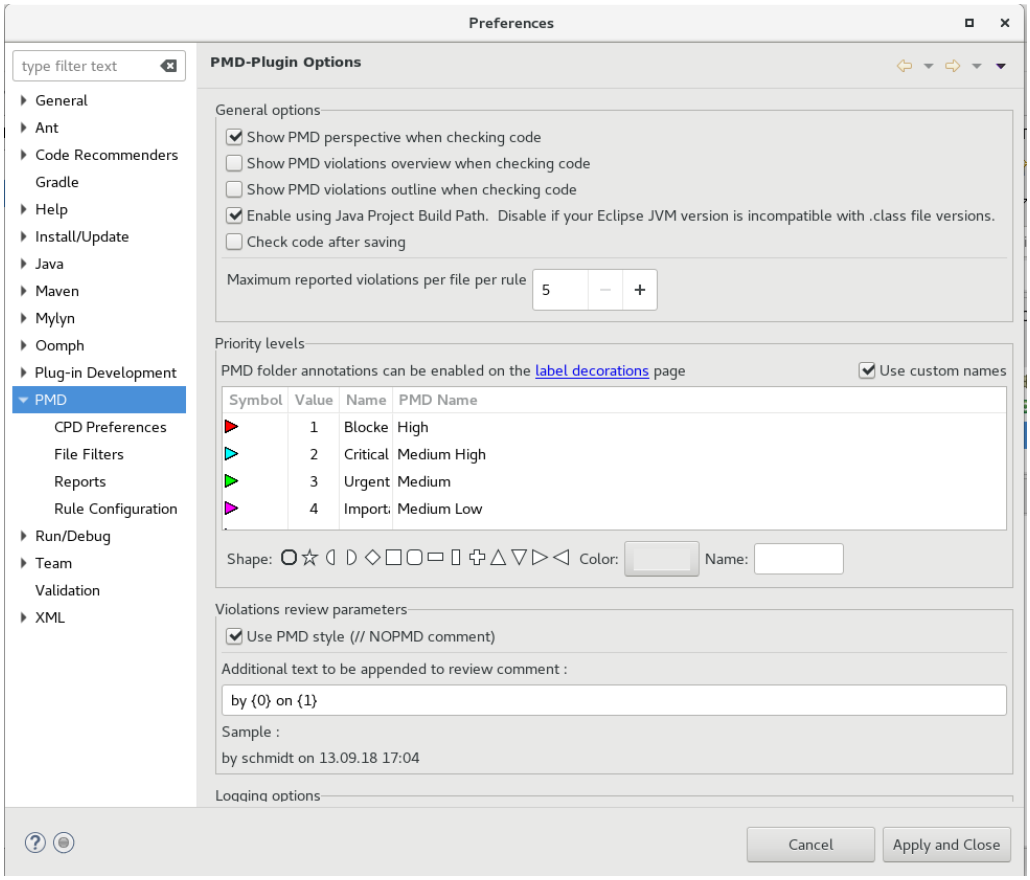

Figure 3.2: PMD is configured in a similar manner to SpotBugs

PMD rules are also separated by categories, e.g. “performance”, and severity, e.g. “blocker”. To use PMD, right click on the preferred project and **PMD** → **Check code**.

For further details on PMD, please refer to the online documentation [7].

Expected Results

After running either of these tools, a list of bugs will be shown in Eclipse. Each bug can be selected, to see more detailed information. Both tools also offer an option to export a bug report as a `.txt` or `.xml` file.

In case both tools are deployed on the same project, the results might overlap: However, they will also be complementary.

3.2 Deployment

- **jarsigner**—Signing and verification tool, that uses key and certificate information from a keystore to generate digital signatures for JAR files (see Section 3.2.1).

3.2.1 jarsigner

JamaicaAMS's target runtime permits the execution of a bundle JAR file that is signed with a private key such that the corresponding certificate can be verified by the public key that was put on the target device (see also 4.1.4.1). To achieve this, the bundle `<bundlename>.jar` must be signed using the following command:

```
> jarsigner -keystore <keystore> <bundlename>.jar <alias>
```

This command uses a key pair generated as in the example below:

Creating a Public/Private Key Pair

The `keytool` that is part of OpenJDK is required to generate a key pair using the following command line:

```
> keytool -genkey -keystore <keystore> -alias <alias> -validity
<validity>
Enter keystore password: <password>
Re-enter new password: <password>
What is your first and last name? John Q. Public
What is the name of your organizational unit? SecurityServices
What is the name of your organization? OEM
What is the name of your City or Locality? Karlsruhe
What is the name of your State or Province?
What is the two-letter country code for this unit? DE
Is CN=John Q. Public, OU=Unknown, O=OEM, L=Karlsruhe, ST=Unknown,
C=DE correct? [no]: yes
Enter key password for <<alias>>
        (RETURN if same as keystore password): <RETURN>
```

Here, `<keystore>`, `<alias>`, and `<validity>` can be chosen freely.

It is extremely important that the generated file, `<keystore>`, remains private. It is recommended to install it only on a machine that is not connected to a network and to limit access to a minimum number of people. Also, the keystore should be protected by a strong password.

3.3 Configuration

- **Bundle Acceleration**—Bundles can be accelerated by compiling methods to machine code. The resultant compiled code is stored in the Bundle JAR and linked when the bundle is loaded. There are several tools that can be used to support this process: the JamaicaJARAccelerator, a profiling AMS, and the Profile Analyzer. For specific information on how to accelerate an OSGi Bundle, please see Section 3.3.1.
- **Bundle Configuration**—The configuration of multiple bundles is made easier through the implementation of the OSGi Configuration Admin service and the Configurator (see Section 3.3.2).

3.3.1 OSGi Bundle Acceleration with JamaicaAMS

JamaicaAMS does not contain a JiT compiler. Instead, JamaicaVM provides, separately from the JamaicaAMS, a means of precompiling Java byte code in an OSGi Bundle to machine code with the JamaicaJARAccelerator. The advantage of accelerating bundles is improved performance. This comes at the expense of a larger image and weaving cannot be used. In any case, weaving is not recommended for realtime embedded systems, since it can adversely affect performance and timing. All methods compiled by the tool are linked into a shared library that is loaded when the JAR, in this case bundle, is loaded by JamaicaAMS. Bundles containing computationally intensive code will benefit the most from acceleration.

The easiest and most common way to use the tool is to simply compile all methods in an OSGi Bundle or JAR. Since compiling methods in a bundle does increase the size of the bundle, this is best used for small bundles. However, in larger bundles it may make sense to be more targeted with compilation. The tool can take a list of methods to compile to provide more selective compilation.

Deciding what to compile becomes increasingly difficult with the size of the bundle. The JamaicaVM tools suite provides the Profile Analyzer to support this process and the JamaicaAMS comes with a special runtime for profiling an application. Together these tools can be used to provide more directed compilation.

Creating A Profile

JamaicaAMS is offered with an additional profiling binary, called the `jamsp` file, that can be used to start the framework. This profiling binary is a version of JamaicaAMS that enables the user to collect information related to the runtime execution. The resulting profiling information is primarily for accelerating OSGi bundles with the JamaicaJARAccelerator.

JamaicaAMS profiling binary

The JamaicaAMS profiling binary collects information on the amount of runtime spent for the execution of different methods. This information is dumped to a file (in `/tmp/jamaica-ams.prof`) after a test run of the application has been performed. This collection of profile information is cumulative; that is, when this file exists, profiling information is appended.

Using A Profile

The JamaicaJARAccelerator is part of the JamaicaVM tools suite. It is used to compile performance critical methods to machine code. All methods compiled by the tool are linked into a shared library that is loaded when the JAR, in this case bundle, is loaded by JamaicaAMS.

Focused Acceleration of an OSGi Bundle

The profiling information is consumed by the Jamaica Profile Analyzer to generate arguments for the JamaicaJARAccelerator, the main part of which is a list of methods to compile. Using the Jamaica Profile Analyzer provides clear documentation about which methods are compiled in the resulting JAR file. More information about how to use these tools can be found in the JamaicaVM User Manual [3]

The XPROF Environment Variable

Another way to profile a running framework is to start JamaicaAMS with the Environment Variable **JAMAICA_AMS_XPROF** enabled. This variable enables collecting simple profiling information using periodic sampling. For detailed information on this subject, please refer to the JamaicaVM User Manual [3] (cf. Section 12.1.2, option **-Xprof**).

To execute JamaicaAMS with the periodic sampling enabled, an Environment Variable needs to be set before JamaicaAMS starts. To export the Environment Variable do as follows:

```
> export JAMAICA_AMS_XPROF=<value from 0 to 1000>
```

Example:

```
> export JAMAICA_AMS_XPROF=100
```

The value to specify is the number of profiling samples to be taken per second, e.g. in the example showed here, 100 samples per second.

This profile is used to provide an estimate of the methods which use the most CPU time during the execution of an application. During each sample, the currently executing method is determined and its sample count is incremented, independent of whether the method is currently executing or is blocked waiting for some other event.

The total number of samples found for each method are printed when the application terminates.

3.3.2 Bundle Configuration

Bundles may load configuration data from the jar, from a local file or from a remote location; and they often require additional configuration, depending on the deployment environment. Furthermore, when multiple bundles require, each of them, particular configuration files, the result is increased

complexity: duplicated files and exception handling code, incoherent file locations and different fetching procedures.

As a solution, OSGi offers specification for the Configuration Admin service and the Configurator bundle. JamaicaAMS packs the Apache Felix implementation of these bundles in `setup/bundle.1` of its distribution tree. This section explains how to use these bundles to access, parse and store configurations in a uniform way.

Note that...

the Apache Felix Configuration Admin and Configurator bundle implement chapters 104 (`https://osgi.org/specification/osgi.cmpn/7.0.0/service.cm.html`) and 150 (`https://osgi.org/specification/osgi.cmpn/7.0.0/service.configurator.html`) of the OSGi compendium specification, being this document the absolute reference.

3.3.2.1 Configurator

Instead of loading the needed configuration data from scattered files, the bundles load the information from the Configuration Admin service. This service acts as a database, simply storing and providing the configurations which are delivered by the Configurator. The role of the Configurator is therefore to fetch and parse configuration files, and then to add the information to the Configuration Admin service.

The Configurator can load files from the local file system, from the bundle jar or from a remote location, supporting the HTTP and FTP protocols. It requires a URL to the file that it should get, and this must contain the protocol to be used (e.g., file:.conf/config1.json, http://www.yourserver.com/standardConfig.json).

Note that...

a list of the URLs for the files to be loaded must be present in the configurator.initial property, which can be modified in the conf/config.properties file.

Configuration File Format

A configuration file may contain multiple configurations, which are identified using a persistent identifier (PID). PIDs are used by the bundles to request a certain configuration from the configuration admin service. Those PIDs that start with the ":configurator:" prefix contain information or instructions that are relevant to the Configurator.

An example of PID and configuration can be seen in the listing below:

```
{
  // Resource Format Version
  ":configurator:resource-version": 1,

  // First Configuration
  "pid.a":
    {
      "key": "val",
      "some_number": 123
```

```
    },

  // Second Configuration
  "pid.b":
    {
      "a_boolean": true
    }
}
```

Configuration files must be written in the JSON format. When those files are not loaded from the bundle jar, it is mandatory to provide a symbolic-name and a version for the configuration (for an example, see table 150.1 in section 150.3.1 in `https://osgi.org/specification/osgi.cmpn/7.0.0/service.configurator.html`).

3.3.2.2 Configuration Admin Service

As already mentioned, the Configuration Admin acts as a database for bundle configurations, storing them persistently (in the JamaicaAMS cache) and distributing them to concerned bundles whenever they are updated. The configurations are handled as java.lang.Dictionary objects and stored in the .properties format.

Retrieving Configurations

There are two ways of interacting with the Configuration Admin: Either synchronously, by fetching the current available configuration through the ConfigurationAdmin interface, or asynchronously, by registering a ManagedService that gets notified whenever its corresponding configuration is updated.

Both methods of accessing the bundle configuration are described in the specification, as they are part of the Configuration Admin service, and are not subject to changes in the different implementations. Thus a bundle that uses the code discussed in the following subsections will work with any implementation of the Configuration Admin.

Synchronous Access

Assuming that an implementation of the Configuration Admin service is installed and a corresponding configuration dictionary is available, a bundle can retrieve and update this configuration using the code presented in te following listing. If no configuration exists, the same code will result in a new configuration.

```
// retrieve the service interface
ServiceReference<ConfigurationAdmin>serviceReference=
context.getServiceReference(ConfigurationAdmin.class);
ConfigurationAdmin configurationAdmin=null;
if (serviceReference!=null)
  {
    configurationAdmin=(ConfigurationAdmin)
    context.getService(serviceReference);
```

```
    // fetch the current configuration
    Configuration configuration=
    configurationAdmin.getConfiguration("ConfigUserTest");
    Dictionary props=configuration.getProperties();
    // if null,the configuration is new
    if (props==null)
      {
        props=newHashtable();
      }
    // set some properties
    props.put (..., ...);
    // update the configuration
    config.update (props);
  }
```

Asynchronous Access

A different approach is that bundles can be reconfigured by a configuration provider while running, thus achieving highly configurable applications. A configuration listener must implement the ManagedService interface, which provides the notification method updated(Dictionary). The listener is then to be published as a service, in association with the configuration name it wants to be updated with.

This approach is also associated with a greater adoption effort, since it imposes a substantial change to the design of the involved bundles. The possibility of updating a bundle with new configurations imposes new logical states on the bundle (which could be called “unconfigured”, “configured” and “running”), while it is in the “active” state. Furthermore, the configuration update occurs in a thread that belongs to the Configuration Admin (thus the name “asynchronous approach”), which may require further handling depending on the code to be executed. For example, if no configuration is present initially, the bundle start method must register a ManagedService and use the update thread to actually initiate the bundle.

The following listing exemplifies a configuration listener published as an OSGi service, which will get notified whenever the associated configuration is changed.

```
private ServiceRegistration configListenerRegistration;

public void start(BundleContext context)
{
  Dictionary props=new Hashtable();
  props.put("service.pid","configurationNameOrPersistanceID");
  configListenerRegistration=
  context.registerService(ManagedService.class.getName(),
  new ConfigurationListener(),props);
}

public void stop(BundleContext context)
{
if (configListenerRegistration!=null)
  {
    configListenerRegistration.unregister();
```

```
      configListenerRegistration=null;
    }
  }
```

Persistence Managers

In cases where the storage logic needs to be overwritten completely (e.g., when the configuration needs to be stored remotely or through a database manager), a persistence manager can be used.

The persistence manager created by the user must provide basic functionality (loading, storing and deleting dictionaries) and must be published as an OSGi service. It needs to implement the interface PersistenceManager present in the package `com.aicas.jamaica.ams.org.apache.felix.cm`, which can be found in the file `<path to JamaicaAMS>/setup/bundle.1/configuration-admin-<version>.jar`.

The Configuration Admin can be configured to use a certain persistence manager by providing its name in the `felix.cm.pm` property.

Note that...

the persistence manager mechanism is specific to the Apache Felix implementation, not being part of the OSGi specification of the Configuration Admin service. The PersistenceManager interface present in the JamaicaAMS distribution is the same one created by Apache Felix and documented in `http://felix.apache.org/apidocs/configadmin/1.6.0/org/apache/felix/cm/PersistenceManager.html`.

The developer must however consider the different package name, which is, as mentioned above:

`com.aicas.jamaica.ams.org.apache.felix.cm`.

Chapter 4

Security

The importance of software security cannot be overemphasized in today's world of the Internet of Things. As applications become more complex and interconnected, they also become more vulnerable to various security threats. The OSGi framework, widely used for embedded Java software development, is no exception. JamaicaAMS leverages the Java and OSGi security models to provide a complete security concept for dynamically loaded software.

4.1 Foundations of Java Security

The Java Security Architecture is built around the concept of a "sandbox" model. This model restricts the operations that loaded code can perform, and effectively isolates potentially malicious code from the rest of the system. The key components of this architecture include the bytecode verifier, class loaders, the security manager, authentication, secure communications, and the fundamental part known as Java Cryptogrphic Architecture.

4.1.1 Bytecode Verification

The bytecode verification acts as a gatekeeper for Java interpreter: when Java source code is compiled, it is transformed into an intermediate known as bytecode, which is then interpreted by the JVM during execution. Before the bytecode is interpreted, the JVM performs a verification process to ensure that the bytecode adheres to certain structural constraints. The bytecode verification process generally involves following steps:

- Checks the bytecode to ensure that the Java language specification is not violated, e.g., no nonexisting object fields are accessed or private fields are accessed from outside the class.
- Checks dataflow to ensure that method calls are legally structured, for example, method calls match the expected number and types of arguments.
- Checks whether the bytecode adhears to the structural constraints of Java langauge, for example, ensuring there is no illegal data type conversion.

Based on the inherent characteristics of Java language, this static verification ensures that the bytecode does not contain operations that could potentially break the Java interpreter and harm the system as well.

4.1.1.1 Limitations

JamaicaAMS inherits the security limitations of JamaicaVM, therefore, it does not cover all the functionalities of bytecode verification.

JamaicaVM is not designed for running untrusted code. Byte code that does not fulfill the static and structural constraints laid out in The Java Virtual Machine Specification, Java SE 8 Edition [15, Sections 4.1–4.9], might lead to undefined behaviour of JamaicaVM.

Classfile verification is currently limited to an incomplete pre Java-6 (classfile version 49 and older) style data flow analysis of the bytecode instructions. The verification algorithm is designed to increase compatibility with regards to the order in which classes are loaded. It does not cover all the functionality described in the JVM specification. Consequently, classfile verification is not sufficient to ensure correctness of class files that are produced by untrusted tools, that were tampered with or that are otherwise broken.

4.1.2 Class Loaders

Class loaders play a crucial role of isolating sensitive classes and resources in Java security. They are responsible for loading classes into the JVM when bytecode has been verified. Originally, Java treated local classes and remotely downloaded classes differently in terms of security. Local classes were trusted, but remote ones were not and shall only run in the "sandbox". Class loaders isolates the namespaces for these classes, that is to prevent untrusted code from accessing sensitive classes and resources. Later this distinction no longer holds, and both local and remote ones were treated as untrusted code unless they are authenticated, but class loaders have always maintained the ability to isolate loaded classes.

There are 3 kinds of class loaders:

- Bootstrap Class Loader: Generally a native implemented code that loads the JDK internal classes, typically `rt.jar` and other core libraries located in `$JAVA_HOME/jre/lib` directory. The Bootstrap class loader serves as the parent of all the other ClasLoader instances.
- Extension Class Loader: The child of the Bootstrap class loader that takes responsible for loading the extension of the stardard core Java classes usually from `$JAVA_HOME/lib/ext`.
- System Class Loader: The child of the Extension class loader that takes care of loading all application level classes from the classpath environment variable defined in the option `-classpath` into the JVM,

Class loaders follow the delegation model. When a class loader is requested to load a class, the class loader will first delegate the loading to its parent class loader. If one of its ancestors loads the class

successfuly, the class is visible and shared to the class loader. Only if all its ancesters failed, it tries to load the class itself.

Class loaders provide critical resource security by controlling namespace visibility. Each class loader creates a namespace, and it shares only to its descendants. For example, 2 distinguish class loaders without ancestral relationship create 2 distinguish namespaces when loading the same class file. If the class file have a public static field, each loaded class has its own field and not shared with another.

Class loaders provide isolated namespaces for classes, thus establish a security boundary that prevents potentially unsafe classes from accessing sensitive classes or resources.

4.1.3 Java Security Manager

Java Security Manager provides fine-grained access control by defining security policies with delegated permissions on API level, compared to the coarse-grained access control of class loaders shown in Section 4.1.2. This section gives the brief introduction to the principle primitives involved in Java Security manager: permission and policy.

4.1.3.1 Permissions

Permissions represent access to system resources. A permission is essentially a statement that grants a specific type of access to a particular resource.

Permissions usually require 2 string parameters: `name` and `action`. The meaning of these parameters is determined by the specific permissions. For example, `java.io.FilePermission` requires a file path for the name and a comma-separated value, which may include `read` and `write` operations.

Java comes with a set of predefined permissions (sub-classing base class `Permission`) that cover many common use cases, such as file access, network access, and property access, for example:

- `java.io.FilePermission`, with actions: `read`, `write`, `execute`, `delete`, and `readLink`.
- `java.net.SocketPermission`, with actions: `accept`, `connect`, `listen`, and resolve.
- `java.security.AllPermission`.
- `java.util.logging.LoggingPermission`, with a single `control` action.
- `java.util.PropertyPermission`, with actions `read` and `write`.

For example, the permissions below represent the allowance to read the `file.txt` file, and to connect the specified host with the port, respectively.

```
// Permission to read a specific file
java.io.FilePermission "/path/to/file.txt", "read";

// Permission to connect to a specific host and port
java.net.SocketPermission "www.example.com:80", "connect";
```

4.1.3.2 Policy

A **policy** is a collection of statements (grouped in `grant entry`) that specify what permissions are available to code, either sourced from a particular location or signed by a particular certificate issuer, or executed as particular principals.

Each grant entry binds its permissions to a protection domain, to encapsulate the security characteristics of a domain. Permissions are granted to protection domains, and all classes belong to a single protection domain. All classes loaded from a code origin are associated with the code's protection domain, thus granting them the permissions granted to the code.

The example of policy file grant entry below grants a permission to read the file `file.txt` to the code from the jar `example.jar` file, and a permission to connect the host `www.example.com` at the port `80` to the code signed by `my-common-name`.

```
grant codeBase "file:/path/to/example.jar" {
    permission java.io.FilePermission "/path/to/file.txt", "read";
};
grant signedBy "my-common-name" {
    permission java.net.SocketPermission "www.example.com:80", "connect";
};
```

4.1.3.3 Access Controlling

Access controlling is achieved by checking whether security-sensitive operations have respective granted permissions. Java provides the `SecurityManager` class, `AccessController` class, and `AccessControlContext` class for this purpose.

- The `SecurityManager` is a class that applications can use to implement a custom security policy. When a SecurityManager is installed, operations that require permissions (like reading a file or opening a network socket) will invoke the SecurityManager to check if the operation is permitted.
- The `AccessController` class is used for making access control decisions based on the current context—a complete call stack. It allows applications to determine whether a particular action will be permitted based on the current security policy and context.
- The `AccessControlContext` class collects the ProtectionDomains of all classes on the call stack leading to the invocation of the sensitive operation. It is used to check that each protection domain on the call stack has at least one permission implying (granting) the specific permission being checked by the method.

Whenever a Java application needs to operate a possibly restricted command, it checks with SecurityManager to see if the operation is allowed or denied. If the permissions have been set up correctly, the operation continues, otherwise a runtime `SecurityException` is thrown.

For example, a jar `example.jar` reads a file `/path/to/file.txt`, and possesses read permission as shown the example above. A high-level call stack of what happens when this application tries to read the file is given below.

1. The application invokes the statement to read a file, e.g., `new FileInputStream("/path/to/file.txt")`.

2. The constructor `FileInputStream` is invoked. Within this constructor (or deeper within the Java standard library), before the file is actually accessed, security check will see if reading from the file is allowed.

3. The security check will invoke the class SecurityManager's `checkRead` method.

```
public void checkRead(String file) {
  SecurityManager securityManager = System.getSecurityManager();
  if (securityManager != null) {
      securityManager.checkPermission(new FilePermission(file, "read"));
  }
}
```

4. If the SecurityManager is not installed, the read operation continues.

5. Otherwise, SecurityManager delegates `AccessControlContext` to check the permissions: `AccessControlContext.checkPermission(Permission perm)` by walking up the call stack. The permissions for a given class are determined by effectively using either its signer or code base as a key to look up the associated protection domain to see which permissions have been granted to it.

6. If the security check fails (e.g., the protection domain of the code is not granted the read permission to the specific file), an `AccessControlException` is thrown, as shown below:

```
Exception in thread "ReadExampleThread"
  java.security.AccessControlException:
  access denied (java.io.FilePermission /tmp/read-example.txt read)
```

7. Otherwise, the operation continues without any interruption.

Java's security manager allows for very detailed control over resources. By using the combination of permissions, policy files, and the SecurityManager, developers can specify which operations are allowed for code from different sources, down to specific actions on individual files, network addresses, or system properties.

4.1.4 Java Cryptography Architecture (JCA)

The Java Cryptography Architecture (JCA) is a framework that provides a comprehensive set of APIs and rubost tools for developers to incorporate cryptographic operations in their Java applications. It's a part of the Java Security API and serves as the foundation for Java security functionalities related to cryptography.

The JCA supports a variety of cryptographic operations, including:

- Key generation
- Encryption and decryption
- Digital signatures
- Message digests (hashing)
- Message Authentication Code (MAC)

These operations allow developers to incorporate strong security features into their Java applications with relative ease.

JCA was designed around these principles:

- Algorithm Independence: Allows developers to use cryptographic services without specifying the exact algorithm, making the code more adaptable to changes in cryptographic preferences.
- Provider-Based Architecture: Cryptographic functionalities are offered by providers, which are implementations of cryptographic services. This modular approach allows for flexibility and the addition of new providers as needed.
- Implementation Interoperability: Different implementations of the same algorithm can work together, ensuring seamless integration and compatibility.
- Algorithm Extensibility: New cryptographic algorithms can be added without altering the existing system, ensuring the framework remains up-to-date with the latest cryptographic standards.
- Engine Classes: Predefined classes that offer specific cryptographic functionalities, such as MessageDigest for hashing or Cipher for encryption and decryption.
- Secure Defaults: If a developer doesn't specify a particular algorithm or provider, JCA chooses secure and widely-accepted defaults.

The following are some principle primitives that are commonly used in secure programming in embedded IoT environments.

4.1.4.1 Public/Private Key Pair

Public key cryptography is the base for Digital Signing. It matches a pair of mathematically related keys used by an asymmetric key algorithm: a public (PuK) and a private (PrK) key. The *Public Key* may be distributed freely, while the corresponding *Private Key* should only be known to the user. In the "Java World" keys are often stored in `KeyStores` (`*.jks` files).

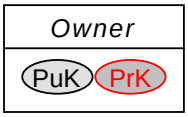

Figure 4.1: A Public/Private Key Pair

Messages signed with the private key can only be verified correctly with the public key. This can be used to authenticate the signer of a message (assuming the public key is trusted). On the contrary, messages encrypted with the public key can only be decrypted with the respective private key. This is generally used to check the integrity of a message.

4.1.4.2 Certificates and Chains

It is mandatory for a secure embedded execution environment to accept trust-only applications, therefore prevent the potentially malicious code from harming the system. Trust can be constructed by verifying the certificates of the code. Java provide a common standard for certificates: X.509. The information commonly stored in a X.509 certificate is shown in Figure 4.2.

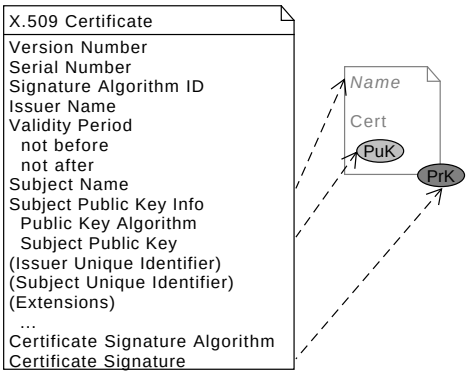

Figure 4.2: X.509 Certificate

Certificates can be arranged in a chain, forming a so called *Chain of Trust*. For this the following conditions must be met:

- The issuer of each certificate (except the last one, the *Root of Trust*) matches the subject of the superordinate certificate.
- Each certificate (except the last one) is signed by the *Private Key* of the superordinate certificate. Hence, its signature can be verified using the *Public Key* of the superordinate certificate.

An example of *Chain of Trust* is depicted in Figure 4.3.

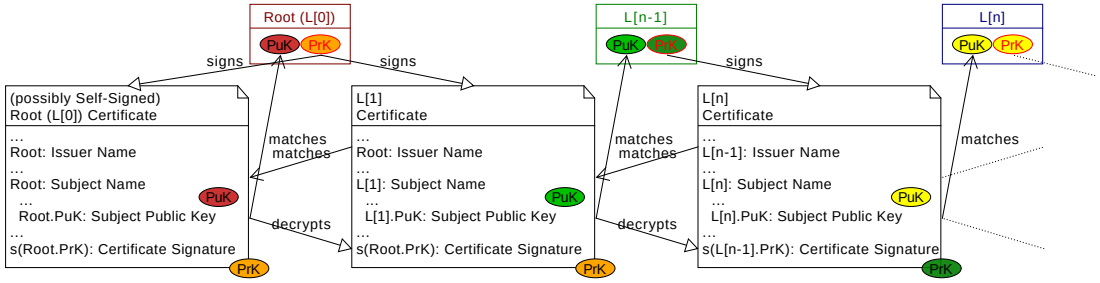

Figure 4.3: A Certificate Chain

Note that...

Even though called “chain” according to the view from the bottommost certificate towards the root, certificate chains can span a tree with multiple instances at any level (except the root). This way, in case a certificate issued at a parent level becomes compromised, all its children can be tracked with their certificates revoked. This breaks the chain back to the root and makes the signer invalid.

4.1.5 Additional Java Security Frameworks

Beyond the foundational security mechanisms in Java, there are several other frameworks and extensions that cater to specific security needs. These include the Java Authentication and Authorization Service (JAAS), the Java Secure Socket Extension (JSSE), and the Java Cryptography Extension (JCE). While we provide a brief overview here, readers interested in a deeper understanding are encouraged to consult the official documentation and references.

4.1.5.1 Java Authentication and Authorization Service (JAAS)

JAAS is a set of APIs for role-based security that allows developers to integrate authentication and authorization into their Java applications.

JAAS provides a way for Java applications to authenticate and enforce access controls upon users. By authentication, it involves verifying the user’s identity, typically by asking for a username and password, but other mechanisms like smart cards or biometrics can also be used. by authorization, it can determine what actions or resources the user is allowed to access, when the user is authenticated. It separates the concerns of user authentication from user authorization, making it a flexible and modular framework.

JAAS is commonly used in scenarios where there's a need to verify the identity of a user (authentication) and then determine what actions or resources the user can access (authorization). For example, there might be a login module that checks usernames and passwords against a database, another that uses LDAP, and another that uses a token-based system. It provides a way to leverage pluggable authentication modules, which can be changed without altering the application itself.

JAAS supports a pluggable and extensible architecture, allowing developers to integrate various authentication mechanisms without changing the core application logic. It also provides a way to establish a security context for the user, which can be propagated across various components and layers of an application.

4.1.5.2 Java Secure Socket Extension (JSSE)

JSSE provides a framework and an API to enable secure Internet communications. It is for secure communication over networks using protocols such as Secure Sockets Layer (SSL) and Transport Layer Security (TLS), and offers functionalities to ensure the confidentiality, integrity, and authenticity of data transmitted between two endpoints in a network.

It's typically used in scenarios where data needs to be securely transmitted over a network, such as encrypted data transmission, secure web communication, client and server mutural authentication commonly in embedded IoT world, and any application that requires encrypted communication.

JSSE is also quite useful in OSGi context:

- Secure Remote Services: If OSGi-based applications offer remote services, JSSE can be used to secure the communication between the service provider and the consumer.
- Bundle Repository Security: When OSGi bundles are fetched from remote repositories, JSSE can ensure that the transmission of these bundles is secure, preventing tampering or unauthorized access during transit.
- Inter-Bundle Communication: In complex OSGi environments where bundles communicate with each other over a network, JSSE can be employed to encrypt this inter-bundle communication.
- Integration with External Systems: OSGi applications that integrate with external systems or services over a network can leverage JSSE to ensure that the data exchanged with these external entities is secure.

JSSE abstracts the complexities of secure communication, allowing developers to focus on application logic. It provides a suite of cryptographic algorithms and protocols, ensuring that data remains confidential and tamper-proof during transmission.

4.1.5.3 Java Cryptography Extension (JCE)

JCE extends the Java Cryptography Architecture (JCA) (See Section 4.1.4) to support stronger cryptographic capabilities.

The distinction between JCA and JCE can be a bit nuanced, especially since they are often used together. Historically, JCA was the core framework for cryptographic operations, while JCE was an extension that provided additional cryptographic functionalities. Over time, the boundary between JCA and JCE has blurred, especially since the JCE has been integrated into the Java Standard Edition (Java SE) starting from Java 1.4.

Some non-exhausted specific features and algorithms that were traditionally associated with JCE:

- Advanced Symmetric Encryption Algorithms, e.g, AES and Triple DES compared to DES introduced in JCA.
- Cipher Modes and Padding Schemes, e.g., CBC, CFB, and OFB, and PKCS5Padding.
- Key Agreement Protocols, e.g, Diffie-Hellman key agreement protocol.
- Password-Based Encryption (PBE).
- Advanced MAC Algorithms, e.g., Hash-based Message Authentication Code (HMAC).
- Permission Classes for Fine-Grained Security Control, e.g., `CipherPermission`, `KeyAgreementPermission`, and `MacPermission` to allow for fine-grained security control over cryptographic operations.
- PKCS#11 standard, which defines a platform-independent API to cryptographic tokens such as hardware security modules (HSMs) and smart cards. PKCS#11 Reference Guide [5] describes how native PKCS#11 tokens can be configured into the Java platform for use by Java applications.

Thus, JCE is typically used in scenarios that require advanced cryptographic operations beyond what's provided by the standard Java libraries.

In summary, JCE provides a provider-based architecture, similar to JCA, allowing for the integration of third-party cryptographic libraries. This ensures that Java applications can leverage the latest and most secure cryptographic algorithms.

4.2 OSGi Security Mechanisms

Although Java security management is powerful, it faces challenges due to the complexity of its management. This often results in users (such as administrators) viewing security as an obstacle, while developers face the overhead of ensuring code compatibility in a secure environment. On the other hand, OSGi security simplifies this situation. It builds on Java's foundational security, but introduces simplified security management mechanisms tailored for loosely coupled applications, making security implementation simpler and more efficient.

This section delves into the comprehensive security mechanisms inherent to OSGi. We will explore the multi-layered security approach of OSGi, from ensuring the integrity of bundles to controlling access to services.

4.2.1 OSGi Class Loading

The OSGi class loading mechanism is an enhanced version of Java's class loader, offering more granular control over symbolic linking within the JVM.

Recall that, Java's traditional class loading mechanism adheres to a set of foundational principles:

- Delegation: Ensures that the request for class loading is initially forwarded to the parent class loader. Only if the parent fails to find or load the class does the child class loader step in.
- Visibility: While a child class loader can access all classes loaded by its parent, the reverse isn't true. The parent class loader remains oblivious to the classes loaded by its child.
- Uniqueness: To maintain the integrity of the loading process, a class is loaded exactly once. This is achieved primarily through delegation, ensuring that a child class loader doesn't reload a class already addressed by its parent.

Unlike the hierarchical parent-child relationship in standard Java class loaders, OSGi class loaders operate on a peer-based model: they expose only those classes that a bundle explicitly chooses to reveal, and rely on classes exposed by peer bundles if they match the desired version. Therefore, it does share all classes like in traditional Java application, instead, it allows to isolate the safety-sensitive classes from the potentially malicious classes by specify the imported and exported packages.

The peer-based model is implemented with a unique mapping table attached to every bundle. This table dictates which class loader is tasked with providing classes from a specific package to the current bundle.

Overall, OSGi class loading reduces the tight coupling and visibility between jar files compared to traditional Java applications, and provides the ability to isolate the access space of classes and allow for multiple versions of java libraries to coexist in the classpath at the same time.

4.2.2 OSGi Security Manager

OSGi's dynamic and modular nature presents distinct security challenges that diverse from traditional Java security paradigms. Specifically tailored to its service-oriented architecture, OSGi's security model offers comprehensive control over bundles and their interconnection. In this section, we will explore the unique security challenges introduced by OSGi, followed by a comprehensive exploration of its solutions, including the bundle protection domain, OSGi-specific permissions, and the intricacies of permission checking.

4.2.2.1 OSGi Security Challenges

OSGi is a modular and dynamic framework that allows applications to be constructed from small, reusable and collaborative components (bundles). This architecture brings forth several security challenges that OSGi needs to address:

- Dynamic Nature: The ability to install, start, stop, update, and uninstall bundles without shutting down the JVM introduces complexities in security management. Ensuring that these operations don't introduce vulnerabilities or compromise the system is challenging, e.g., ensuring only trusted bundles can perform sensitive operations, and security policies and permissions remain consistent after each operation.
- Modular Nature: In the OSGi framework, applications are modularized into "bundles." Bundles can communicate with each other, which means there is potential for malicious or compromised bundles to affect others. Ensuring secure communicate and isolation between bundles is crucial. In addition, Bundles can depend on other bundles, by getting services or importing packages. Ensuring that these dependencies are secure and that a bundle doesn't maliciously exploit another bundle it depends on is crucial. Finally, Each bundle might require specific permissions, thus managing these permissions without costing huge amount of energy and making the system overly complex is challenging.
- Authenticity: Given that bundles can be from various providers, verifying the authenticity of each bundle is essential to prevent malicious code injections.

To address these challenges, OSGi provides a comprehensive security model, including a detailed permission system, bundle signing, and more. We will explain these details in following sections.

4.2.2.2 OSGi Permissions

At its core, Java enforces constraints on third-party code execution by granting specific permissions. By assigning varied permissions, Java can achieve distinct and precise operational restrictions. In contrast, OSGi permissions offer a more dynamic and modular approach, tailored for the unique challenges of component-based systems:

- `Bundle Permission`: Controls which bundle a bundle is allowed to require. It is for dependency security at bundle level.
- `Service Permission`: Regulates bundle's service-oriented operations: service registration and get, i.e., which services a bundle is allowed to publish and/or use.
- `Package Permission`: Controls which packages a bundle is allowed to import and/or export.
- `Admin Permission`: Controls which bundles are allowed to perform sensitive lifecycle operations, such as install, start, stop, and uninstall.

These permissions together with the standard Java security permissions are distributed in difference layers of OSGi environment to form a hierarchical and sequential permission judgment paradigm.

- **System Permission**: Applies to all bundles, serves as a baseline set of permissions that every bundle will have, unless specifically denied. Typically the system permissions are defined in a global policy file referred by the **osgi.security.policy** property. This file defines conditions

under which certain permissions are granted or denied to bundles. It's a way for the system administrator or the framework to impose restrictions or grant additional permissions based on various conditions.

- **Local Permission**: It is bundle's own declaration of what it intends to do—maximum permissions it needs, however, providing more permissions to the bundle is irrelevant because the Framework must not allow the bundle to use them when they are denied in system policy.

  Local permissions are defined in `OSGI-INF/permissions.perm` file (Bundle Permission Resource) within a bundle. The bundle permission resource file is mandatory when OSGi framework is in secure mode. Being empty, syntax-error, or non-existing, will cause the bundle to have `AllPermission`.

  The fine-grained permissions allowed by the OSGi framework are very effective with the local permissions because they can be defined by the developer instead of the deployer.

  As a losely-coupled system, third-party code comes and the deployer need to audit the code. It is difficult to analyse the byte code to get the security requirements of the code, however, it is much easier to just audit the specification, the bunle permission resource located in the bundle. It forms the security contracts between the framework and third-party code.

- **Implied Permission**: Given by the framework automatically when a bundle is installed. They are required for normal operation, such as File permission for bundle persistent storage area, and Package permission for importing `java.*` packages.

The **Effective Permissions** for a bundle are the union of its implied permissions and the intersection of its local permissions and the system permissions. This means that for a bundle to have a particular permission, it must be granted both locally and globally. If either the local or global permissions deny a specific action, the bundle won't be able to perform that action. If the local permissions declare a certain action and the system permissions also allow that action, then that action is part of this intersection.

$$\text{Effective} = (\text{Local} \cap \text{System}) \cup \text{Implied}$$

4.2.2.3 Bundle Protection Domain

In OSGi, each bundle operates within its own protection domain. This domain determines the permissions granted to the bundle, ensuring isolation and reducing the risk of unintended interactions between bundles.

- **Definition**: A bundle's protection domain encompasses the set of permissions assigned to it.

- **Implications**: This domain-centric approach ensures that bundles operate within their designated boundaries, preventing unauthorized actions.

4.2.2.4 Conditional Permission Admin

OSGi defines Conditional Permission Admin Service as an enhanced version of Java Security Manager.

Conditional Permission Admin Service supports flexible condition definition to grant permissions to bundle's protection domain. In traditional Java security architecture, when a permission checking request is issued for an operation, JVM checks whether all classes (i.e., all protection domains where classes are located) referred in the call stack have the required permission. Permissions are granted to a protection domain based on 2 conditions: location of the code base and signer of the code base. Compared to Java's simplistic conditions, Conditional Permission Admin Service improves on this by introducing an abstract condition concept, which supports permission granting based on arbitrary conditions.

Thus, Conditional Permission Admin Service allows for dynamic assignment of permissions based on fine-grained conditions. For example, we can create a condition to only grant permissions based on license status via remote server communication. This provides a flexible way to manage security policies, ensuring that bundles only get the permissions they absolutely need.w

When a security check is performed at runtime:

The framework first checks if the action being attempted by the bundle is declared in its local permissions. If the action isn't declared in the local permissions, the check fails immediately.

If the action is declared in the local permissions, the framework then evaluates the global OSGi policy file. It traverses the policy entries in order to determine if the action is allowed or denied based on the conditions specified.

The first matching entry in the OSGi policy file determines the outcome. If it's an ALLOW entry that matches, the action is allowed. If it's a DENY entry that matches, the action is denied.

4.2.2.5 Differences from Java's Security Model

While both Java and OSGi aim to provide a secure environment for code execution, their approaches differ significantly due to their underlying architectures.

- **Policy Management**: OSGi's policy management, especially with the Conditional Permission Admin service, offers more granularity and dynamicity compared to Java's static policy files.
- **Permission Checks**: OSGi's dynamic nature requires real-time permission checks, ensuring that bundles adhere to their permissions even when the environment changes.
- **Modularity**: OSGi's modular approach, with its service-oriented architecture, necessitates unique permissions like service and package permissions.

OSGi's security model, while complex, offers a robust framework for managing permissions in a dynamic and modular environment. By understanding its intricacies, developers can harness its capabilities to build secure and modular applications.

4.2.3 Code Signing in OSGi

To ensure the integrity and authenticity of bundles, OSGi supports code signing. Bundles can be signed using a private key, and then verified using the corresponding public key before installation. This ensures that only bundles from trusted sources are installed and run.

The OSGi Security Layer is based on the Java security architecture. It provides the infrastructure to deploy and manage applications that must run in controlled and resource constrained environments. Both JAR signing and the Java security manager are used.

For JamaicaAMS, the primary requirement for security is to ensure that only code that has been approved can run on the framework and that code can only use those facilities which are approved. This means that JamaicaAMS must know how to determine whether or not the source which provided the code has been validated and what internal facilities that code may use. JamaicaAMS uses X.509 certificates and the Java security manager provides a sound security mechanism.

As part of its security mechanisms, JamaicaAMS enables the system owner to determine who may write applications (bundles) for its application management system and to provide fine-grained control of what each application may access in the system. Each application provider is identified via one or more certificates with associated permissions.

4.2.4 Signed JAR File

An OSGi framework authenticates code by checking the signature on the JAR files which contains the code. The signature on the JAR file makes it possible to incorporate a JAR file as a leaf into a *Chain of Trust*. Permission associated with the signature determine which service and APIs a given bundle may use. Figure 4.4 shows the security-related content of a JAR file.

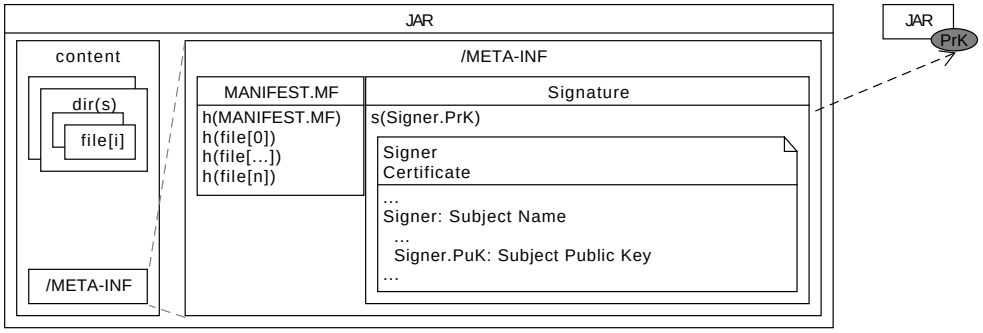

Figure 4.4: A Signed JAR file

However, before a JAR is signed, the creation of a public-private key pair and the corresponding certificate is needed. Furthermore, the public key used for verification of a certificate must be present on the target device. For more information on the process of signing a JAR file, please refer to Section 3.2.1.

OSGi Core Release 8 specifies the following requirements.

JAR Structure and Manifest

OSGi JARs must be signed by one or more signers that sign all resources except the ones in the META-INF directory; the default behavior of the jarsigner tool. This is a restriction with respect to standard Java JAR signing; there is no partial signing for an OSGi JAR.

The OSGi specification only supports fully signed bundles. The reason for this restriction is because partially signing can break the protection of private packages. It also simplifies the security API because all code of a bundle is using the same protection domain.

Signature files in nested JAR files (For example JARs on the Bundle-ClassPath) must be ignored. These nested JAR files must share the same protection domain as their containing bundle. They must be treated as if their resources were stored directly in the outer JAR.

Each signature is based on two resources. The first file is the signature instruction file; this file must have a file name with an extension .SF. A signature file has the same syntax as the manifest, except that it starts with Signature-Version: 1.0 instead of Manifest-Version: 1.0.

The only relevant part of the signature resource is the digest of the Manifest resource. The name of the header must be the name algorithm (e.g. SHA1), followed by -Digest-Manifest. For example:

Signature-Version: 1.0

SHA1-Digest-Manifest: RJpDp+igoJ1kxs8CSFeDtMbMq78=

MD5-Digest-Manifest: IIsI6HranRNHMY27SK8M5qMunR4=

The signature resource can contain name sections as well. However, these name sections should be ignored.

If there are multiple signers, then their signature instruction resources can be identical if they use the same digest algorithms. However, each signer must still have its own signature instruction file. That is, it is not allowed to share the signature resource between signers.

Source: `https://docs.osgi.org/download/r8/osgi.core-8.0.0.pdf`.

4.2.5 Authentication and Permissions

In order to ensure that a tampered bundle will be detected by JamaicaAMS's security mechanism, before a bundle can be distributed for use on the target device, it needs to be signed by the platform provider or a trusted entity using a public/private key signature. All files in the JAR need to be included in the signatures, including resources and not only class files.

On the target device, JamaicaAMS verifies the signature against a public key that is pre-installed on the device. Permissions of all accesses to resources must be checked and resource budgets must be enforced during execution.

Permissions

The OSGi Framework uses `Java 2` permissions for securing bundles. Each bundle is associated with a set of permissions. During runtime, the permissions are queried when a permission is requested through the Security Manager which is a software component in charge of permission validation for the bundles.

The management of the bundle's permissions is handled through "Conditional Permission Admin", "Permission Admin", or another security agent. The Apache Felix implementation includes "PermissionAdmin" and "ConditionalPermissionAdmin", provided by a "framework.security" extension bundle. JamaicaAMS incorporates the "org.apache.felix.framework.security" as "bundle.1" into the framework.

4.3 Configuring Security for JamaicaAMS: A Step-by-Step Guide

JamaicaAMS's security framework is built upon the foundational principles of both Java and OSGi. While previous sections provided an overview of the security mechanisms inherent to Java and OSGi, this section delves deeper into the specific security configurations tailored for JamaicaAMS, elucidated with practical examples.

4.3.1 Initial Setting Up

Security is disabled when the system property `org.osgi.framework.security` is undefined. To enable security of JamaicaAMS, you need to establish a system configuration file with the following specifications:

```
org.osgi.framework.security=osgi
java.security.policy=<path to a Java global policy file>
jamaica-ams.security.policy=<path to an OSGi global policy file>
org.osgi.framework.trust.repositories=<path to a trust store>
```

By default, JamaicaAMS extends full permissions to all bundles operating within its environment. However, individual bundles have the flexibility to limit these permissions by specifying local constraints. The auto-deployed Java global security policy file, located at `conf/java.global.policy`, delineates these broad permissions, as showcased below:

```
grant {
    permission java.security.AllPermission;
};
```

On top of this, the OSGi global policy file, located at `conf/osgi.global.policy`, also grants all permissions without reservation:

```
ALLOW {
    (java.security.AllPermission "*" "*")
} "Give AllPermission to all Bundles"
```

Lastly, the trust repositories, defined by the **org.osgi.framework.trust.repositories** property, should encompass the trust anchors that JamaicaAMS relies on. For an OSGi bundle to be considered as properly signed, it must be authenticated using a certificate present in this trust store. Furthermore, the bundle's integrity is ensured only if it can be successfully decrypted using a public in this trust store. In the OSGi global policy file, you have the flexibility to adjust permissions, e.g, for lifecycle of bundles, based on whether a bundle is signed or based on the bundle's location.

In this section, we detailed the initial setup required to enable security within JamaicaAMS. By default, security remains disabled unless the system property `org.osgi.framework.security` is defined as `osgi`. Once enabled, JamaicaAMS provides a comprehensive security framework, granting all bundles full permissions by default. However, this broad access can be fine-tuned at the individual bundle level, allowing for a more granular security approach. The system relies on both Java and OSGi global policy files to outline these permissions. Additionally, the trust store plays a pivotal role in ensuring that only properly signed and authenticated OSGi bundles operate within the environment. This setup ensures a robust and flexible security mechanism, allowing developers to strike a balance between accessibility and protection.

4.3.2 Generating Self-Signed Certificates

To ensure the integrity and authenticity of bundles installed on JamaicaAMS, it's essential to sign them using a private key and then verify them using a trusted certificate. This section provides a step-by-step guide on generating a keystore, exporting a certificate, and creating a truststore with a self-signed certificate, which can be used for this purpose.

4.3.2.1 Generating a Keystore with a Key Pair

Ensure you have the Java Development Kit (JDK) installed on your machine, as we'll be using the keytool utility that comes with it.

1. Open a terminal or command prompt.
2. Navigate to the root directory of the JamaicaAMS distribution.
3. Use the following command to generate a key pair and store it in a keystore:

```
> keytool -genkeypair -v
-keystore my-keystore.jks -keyalg RSA
-keysize 2048 -validity 365
-alias my-key-alias
-dname "CN=my-common-name, ou=my-organization-unit,
o=my-orgnization, l=my-city, c=my-country-code"
```

- -keystore my-keystore.jks: This specifies the name of the keystore where the key pair will be saved.
- -keyalg RSA: This specifies the key algorithm to be used (RSA in this case).
- -keysize 2048: This defines the size of the key.
- -validity 365: This sets the validity period of the key pair in days.
- -alias my-key-alias: This sets an alias for the key pair, which will be used later when signing bundles.
- -dname "CN=my-common-name, ou=my-organization-unit, o=my-orgnization, l=my-city, c=my-country-code": This Distinguished Name contains the details for the certificate, later you could assign specific permissions based on the distinguished name.

This command creates a new keystore named `my-keystore.jks`, generates a key pair (private and public keys) using the RSA algorithm, and stores the key pair in the keystore protected by passwords. Set a password for the keystore when prompted. Remember this password, as you'll need it later.

4.3.2.2 Generating a Self-Signed Certificate

Use the following command to generate a self-signed certificate:

```
> keytool -exportcert -v
-alias my-key-alias -file my-certificate.crt
-keystore my-keystore.jks
```

- -alias my-key-alias: Use the same alias as specified during key pair generation.
- -file my-certificate.pem: This specifies the name of the file where the certificate will be saved.

Enter the keystore password when prompted.

NOTE: For the sake of simplicity in this guide, to facilitate the access to the truststore in JamaicaAMS, it is recommended not to set a password (i.e., pressing `Enter` key directly when password prompted) for the truststore

This command actually exports the certificate (public key) associated with the alias `my-key-alias` from the keystore `my-keystore.jkd` to a file named `my-certificate.crt`.

We use self-signed certificates in this guide since they are good enough for testing and local development purpose, but they should not be used in production environments. For production, it is mandatory to obtain a certificate from a trusted Certificate Authority (CA). You need generate a Certificate Signing Request (CSR) from the keystore and send it to a CA, and import the response (a certificate signed by CA using CA's private key) to your Trust Store, therefore, you use a chain of certificates (where CA's self-signed certificate is the ancestor on the chain) instead of the self-signed certificate.

It is a good practice to verify the contents of the certificate to ensure it was created correctly.

```
> keytool -printcert -file my-certificate.crt
```

4.3.2.3 Importing the Self-Signed Certificate into a Truststore

```
> keytool -importcert -file my-certificate.crt
-keystore my-truststore.jks
-alias my-key-aicas
```

This command creates a truststore named `my-truststore.jks` and imports the certificate `my-certificate.crt` into it.

Up to now, we have a key pair stored in our keystore `my-keystore.jks` and a self-signed certificate in our truststore `my-truststore.jks`. Next we will use the private key from the keystore to sign our bundles and the certificate to verify them on JamaicaAMS.

4.3.3 Signing a Bundle

Again, ensure you have the Java Development Kit (JDK) installed on your machine, as we'll be using its another utility named jarsigner that comes with it.

Taking the auto-deployed bundle `primes` located at `examples/primes-<version>.jar` file as example, sign this bundle using the keystore `my-keystore.jks`:

```
> jarsigner -keystore my-keystore.jks
-signedjar primes-<version>-signed.jar
primes-<version>.jar my-key-alias
```

This command users the `jarsigner` to sign the `primes-<version>.jar` bundle with the key pair associate with the alias `my-key-alias` in the keystore `my-keystore.jdk`. The signed bundle will be saved as `primes-<version>-signed.jar`.

Before deploying the signed bundle to JamaicaAMS, it is a god practice to verify its signature to ensure the bundle is signed properly:

```
> jarsigner -verify -verbose -keystore my-keystore.jks
-certs primes-<version>-signed.jar
```

This command checks the validity and other details of the signed jar file, as shown below:

```
[entry was signed on 14/8/23 20:39 PM]
      X.509, CN=JamaicaAMS, OU=Engineering, O=aicas GmbH,
      L=Karlsruhe, ST=Baden-Wurttemberg, C=DE
      [certificate is valid from 14/8/23 20:40 AM
      to 14/8/24 20:39 AM]
```

4.3.4 Configure JamaicaAMS to Trust the Signed Bundle

Recall that JamaicaAMS security has been activated as illustrated in Section 4.3.1, we can now configure JamaicaAMS to trust and grant permissions to the signed bundle.

To configure JamaicaAMS to trust signed bundle, we need to update the **org.osgi.framework.trust.repositories** property to point to the previously generated truststore `my-truststore.jks` file, therefore, JamaicaAMS will utilize the truststore to verify the signed bundle.

```
> org.osgi.framework.trust.repositories=<path to
my-truststore.jks>
```

4.3.5 Configure JamaicaAMS to Grant OSGi Global Permissions

To grant permissions to bundles signed by a specific signer and deny permissions to other bundles, we can update the `osgi.global.policy` file as follows:

```
ALLOW {
    (org.osgi.framework.AdminPermission
    "(signer=O=aicas GmbH,*)" "lifecycle")
} "Allow installation of properly signed Bundles"

DENY {
    (org.osgi.framework.AdminPermission "*" "lifecycle")
} "Deny installation of all other Bundles"

ALLOW {
    (java.security.AllPermission "*" "*")
} "Give AllPermission to all Bundles"
```

In the policy above, the first rule allows bundles signed by `O=aicas GmbH` to have lifecycle permissions, i.e., they can be installed, started, stopped, etc. The second rule denies lifecycle permissions to all other bundles, and the third rule grants all permissions to all bundles, as defined by the default deployment of JamaicaAMS, and hereby we refine this board access to necessary permissions using first 2 rules.

1. As a next step, the signed bundle will be installed and demonstrated. To do so:

- Open a terminal, go to the setup directory of the JamaicaAMS distribution and start JamaicaAMS by entering "`./bin/jams`"
- Enter "`install ../example/primes-<version>-signed.jar`"
- Enter "`start`" ID and "`stop`" ID to run and stop the bundle

2. After running the signed versions of the bundle "primes", try to install the unsigned bundle `example/primes-<version>.jar` to get the security check and the *"Access denied"* message displayed.

3. To see the IDs, you can enter "`lb`" (for "list bundles")

With these settings, unsigned bundles or bundles signed by other entities will be denied lifecycle permissions, ensuring that only specific signed bundles (i.e., trusted bundles) can be installed and executed.

4.3.6 Configure JamaicaAMS to Grant OSGi Local Permissions

The global policy file to assign the same set of permissions to all bundles. While this approach provides a uniform security policy, it lacks the flexibility to cater to the specific needs of individual bundles and does not support dynamic permission management when bundles are installed or uninstalled.

Local permissions in OSGi allow bundles to specify the exact permissions they require to operate. These permissions are defined in a policy file within the bundle. In this section, we'll walk through two examples to demonstrate how to set up local permissions for specific bundles in JamaicaAMS.

4.3.6.1 Grant Local Permissions to a File Write Bundle

The first bundle we'll consider is designed to create and write the text "Hello World!" to a file located at `/tmp/local-permission-test.txt`. When the bundle stops, it will delete this file.

To grant the necessary file write and delete permissions, we create a local permission file named `permissions.perm` with the following content:

```
(java.io.FilePermission "/tmp/local-permission-test.txt" "write, delete")
(org.osgi.framework.PackagePermission "org.osgi.framework" "import")
```

This file should be placed in the specific folder `OSGI-INF` within the bundle. The major logic of the bundle is shown in the provided source code snippet.

```
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

public class FileWriteBundleActivator implements BundleActivator {
```

```
    private static final String FILE_PATH = "/tmp/local-permission-test.txt";

    @Override
    public void start(BundleContext context) throws Exception {
        try (BufferedWriter writer = new BufferedWriter(
          new FileWriter(FILE_PATH))) {
            writer.write("Hello World!");
        }
    }

    @Override
    public void stop(BundleContext context) throws Exception {
        java.nio.file.Files.deleteIfExists(java.nio.file.Paths.get(FILE_PATH));
    }
}
```

The syntax for the `permissions.perm` are:

```
conditions  ::= ( '[' qname quoted-string* ']' )*
permissions ::= ( '(' qname (quoted-string
                        quoted-string?)? ')' )+
```

4.3.6.2 Grant Local Permissions to a Host Resolving Bundle

Our second bundle aims to resolve the hostname `localhost` and print the result to the console.

The required network resolve permission is defined in a local permission file named `permissions.perm`:

```
(java.net.SocketPermission "localhost:0" "resolve")
(org.osgi.framework.PackagePermission "org.osgi.framework" "import")
```

Like the previous example, this file should be placed in `OSGI-INF` folder within the bundle. The major logic of the bundle is illustrated in the provided source code snippet.

```
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.net.InetAddress;

public class NetworkBundleActivator implements BundleActivator {

    @Override
    public void start(BundleContext context) throws Exception {
        InetAddress address = InetAddress.getByName("localhost");
        System.out.println("Ping Result: " + address.getHostAddress());
    }

    @Override
```

```
    public void stop(BundleContext context) throws Exception {
        // No cleanup required for this example
    }
}
```

These 2 bundles have their own permissions and have flexibility to update permissions they require. This fine-grained secure management provides a more granular and flexible security model, make it easier to audit and enforce the permissions needed by each bundle, and to enhance the security of the system.

4.4 JamaicaAMS Security Protection

Trustworthiness is paramount for embedded systems, underpinned by five fundamental pillars: Safety, Security, Privacy, Resilience, and Reliability. While ensuring security and privacy for offline devices can be straightforward, it becomes complex for connected devices that demand stringent security, like automotive head units. A device's connection introduces vulnerabilities, making security pivotal for maintaining the other trustworthiness pillars. Thus, the security execution of JamaicaAMS is crucial for its success.

4.4.1 Common Understanding of Computer Security

Before diving into the details of how security is provided by JamaicaAMS, it is important to understand what is meant by security and what the common attack scenarios of connected systems and how they apply to connected embedded devices. The focus is on computer security:

> "Computer security, also known as cybersecurity or IT security, is the protection of computer systems from the theft and damage to their hardware, software or information, as well as from disruption or misdirection of the services they provide." [14]

The attacks to consider are not only via a network, but also direct access to the hardware. As far as they apply, JamaicaAMS must defend against these attacks. The JamaicaAMS framework must provide, together with the rest of the system, effective counter measures for each of the applicable scenarios.

4.4.2 Common Vulnerabilities and Attacks

The common vulnerabilities and attacks are well documented. Wikipedia describes a large collection of network vulnerabilities and attacks [13]. These are summarized here to provide a basis for the security requirements of the JamaicaAMS framework.

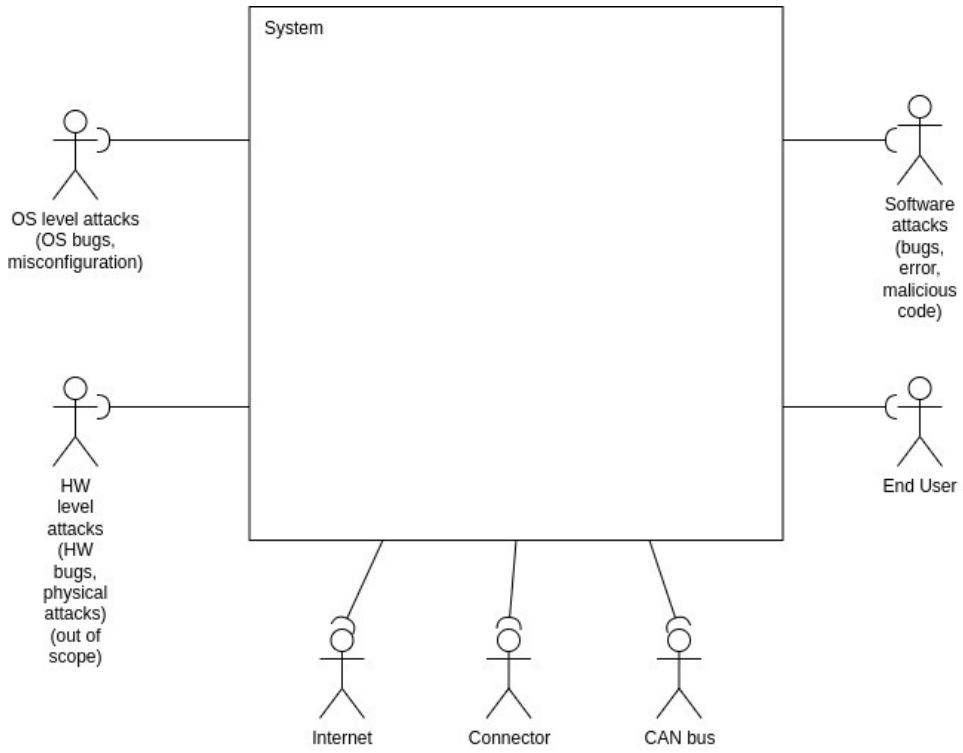

Figure 4.5: System Picture

4.4.2.1 Backdoor

A backdoor in a computer system, a cryptosystem, or an algorithm, is any secret method of bypassing normal authentication or security controls. It may exist for a number of reasons, including by original design or from poor configuration. It may have been added by an authorized party to enable some legitimate access, or by an attacker for malicious reasons. However, regardless of the motive for creating a backdoor, each one creates a vulnerability.

4.4.2.2 Denial of Service (DoS)

Denial of service attacks (DoS) are designed to make a machine or network resource unavailable for its intended users. Attackers can deny service to individual victims, such as by deliberately entering a wrong password enough consecutive times to cause the victim account to be locked, or they may overload the capabilities of a machine or network and block all users at once. While a network attack from a single IP address can be blocked by adding a new firewall rule, many forms of Distributed denial of service (DDoS) attacks are possible, where the attack comes from a large number of points, where defenses is much more difficult. Such attacks can originate from the zombie computers of a botnet, but a range of other techniques are possible including reflection and amplification attacks, where innocent systems are fooled into sending traffic to the victim.

4.4.2.3 Direct Access

An unauthorized user gaining physical access to a computer is most likely able to directly copy data from it. He may also compromise security by making operating system modifications, installing software worms, key loggers, covert listening devices, or using wireless mice. Even when the system is protected by standard security measures, bypassing these may be possible by booting another operating system or tool from a CD-ROM or other bootable media. Disk encryption and Trusted Platform Module are designed to prevent these attacks.

4.4.2.4 Eavesdropping

Eavesdropping is the act of surreptitiously listening to a private conversation, typically between hosts on a network. For instance, programs such as Carnivore and NarusInSight have been used by the FBI and NSA to eavesdrop on the systems of internet service providers. Even machines that operate as a closed system, i.e., with no contact to the outside world, can be eavesdropped upon via monitoring the faint electromagnetic transmissions generated by the hardware; TEMPEST is a specification by the NSA referring to these attacks.

4.4.2.5 Spoofing

Spoofing is the act of masquerading as a valid entity through falsification of data, such as an IP address or user name, in order to gain access to information or resources that one is otherwise unauthorized to obtain. This can be done in tandem with phishing (compare Section 4.4.2.8), and can lead to privilege escalation.

4.4.2.6 Tampering

Tampering describes a malicious modification of a system. “Evil Maid” attacks and security services planting of surveillance capability into routers are examples. Preventing these may require hardware changes.

4.4.2.7 Privilege Escalation

Privilege escalation describes a situation where an attacker with some level of restricted access is able, without authorization, to elevate his privileges or access level. For example, a standard computer user may be able to fool the system into giving them access to restricted data; or even to “become root” or obtain administration privileges and thereby obtaining unrestricted access to a system.

4.4.2.8 Phishing

Phishing is the attempt to acquire sensitive information such as user names, passwords, and credit card details directly from users. Phishing is typically carried out by email spoofing or instant messaging, and it often directs users to enter details on a fake website whose look and feel are almost identical to the legitimate one.

4.4.3 Attack Levels

As depicted in Figure 4.6 there are several levels of attack surfaces. Together with the possible attacks they span a matrix as in Table 4.1. That matrix will provide guidance throughout this document. In the matrix a ✗ means “not (yet) covered”, a ✓ means “covered”, a ● means “put out of scope”, and a ⇝ means “described in the surrounding chapter”.

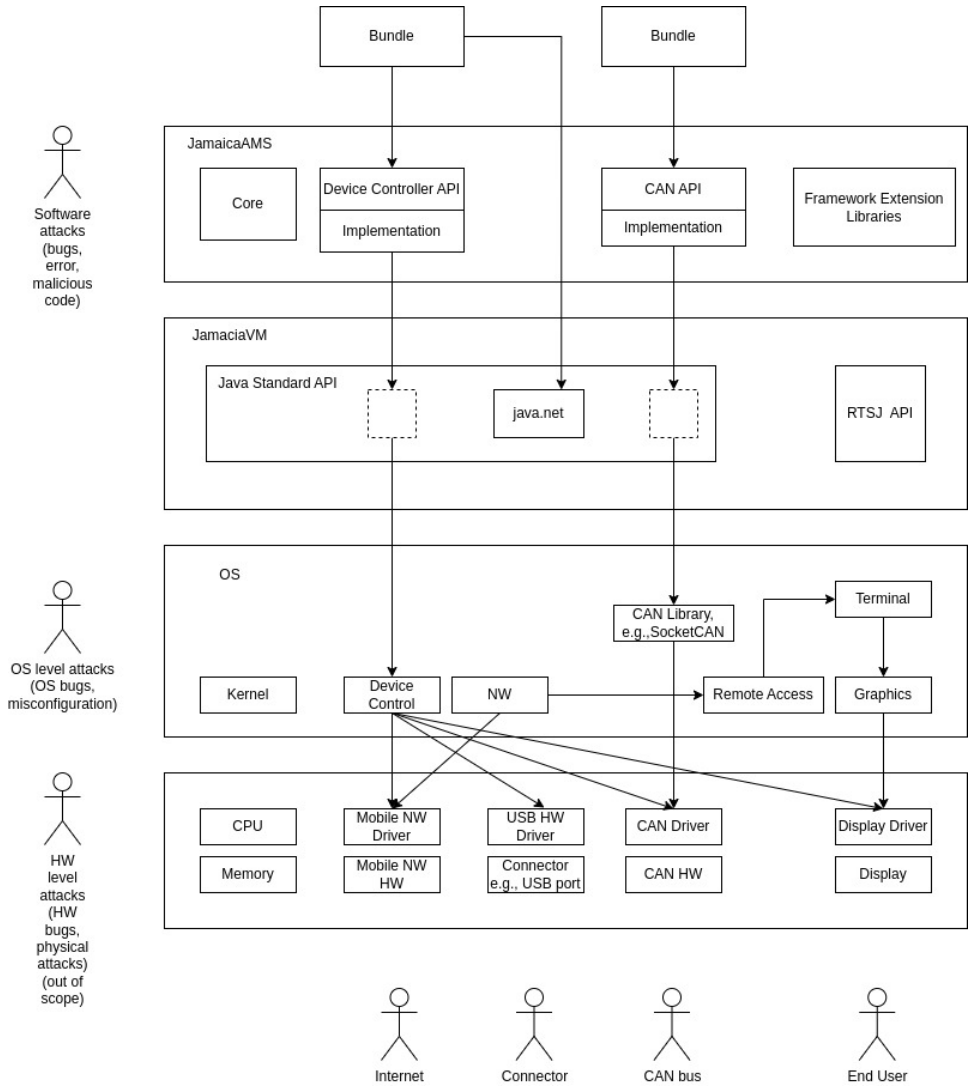

Figure 4.6: Security Architecture

4.4.3.1 Hardware Attacks

Hardware attacks include physical destruction of the device, opening sealed electrical connectors, but also the application of technical tools to gather information about the hardware architecture and hidden secrets. Since the scope of this document is software platforms, such as JamaicaAMS, where there are limited means of guarding against physical access to the device, these must be considered separately in the hardware architecture.

Table 4.1: Coverage Matrix

	Backdoor	Denial of Service	Direct Access	Eavesdropping	Spoofing	Tampering	Privilege Escalation	Phising
Hardware Level	✗	✗	✗	✗	✗	✗	✗	✗
Firmware Level	✗	✗	✗	✗	✗	✗	✗	✗
Application Level	✗	✗	✗	✗	✗	✗	✗	✗

4.4.3.2 Firmware Attacks

Firmware is vulnerable to known issues due to errors in the OS software or a misconfiguration of the system widening the attack surfaces. Since most firmware and OS software is written with unmanaged languages, the scope for error is quite large. Buffer overruns are particularly problematic. The only real defense is a proper design process with thorough verification.

4.4.3.3 Application Level Attacks

The most obvious attacks can take place at application level as depicted in Figure 4.6, since a full programming interface is provided. Aside from using a managed language to reduce the scope of programming errors, three techniques are available for reducing this risk: modern language, a fine-grained security model, and author identification via code signing. The type-safe language verifies the intermediate code before execution and runtime range-checking ensures that memory access safety; the permission system can be used to provide minimal access for a given program class and a signature can be used to find the source of an error to audit the processes used to create software.

4.4.4 Deriving Attack Scenarios

Given the aforementioned attacks, a set of plausible attack scenarios for a connected embedded device, taking automotive head-unit as an example, can be derived. This forms the basis for security requirements for JamaicaAMS. In any case, it should be noted that this list, and in fact any list, cannot be complete! As in war, attacks and defenses evolve continuously. It is not uncommon to strike a balance between security and performance.

4.4.4.1 Backdoors

Any hardware and software component might keep a backdoor open, which also can be attacked. Though limiting permissions helps by preventing code that does not need direct network access from opening a backdoor, this attack can only be fully prevented by tracking the functionality of the system soundly. The responsibility of not providing a backdoor must be enforced by contract on each and

every component provider. The functionality of the system must be tightly specified and verified with the running system.

4.4.4.2 Forced System Breakdown by Signal Input (DoS)

Any open interface, e.g., network, bus, or display, can be flooded with signals to achieve a system overload and denial of service. Hardware and software components must be designed with regard to preventing this kind of attack. Limiting the CPU available to software that may be affected can help keep the system alike, but no general solution is available. This does mean the system must be able to continue to work autonomously for the duration of the attack.

4.4.4.3 System Access

At runtime, the runtime system, both hardware and software, must be sealed against unauthorized access from any direction. Possible attack vectors of access are via code, terminal, network, or configuration. All access should be authenticated, preferably with more than one factor.

4.4.4.3.1 Access by Code / API Since a head-unit is designed to run third-party code, the runtime system must make sure, that this code can only use the functionality actually required for its correct function and no more.

4.4.4.3.2 Access by Terminal / HMI In any system, the access to a terminal is security critical. A shell or application can provide arbitrary functionality. This attack surface can be managed by distributing roles and implementing user authentication on the system level. Limiting API access can also play a role here.

4.4.4.3.3 Access by Network The surface to the network must be seen very critical because this is the most exposed attack surface of all. The designer must assume, that literally anybody can try to access and break the system at this entry. Limiting ports open to traffic originating from the network and restricting outgoing traffic to verified partners can reduce the opportunity for such access. It is better to call out to a known set of hosts, rather than have an open port. Any input must be properly checked in order to reject incorrect input and prevent code injection. This is particularly an issue with access over a web server.

4.4.4.3.4 Access by Configuration The system is designed to allow a certain level of configuration by the integrator end user. Signatures are used to protect the integration configuration. For end users, the scope of the configuration must be clearly limited to ensure that new, unforeseen attack surfaces cannot be created.

4.4.4.4 Listening to DATA IN MOTION

At least some of the data to be processed or communicated is sensitive. The software and even the hardware shall ensure, that this data is not exposed to unauthorized readers. This applies not only to data from and to the internet, but also data from and to the human interface (display) and volatile data in the system itself. Encryption is an important technique for keeping such data secure.

4.4.4.4.1 Data Send to or from the Internet Several possible attack methods must be considered when sending data over the internet. They include sniffing the raw signal as well as trying to fake the endpoint or link into an existing connection. All internet connection must be considered fully exposed.

4.4.4.4.2 Data Send over Other Connectors The use of other connectors carrries the risk of importing malware or at least open an entry point for system manipulation. The number of open connectors should be limited and unneeded connectors should be disabled. Also, the protocol for importing data from a connector must be sound so as not to import malware. Any data leaving the system via an accessible connector must be seen fully exposed.

4.4.4.4.3 Data on the Display The Display may not provide access or visibility to any sensitive data.

4.4.4.4.4 Data in the System Even though the system can be seen as a sealed box and any physical intrusion into the system as a major act of criminality, one should consider that most secret data, user accounts and passwords, private keys, and state-of-the-art technology is a lucrative target for intruders. Security should not stop at the system surface, but also consider data being passed in the system by buses, in memory, and in processor registers or drivers.

4.4.4.5 Spoofing and Phishing

Spoofing, meaning masquerading as a valid entity, and Phishing, stealing valid user credentials, mainly affect the internet connection of the system, but might also affect authorization mechanisms.

4.4.4.6 Tampering with the System Configuration

Each and every unauthorized entity must be prohibited from changing the system configuration. Changing the configuration could change the functionality from what was specified by the system integrator. This is hard to achieve, since it requires a sound specification of both the software and the hardware. Any unauthorized change must be detected and prevented from changing any computation.

4.4.4.7 Tampering with DATA AT REST

Data can be stored on the system. Some of this data, whether its system data or user data containing sensitive information, must be kept secrete. Encryption can be used, but requires a secret key to be stored in the system.

4.4.4.8 Privilege Escalation

Given there already is a set of privileges designed in the system and these are assigned to roles, such as “root” user, each role must be prevented from acquiring any permission it was not designed to have.

4.4.5 Countermeasures

A core responsibility of the JamaicaAMS framework is to guard the system from attacks through malicious code by ensuring that only code of known providence can run on the system. The framework provides a walled garden for running such code. The assumption is that such authorized code is not malicious. Thus, this is the starting point, for targeting Section 4.4.3.3, Section 4.4.4.3.1 for all vulnerabilities as in Section 4.4.2.

4.4.5.1 Managed Programming Language

By employing a managed programming language, here the Java programming language with realtime extensions, a large class of errors that can be made with unmanaged languages, such as C or C++, are prevented by design. This reduces the possibility of erroneous code leading to malfunction, dysfunction, or providing new attack vectors. Most prominently all attacks by pointer manipulation or stack overflows are prohibited.

4.4.5.2 Managed Runtime Environment

An intrinsic feature of a managed language is executed with a well-defined runtime environment. When combined with an interpreter, this runtime environment is often referred to as a *Virtual Machine* providing virtual abstraction of the the system to an application. In case of the JamaicaVM virtual machine, it provides additional means to guard the underlying system. This targets directly Section 4.4.4.3.1.

4.4.5.3 Commonly Used Programming Language

While the implementation of executing byte code is unique in JamaicaVM, the base security mechanisms are not. These are shared by most Java virtual machines and have both a long service history and are under observation of a huge user base. This helps both help minimize defects and to provide a small time window from vulnerabilities being detected to being closed.

4.4.5.4 Java Language Features

The next barrier is an architectural feature of the Java programming language. All functionality is ordered in well-defined, standardized APIs. Specification, documentation and testing of these APIs is supported intrinsically. There is a lot of tools for code analysis to be applied.

4.4.5.5 Service History

JamaicaAMS and JamaicaVM use the security APIs of Java, in particular, the OpenJDK implementations. This means that JamaicaAMS benefits from the service history of OpenJDK. These APIs are used across the internet and are challenged daily. Any failures are addressed immediately by the OpenJDK security team, giving the user an extra level of oversight.

4.4.5.6 API Security

All APIs accessing sensitive functionality, whether belonging to the core VM or the JamaicaAMS extension, are and user APIs providing hardware dependent access to sensitive functionality must and can easily be protected against unauthorized access by the concept of Permissions. More details on permissions can be found in the documentation that comes with JamaicaAMS.

4.4.5.7 Intermediate Summary

These mechanisms alone already provide a certain level of security and robustness to the system, when carefully applied. The attack vectors, as described in Section 4.4.2.1, Section 4.4.2.2, Section 4.4.2.3, and Section 4.4.4.8 are covered and closed to completeness on Application Level as in Section 4.4.3.3 not in the System as a whole, compare Table 4.2.

Table 4.2: Coverage Matrix I

	Backdoor	Denial of Service	Direct Access	Eavesdropping	Spoofing	Tampering	Privilege Escalation	Phising
Hardware Level	✗	✗	✗	✗	✗	✗	✗	✗
Firmware Level	✗	✗	✗	✗	✗	✗	✗	✗
Application Level	✓	✓	✓	✗	✗	👁	✓	✗

4.4.5.8 Application Robustness

JamaicaAMS additionally provides separation between single Applications to guarantee a fair distribution of the system resources, but this is more of a robustnees feature.

4.4.5.9 Validity of Application Code

JamaicaAMS additionally provides mechanism to verify at runtime, that the executed App was authorized to run on the system. This provides an additional level of security. At the same time, this mechanism guarantees, that the application, being once admitted to the system cannot be changed.

4.4.5.10 Environment

So far, a safe runtime environment as in Figure 4.7 has been discussed. This is good, but does not cover all security requirements. Issues at the level of hardware and OS are important as well.

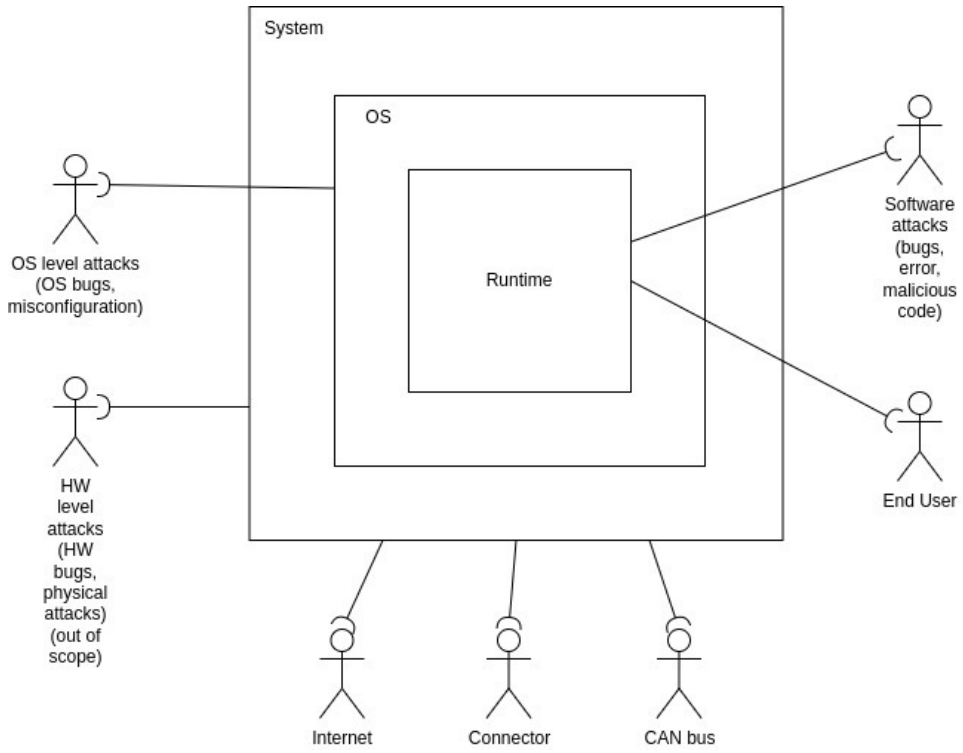

Figure 4.7: Safe Runtime and Environment

4.4.5.10.1 Hardware Measures Measures to secure the Hardware against the common attack scenarios go beyond the scope of what JamaicaAMS can do alone. Suffice is it to say that they must be considered in the system architecture for running JamaicaAMS. See reduced scope in Table 4.3.

4.4.5.10.2 OS Measures Likewise, measures to secure the OS against the common attack scenarios also go beyond the scope of what JamaicaAMS can do alone. Again, this must be part of the complete architecturaly design of a system using JamaicaAMS. See reduced scope in Table 4.3.

Table 4.3: Coverage Matrix II

	Backdoor	Denial of Service	Direct Access	Eavesdropping	Spoofing	Tampering	Privilege Escalation	Phising
Hardware Level	●	●	●	●	●	✗	●	●
Firmware Level	●	●	●	●	●	✗	●	●
Application Level	✓	✓	✓	✗	✗	⊸	✓	✗

4.4.5.11 Initialization

For the initialization of the System, a pristine system state is assumed at power on. This is depicted as the System's "[0]" component in Figure 4.7. Since the runtime environment has been shown to be a safe sandbox in Section 4.4.5.2 and related sections, for complete security one needs to provide a link from the pristine state to the running runtime environment, including a verification of the runtime environment. This links tightly to Section 4.4.4.6 and Section 4.4.4.7.

4.4.5.11.1 Secure Boot To maintain a correct (validated, trusted) system state after starting in the pristine state, the system requires a secure boot process. The extend of this needs to be discussed, since it might include a verification of the hardware. For doing so, the secure boot mechanism requires an **Per Device Entity of Trust**. The secure boot process must further ensure the integrity of all static data in the system. It cannot verify the integrity of dynamic data. The requirements can, for example, be met by hashing the hardware and data items in question and storing the hash in a **Secure Location**.

4.4.5.11.2 Consequence Leaving the open issues from Section 4.4.5.10.1 and Section 4.4.5.10.2 aside, assuming dynamic code only in the Runtime Environment the system is trusted as in Table 4.4.

Table 4.4: Coverage Matrix III

	Backdoor	Denial of Service	Direct Access	Eavesdropping	Spoofing	Tampering	Privilege Escalation	Phising
Hardware Level	●	●	●	●	●	⊸	●	●
Firmware Level	●	●	●	●	●	⊸	●	●
Application Level	✓	✓	✓	✗	✗	⊸	✓	✗

4.4.6 DATA AT REST protection

The open issue in the "Tampering" column of Table 4.4 is the storage of data accessed, created, modified, and deleted during system runtime. This data might or might not be persistently stored between power cycles. However, it cannot be statically verified.

As for Application Data, it might be sufficient to guard it by separation, so that no other Application is able to access it. This is already done by JamaicaAMS. If further security is required, the App could bring its own encryption mechanism, taking into account, that this itself is easily attackable. This level of security is sufficient for sensitive user data, though.

Thinking at the System Level, DATA AT REST includes any part of the system that can dynamically be changed, hence updates to the Software, the Runtime, or the Firmware. To reduce the attack potential of these mechanisms, the mechanism, e.g., for doing a firmware update, must be strong and robust. The system (pristine state) must have verified the mechanism and the update package must be trusted. This links back to Section 4.4.5.11.1, especially the **Per-Device Entity of Trust**.

4.4.7 DATA AT MOTION protection

The same applies to data at motion, assuming common technologies like SSL/TLS to be understood, applicable and strong, both themselves and in their implementation on the system dependent network stack. Given the link to the trust anchor is guaranteed, the mechanisms and credentials needed to close the remaining gap as in Table 4.5 can be devised, Figure 4.8.

Table 4.5: Coverage Matrix IV

	Backdoor	Denial of Service	Direct Access	Eavesdropping	Spoofing	Tampering	Privilege Escalation	Phising
Hardware Level	●	●	●	●	●	✗	●	●
Firmware Level	●	●	●	●	●	✗	●	●
Application Level	✓	✓	✓	✗	✗	✗	✓	✗

4.4.8 Conclusion

JamaicaAMS can provide the necessary system security from the boot loader up into the cloud. This includes all code that runs in JamaicaAMS and all connections to the internet. To finish the picture, JamaicaAMS needs support from the hardware and OS. The loader must be part of the secure boot system and security can be enhanced by providing secure keys for identification and local encryption. For this reason, close collaboration with the system architect is necessary for ensuring full security.

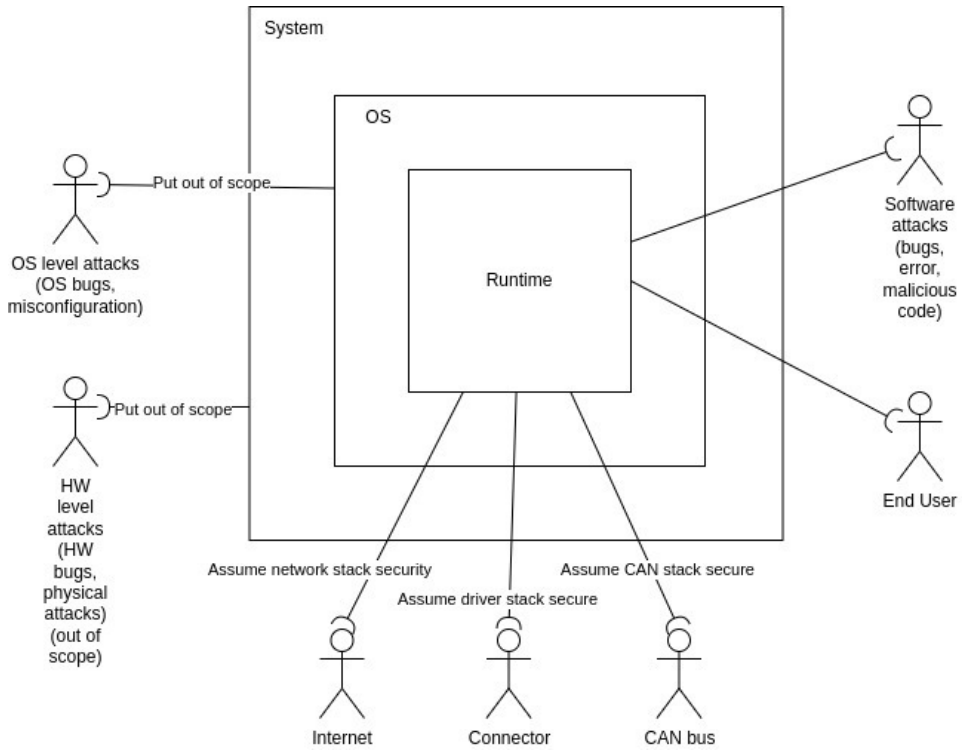

Figure 4.8: Final Picture

Chapter 5

OSGi Framework and Bundles

5.1 Framework Layers

OSGi offers a software layer, running on top of a Java platform, that supports the design and implementation of modular systems. It does that by specifying a secure infrastructure that enables the distribution, interoperability and remote management of application- and service components, called bundles and sometimes also referred as packages.

The central part of the specification is the framework, that defines a model to manage the lifecycle of the bundles, also including a registry, that exposes functionalities made available as services to be imported and exported by the bundles, and an execution environment.

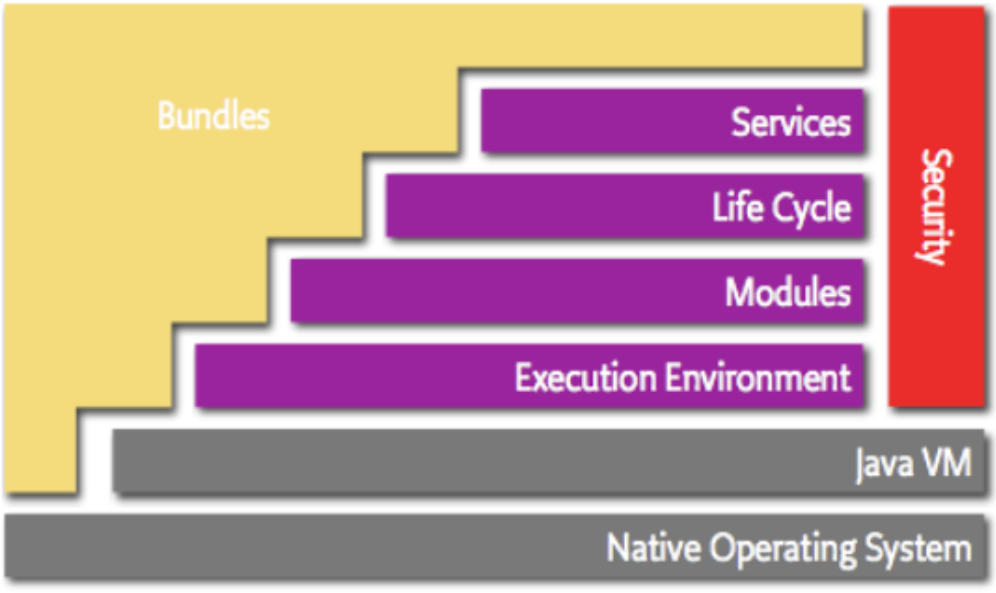

Figure 5.1: OSGi Framework: Layers Overview

Source: `https://www.osgi.org/developer/architecture/layering-osgi/`

5.2 Bundle Lifecycle

The Lifecycle Layer provides an API for controlling the different phases of the bundle operations.

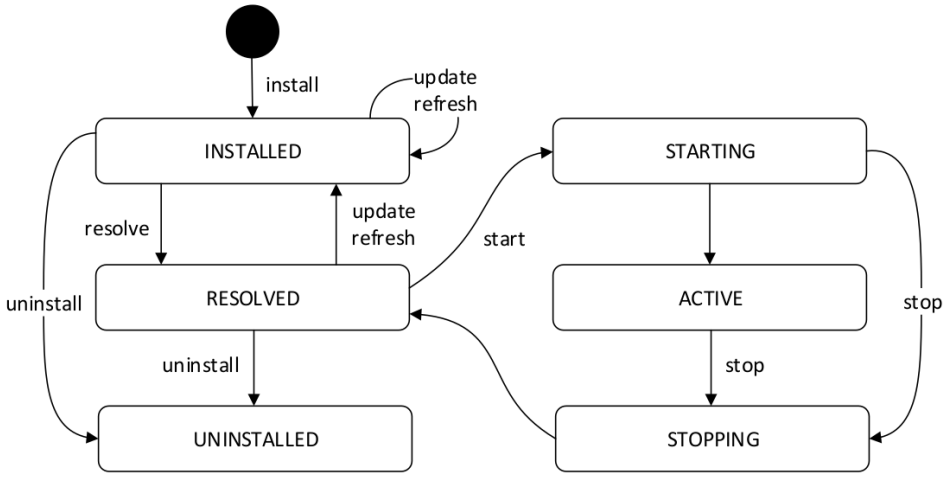

Figure 5.2: Bundle's lifecycle

Figure 5.2 shows the states that a bundle can experience in a lifecycle, as well as the possible transitions it could make to other states. Those can be summarized as follows:

- **Installed:** The bundle was successfully installed.
- **Resolved:** This state evaluates if the bundle is ready (complete), by checking things like the Java version, if the imported packages are available etc.
- **Starting:** This is a transition state in the lifecycle; in this state, the method that activates and starts the bundle is called.
- **Active:** In this state, the bundle has been successfully validated and is running.
- **Stopping:** Another transition state; the method that stops the bundle is being executed and the bundle is being stopped.
- **Uninstalled:** The bundle is uninstalled and cannot be induced into another state.

5.3 Service Orientation

The service layer of the Framework rules the way the bundles dynamically connect and collaborate, through a process known as "publish-find-bind".

The concept of service in the framework's context involves the following aspects:

- **Specification:** An interface that specifies the public methods.
- **Implementation:** A Java class that implements the methods specified in the interface.
- **Registry:** Where the services are included and made available to other bundles.
- **Customers:** The bundles that make use of a given service.

5.4 Controlling the Bundles

The control of the bundles' lifecycle as well as other functionalities can be done through administration consoles or via command line, depending on the implementation. This way, any bundle can be dynamically installed or uninstalled without the application having to be stopped or restarted. Also information about active services, the listing of imports and exports of each bundle and the metadata contained in the headers are shown this way.

As mentioned in Section 2.3.3, JamaicaAMS provides the Apache Felix Gogo shell as an auto-deploy bundle. The command "`lb`" lists the bundles and their present states, as seen below.

```
g! lb
START LEVEL 3
   ID|State      |Level|Name
    0|Active     |    0|System Bundle (0.0.0)|0.0.0
    1|Active     |    1|Apache Felix Log Service (1.3.0)|1.3.0
    2|Active     |    1|JamaicaAMS OSGi Log Writer (1.3.1)|1.3.1
    3|Resolved   |    3|JamaicaAMS Security Provider (1.3.1)|1.3.1
    4|Active     |    2|JamaicaAMS Configuration Admin (1.3.1)|1.3.1
    5|Active     |    2|JSON-P Default Provider (2.0.1)|2.0.1
    6|Active     |    2|Apache Felix Configuration Json (2.0.6)|2.0.6
    7|Active     |    2|Apache Felix Configurator Service (1.0.18)|1.0.18
    8|Active     |    2|Apache Felix Declarative Services (2.2.14)|2.2.14
    9|Active     |    2|org.osgi:org.osgi.service.component (1.5.1.202212101352)|1.5.1.202212101352
   10|Active     |    2|org.osgi:org.osgi.util.converter (1.0.9.202202082230)|1.0.9.202202082230
   11|Active     |    2|org.osgi:org.osgi.util.function (1.2.0.202109301733)|1.2.0.202109301733
   12|Active     |    2|org.osgi:org.osgi.util.promise (1.3.0.202212101352)|1.3.0.202212101352
   13|Active     |    2|JamaicaAMS Policy File Reader (1.3.1)|1.3.1
   14|Active     |    3|Apache Felix Gogo Command (1.1.2)|1.1.2
   15|Active     |    3|Apache Felix Gogo Runtime (1.1.6)|1.1.6
   16|Active     |    3|Apache Felix Gogo Shell (1.1.4)|1.1.4

g!
```

5.5 Enhanced Life Cycle Layer with Forced Thread Termination

JamaicaAMS uses OSGi Life Cycle Management that provides an API to control life cycle operations of bundles. This means that bundles may be installed, started, updated, stopped, and uninstalled during the execution of the framework. This is a key feature for remote software management in JamaicaAMS.

In a conventional OSGi implementation, it is the programmer's responsibility to ensure that all threads spawned in the bundle are terminated, when a bundle is stopped. This can make the entire system unstable, when a bundle has an infinite loop or blocks an I/O stream. In particular, applications that periodically acquire or transfer data, such as Internet of Things (IoT) applications, must be

carefully written and tested. Not terminated threads may occupy limited system resources, cause the system to become unresponsive and ultimately the whole system to fail.

JamaicaAMS provides enhanced capability of terminating threads spawned in a bundle. This enhances, in most cases, its chances to terminate uncooperative and faulty bundles which do not terminate quickly when stopped.

Note that...

Threads with non-terminating `finally` blocks are not guaranteed to be terminated. More details can be found in the Realtime Java Standard [8].

This capability is provided using features from the standard for Realtime Java. RTSJ [8] provides two mechanisms for asynchronous control flow, a general transfer mechanism called Asynchronous Transfer of Control (ATC), which provides a means for stopping some calculation prematurely, and an abort mechanism to safely terminate any task called Asynchronous Task Termination (ATT). Whereas ATC provides a general transfer mechanism for code declared to be interruptible, ATT is designed to safely terminate code that is not explicitly programmed for being interrupted asynchronously. Compared to ATC, that relies on the programmer to declare the section of code for termination and handle the resource cleanup, ATT handles these for programmers. It enables ending computation more generally and always results in task termination.

By taking advantage of ATT, JamaicaAMS provides a more robust life cycle management than conventional OSGi. Bundle termination works more like process termination in that any bundle can be terminated at any time. This is an essential feature for robustness and remote bundle management.

Chapter 6

How to write a Bundle with Eclipse

This section describes how to write an OSGi bundle for JamaicaAMS with the Eclipse IDE.

6.1 Prerequisites

The steps described in this section assume the following knowledge:

- General experience with a Java Runtime Environment
- General experience in writing applications in the Java programming language
- Basic understanding of the OSGi framework, especially of the bundle's interface and lifecycle
- General experience with the Eclipse IDE [11]

The steps described in this section require the following setup to be present:

- An up-to-date version of the Eclipse IDE from [10]. The required minimum version is 3.6, but the latest version is recommended.

6.2 Using PDE

If the Plug-In Development Environment (PDE) [12] is not contained in your Eclipse distribution, it can be obtained using the Eclipse Update manager, by choosing **Menu** → **Help** → **Install New Software**

6.2.1 Create a new Plug-In Project

1. Create a new Plug-in project by navigating to **File** → **New** → **Project**. In the **New Project** wizard choose **Plug-In Development** → **Plug-in Project**

2. In the **New Plug-in Project** wizard on the **Plug-in Project** page enter `HelloWorld` into the **Project Name** textfield.

   Leave the **Project Settings** options unchanged.

   Set the **Target Platform** options to match "This plug-in is targeted to run with: an OSGi framework: standard".

   Proceed by clicking the [**Next >**] button.

3. On the **Content** page leave the **Properties** options unchanged. Make sure that in the **Options** section, the **Generate an activator, a Java class that controls the plug-in's life cycle** checkbox is checked.

   Proceed by clicking the [**Next >**] button.

4. On the **Templates** page you can see a selection of project templates.

   Choose **Hello OSGi Bundle** for now.

   Confirm by clicking the [**Next >**] button.

5. The template **Basic OSGi Bundle** will let you choose a message to be displayed on start and one on stop respectively. Freely choose any or keep with the defaults.

6. End this procedure by clicking the [**Finish**] button. At this point, Eclipse may suggest switching to the **Plug-in Development** *Perspective* because this is the default for Plug-in Projects. Please do so.

6.2.2 Make Yourself Familiar with the UI

Verify that you have successfully finished the first stage by finding the `HelloWorld` project with `src` and a *Manifest* as `META-INF/MANIFEST.MF` in the *Workbench*'s **Packages / Package Explorer** *View*.

If you never have used the **Plug-in Development** *Perspective* before, it might be a good moment to make yourself familiar with it by, e.g., having a look how `META-INF/MANIFEST.MF` is represented.

6.2.3 Implement the Functionality

There is literally nothing that needs to be done. See how `src/helloworld/Activator.java` implements `BundleActivator` with the input you chose in the wizard.

6.2.4 Run the Bundle on the Integrated Framework

As a next step you might want to run the newly created bundle. You can run it in the Equinox OSGi framework, which is a fundamental part of every Eclipse installation.

1. Select the `HelloWorld` project that contains your OSGi bundle.
2. Choose **Run** → **Run Configurations...**. This opens the **Run Configurations** dialog.
3. Create a new **OSGi Framework** configuration.
4. On the **Bundles** tab, please ensure that Equinox is set as runtime framework and select the following bundles:

   For Eclipse 3.6 and 3.7 users:

   - `HelloWorld`
   - `org.eclipse.osgi`

   For Eclipse 4.x users:

   - `HelloWorld`
   - `org.eclipse.osgi`
   - `org.eclipse.equinox.console`
   - `org.apache.felix.gogo.command`
   - `org.apache.felix.gogo.runtime`
   - `org.apache.felix.gogo.shell`
5. On the **Settings** tab, enable the **Clear the configuration area before launching** checkbox.
6. Now click the [**Apply**] button to save the configuration and then press **Run** to launch an Equinox instance with the selected bundles. The **Console** *View* will show you the expected output of the `Activator` class and the OSGi shell.

Since you have started a regular OSGi framework, you can start and stop your bundle by using the commands `start <bundle-id>` and `stop <bundle-id>`. Use the `help` command to get a list of the most important OSGi commands.

6.2.5 Deployment

As a next step you might want to deploy your bundle to a target system. To obtain a `Java Archive (JAR)` file for your bundle, in order to install it into any standard OSGi framework, take the following steps:

1. Select the `HelloWorld` project in the **Package Explorer** *View*.
2. Go to **File** → **Export...**.
3. In the **Export** wizard, on the **Select** page choose **Plug-in Development** → **Deployable plug-ins and fragments**.

4. Proceed by clicking the [**Next >**] button

5. On the **Deployable plug-ins and fragments** page, in the **Destination** tab select the **Directory** option and determine the place where to put the bundle *JAR* by editing the textfield.

   Shortcut Hint:

   - By choosing the *Bundle Directory* of an OSGi framework here, the framework will use the bundle on next start.
   - If the framework has a file install mechanism activated, you can use this option or the **Install into host** option to install the bundle on-the-fly.

6. Click the [**Finish**] button to create the bundle *JAR* file.

6.3 Using M2E

Most Eclipse IDE downloads already include support for the Maven build system M2E. To check, use **Menu** → **Help** → **About** and check if you can see the Maven logo (with the M2E) sign. If Maven support is not yet installed, the following description can be used to install it.

Select **Menu** → **Help** → **Install New Software** menu entry.

6.3.1 Create a new Maven Project

TBD: Add how to create a maven project.

6.3.2 Importing the Examples into Eclipse

The example bundles contained in the JamaicaAMS distribution are Maven Projects. To import these correctly into Eclipse, the following steps have to be taken:

- Extract the example bundle source (zip file)
- In Eclipse, proceed with **Select** → **File** → **Import** → **Maven** → **Existing Maven Projects** (see Figure 6.1).
- In the following window, select **Browse** and navigate to the extracted source folder, clicking the [**OK**] button. Please note that the `pom.xml` will be automatically selected.
- Click the [**Finish**] button and the example bundle will be imported as a Maven Project into Eclipse

After the successful import, the project can be modified.

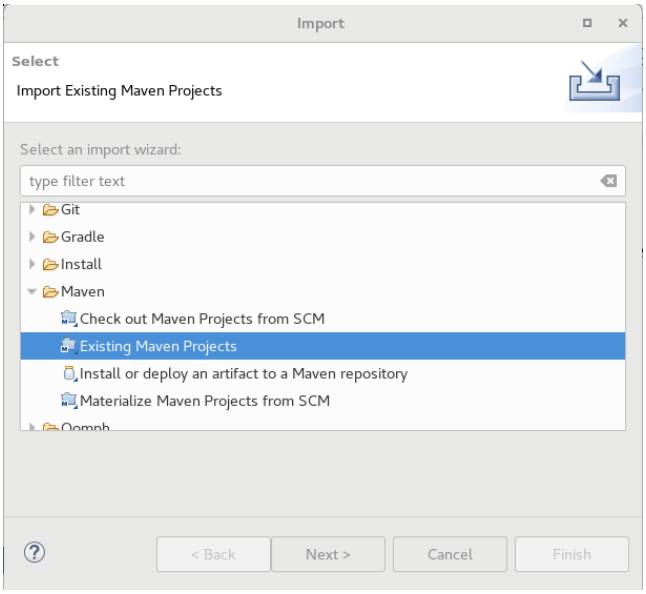

Figure 6.1: Importing the examples in Eclipse

6.3.3 How to build the Examples

TBD: Add how to build the example using M2E UI.

The sources for the examples are included in JamaicaAMS. To be able to build the examples, for instance after some changes are made to the source, Maven has to be installed on the system. The required Maven version is 3.6.0. It is also required to have at least Java 8 and an internet connection.

To build the examples, run:

```
> mvn clean package
```

The bundles can then be found in the `target` directory created by Maven.

6.3.4 Implement the Functionality

TBD: Add how to implement the HelloWorld.

6.3.5 Run the Bundle on the Integrated Framework

TBD: Refer to 6.2.4

6.3.6 Deployment

TBD: Add how to deploy a bundle using M2E.

Chapter 7

Debugging Bundles with Eclipse

This section describes how to debug an OSGi bundle with the Eclipse IDE. Besides the traditional debug facilities in Eclipse for debugging on the local host machine, here the focus is especially set on remote debugging. The latter is particularly required when a bundle is executed and debugged directly on a target platform running JamaicaAMS.

7.1 Prerequisites

The steps described in this section assume the following knowledge:

- General experience with a Java Runtime Environment
- General experience in debugging applications in the Java Programming Language
- Basic understanding of the OSGi framework, especially the Bundle interface and lifecycle
- General experience with the Eclipse IDE, especially how to debug applications using Eclipse IDE

The steps described in this document require the following setup:

- The minimum required Eclipse IDE is version 3.6, but the latest version is recommended.
- Java Virtual Machine (JVM) V5.0 or later must be used, such as JamaicaVM or OpenJDK 8.
- A JamaicaAMS release

7.2 Background

Debugging an OSGi bundle can be accomplished in several different ways for different scenarios, e.g.:

- a bundle is deployed and debugged locally in the OSGi environment hosted by the Eclipse IDE;
- a bundle is running in a custom target OSGi environment apart from Eclipse (e.g. JamaicaAMS) and is debugged remotely;
- the target OSGi framework may run on the local or on a remote host

However, in either case the Eclipse IDE serves as an interface for debugging, which may connect to an integrated or a separate OSGi platform. In the case of JamaicaAMS, the Debuggee, i.e. the bundle to be debugged, is always executed within a dedicated JamaicaAMS framework running on a local or a remote host. In both cases, Eclipse IDE connects remotely to the target JamaicaAMS framework.

In order to facilitate debugging activities, the JamaicaAMS distribution provides a binary application that has been built by Jamaica Builder with JVMTI agent enabled (JVMTI [1]). The application is located in:

- `<JamaicaAMS>/setup/bin/jamst`

The following example shows how to remotely debug a JamaicaAMS built-in bundle (`<JamaicaAMS>/example/primes-example-<version>.jar`) using `jamst` and the Eclipse IDE.

7.3 Setup of the Debugging Environment

To debug the "primes-example" bundle, setup the following environments:

- Create a Plug-in Project from the extracted bundle JAR that is located at `<path to JamaicaAMS>/example/primes-example-<version>-sources.jar` and set a breakpoint (e.g., somewhere in the `PrimesExampleActivator.start()` method). Figure 7.1 shows an example of a breakpoint that stops the execution before checking whether a number is prime or not.
- Copy the `<path to JamaicaAMS>/example/primes-example-<version>.jar` into the folder `<path to JamaicaAMS>/setup/bundle.3`, in order to auto start the bundle. You can also manually export the "primes-example" project as a bundle and install the bundle when JamaicaAMS runs.

7.3.1 Start the Debug Server

Type the following command to start `jamst` and you will see the message below it.

```
> ./bin/jamst

 Listening for transport dt_socket at address: 4000
```

This shows that the debug server (`jamst`) listens for a socket connection request on the pre-defined port 4000 and will be suspended until the debug client connects.

[1] https://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html

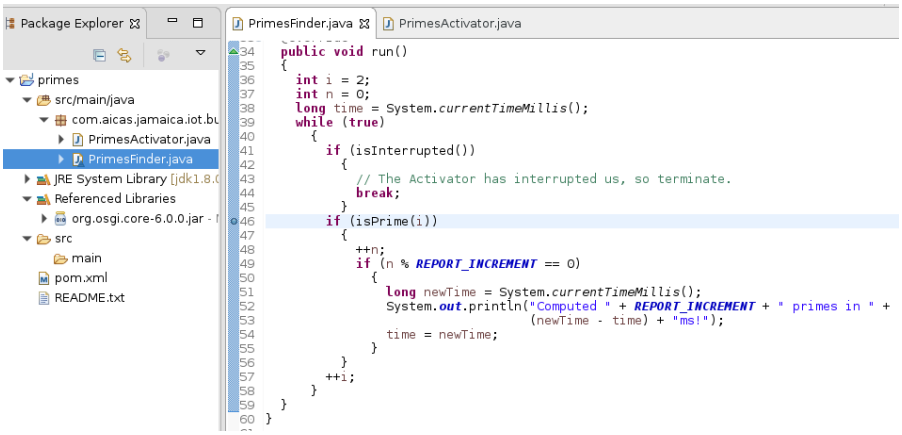

Figure 7.1: Setup the Java project for Computing Prime numbers

7.3.2 Start the Debug Client

In Eclipse, right click the Class `PrimesFinder.java` that contains a breakpoint and then select the **Debug as** option, followed by the **Debug Configurations...** option. In the [**Debug Configurations**] dialog on the **Connect** tab, choose the following options (depicted in Figure 7.2).

- ConnectionType: `Standard (Socket Attach)`
- Host: `localhost`
- Port: `4000`
- Allow termination of remote VM

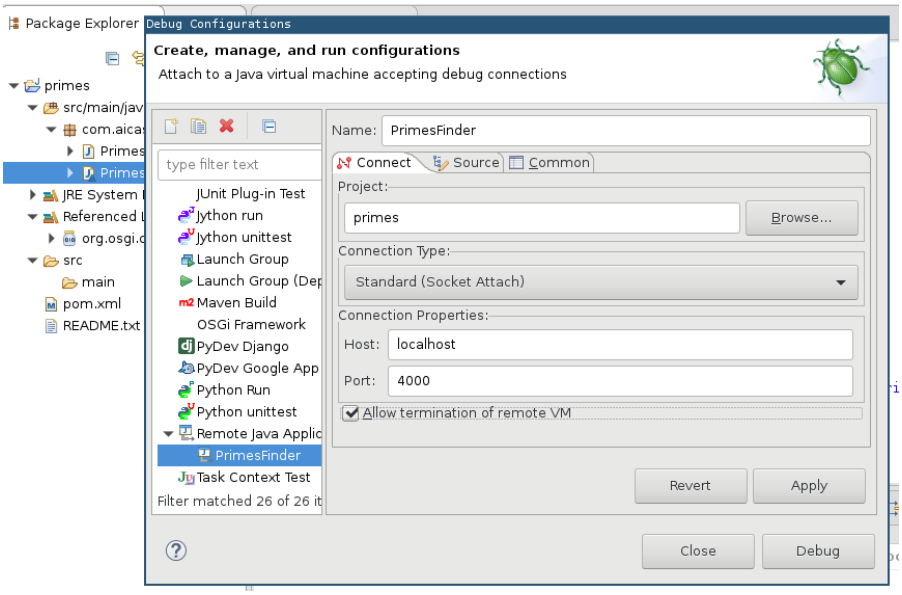

Figure 7.2: Debug Configurations for debugging with `jamst`

Click the [**Debug**] button and you can debug the “primes-example” bundle as usual.

Chapter 8

JamaicaAMS Runtime Reference

8.1 JamaicaAMS Properties

In the JamaicaAMS `setup/conf` directory there are three configuration files: `config.properties`, `system.properties` and `logging.properties`. The `config.properties` is typically used to configure the behavior of the framework and the bundles; the `system.properties` is purely a convenience, to avoid having to set complicated system properties via command line. JamaicaAMS only directly loads the configuration properties.

When JamaicaAMS starts, its launcher is activated and configures the system by passing the properties loaded from `system.properties` to the framework. Though it is allowed to specify duplicated properties in both configuration files, the configuration properties will override the system properties for duplicated cases.

Note that...

A complete description of the OSGi launching properties implemented by JamaicaAMS can be found in section 4.2.2 of the OSGi Core Release 8 specification [6].

8.1.1 Config Properties

This section presents the properties listed in the `config.properties` file.

- **org.osgi.framework.system.packages**

  To override the packages the framework exports by default from the classpath, this variable must be set.

- **org.osgi.framework.system.packages.extra**

  To append packages to the default set of exported system packages, this value must be set.

- **org.osgi.framework.bootdelegation**

  This property makes specified packages from the classpath available to all bundles and should be avoided. However, if such a configuration is to be made, an example of values listed could be *sun.*,com.sun.*,jdk.**.

- **jamaica-ams.bootdelegation.implicit**

  According to the boolean value of this property, JamaicaAMS tries to infer when to implicitly boot delegate to ease integration with external code. This feature is set to *true* by default.

- **jamaica-ams.readonly.cache**

  This property specifies the location of the read-only cache directory. The specified path must be absolute. There is no default value, if this property is not set, the read-only cache will not be used. Bundles placed in the read-only cache are assumed to be stable and immutable: they are not expected to be updated or uninstalled during runtime.

- **org.osgi.framework.storage**

  This property specifies the location of the bundle cache, which defaults to `jamaica-ams-cache` in the current working directory `${user.dir}`. If this value is not absolute, then the `${jamaica-ams.cache.rootdir}` will control how the absolute location is calculated as in

  org.osgi.framework.storage=${jamaica-ams.cache.rootdir}/jamaica-ams-cache.

- **jamaica-ams.cache.rootdir**

  This property is used to convert a relative bundle cache location into an absolute one, by specifying the root to prepend to the relative cache path. The default for this property is the current working directory `${user.dir}`.

- **org.osgi.framework.storage.clean**

  This property controls whether the bundle cache is flushed the first time the framework is initialized. Possible values are “none” and “onFirstInit”. The default value is “none”.

- **jamaica-ams.cache.locking**

  This boolean property is used to enable/disable bundle cache locking. On JVMs that do not support file channel locking, you may want to disable this feature. The default is enabled.

- **jamaica-ams.cache.filelimit**

  The integer value of this property limits how many open files the bundle cache is allowed to use. The default value is *0*, which is unlimited.

- **jamaica-ams.auto.deploy.action.1=install,start**

- **jamaica-ams.auto.deploy.action.2=install,start**

- **jamaica-ams.auto.deploy.action.3=install,start**

  The properties above determine which actions are performed when processing the auto-deploy directory. It is a comma-delimited list of the values “install”, “start”, “update”, and “uninstall”.

  An undefined or blank value is equivalent to disabling auto-deploy processing. The numerical ending component is the targeted start level. Any number of these properties may be specified for different start levels.

- **jamaica-ams.auto.deploy.dir.3**

  This property specifies the directory to use as the bundle auto-deploy directory; the default is `bundle.n` in the working directory. The numerical ending component is the target start level. Any number of such properties may be specified for different start levels.

- **jamaica-ams.auto.install.3**

  This property is a space-delimited list of bundle URLs to install when the framework starts. The numerical ending component is the target start level. Any number of such properties may be specified for different start levels.

- **jamaica-ams.auto.start.3**

  This property is a space-delimited list of bundle URLs to install and start when the framework starts. The ending numerical component is the target start level. Any number of such properties may be specified for different start levels.

- **jamaica-ams.log.uncaught.exceptions.bundle.threads**

  This property suppresses logging of uncaught exceptions in threads belonging to bundles. The default is *false*.

- **org.osgi.framework.startlevel.beginning=3**

  This property sets the initial start level of the framework upon startup.

- **jamaica-ams.startlevel.bundle=3**

  This property sets the start level of newly installed bundles.

- **jamaica-ams.service.urlhandlers**

  JamaicaAMS installs a stream and content handler factories by default. The value of this property should be actively set to *false*, in order not to install them.

- **jamaica-ams.shutdown.hook**

  By default, JamaicaAMS's launcher registers a shutdown hook to cleanly stop the framework. The value of this property should be actively set to *false*, in order to disable this hook.

- **jamaica-ams.shutdown.hook.timeout**

  This property sets the the maximum number of milliseconds the shutdown hook has to wait until the Framework has completely stopped. A value of *0* means the shutdown hook will wait indefinitely. The default is *3000*.

- **org.osgi.framework.security=osgi**

  This property enables security.

- **jamaica-ams.security.policy=./conf/osgi.all.policy**

  This property sets the path to the JamaicaAMS security policy files and should be adapted as needed.

- **jamaica-ams.log.level=2**

- **jamaica-ams.BundleLogger.level=2**

  These properties are used to set the log levels. The JamaicaAMS logging levels match those specified in the OSGi Log Service (i.e., 1 = error, 2 = warning, 3 = information, and 4 = debug).

- **java.util.logging.config.file=./conf/logging.properties**

  This property specifies the configuration file for Java Logging API.

- **org.apache.felix.log.maxSize=100**

  This property sets the maximum size of entries in the log history. A value of *-1* means the log has no maximum size; a value of *0* means that no historical information is maintained.

- **org.apache.felix.log.storeDebug=false**

  This property determines whether or not debug messages will be stored in the history.

8.1.2 System Properties

JamaicaAMS also has built-in system properties, listed in `<path to JamaicaAMS>/setup/conf/system.properties`, as seen below:

- **java.security.policy=./conf/all.policy**

  This property sets the path to the Java security policy files and should be adapted as needed.

8.1.3 Logger Properties

Note that...

The Log Services specification defines the methods and semantics of interfaces which bundle developers can use to log entries and to retrieve log entries. Bundles can use the Log Service to log information for the Operator. Other bundles, oriented toward management of the environment, can use the Log Reader Service to retrieve Log Entry objects that were recorded recently or to receive Log Entry objects as they are logged by other bundles.

Source: The OSGi Alliance—OSGi Compendium

In JamaicaAMS, Log Service is used to expose event messages. There are two JamaicaAMS loggers, with distinguished identifiers:

- **jamaica-ams.log**

- **jamaica-ams.BundleLogger**

They are used for the modules “framework” (i.e. the system bundle) and “bundle” (i.e., the bundles except the system bundle), respectively.

Generally, log levels can be set to control the amount of logging performed, where a higher number results in more logging. A log level of zero turns off the logging functionality completely.

The loggers are implemented using **java.util.logging**. It can be configured using the file specified in **java.util.logging.config.file**.

8.2 Budgets

Note that...

One distinguishing feature of JamaicaAMS in comparison to other OSGi frameworks is that resource budgets may be imposed on the separate bundles.

The possibility to define budgets is a very important aspect, since all bundles are executed within the same process and it becomes necessary to limit the resources used by each one of them. Otherwise, a single misbehaving bundle could impact the performance of the system as a whole.

In order to impose budgets on a bundle, the bundle's JAR file needs to contain an entry named **bundle.properties**. This entry needs to be in Java property file format (see [4]).

Currently JamaicaAMS supports CPU, memory, and thread budgets. Budgeting the CPU time ensures that all bundles get a fair share of execution time. Budgeting the number of threads and the memory usage ensures that no single bundle monopolizes these shared resources. In JamaicaAMS, budgets are implemented through the following properties:

- **budget.cpu** It defines the CPU budget as a percentage of each period of 100 milliseconds. Values for the **budget.cpu** property have the format *number%*, where number is an integer between *0* and *100*.

  If a bundle exceeds its budget in a given period, the priority of all threads started by the bundle is lowered to 0 for the remainder of the period. Priority 0 is a special priority that is lower than all Java priorities. At the beginning of the next period, the original priorities are restored for all threads of the bundle.

  The effect of a priority being dropped to the lowest possible value is that the bundle's threads will only be scheduled to run if no other thread in the system is ready to be executed.

  Example:

  In the JamaicaAMS subdirectory `example`, the bundles "primes" and "primes-with-budgets" display a sequence of prime numbers. For the variant "primes-with-budgets" however, it is specified a cpu budget that the bundle is allowed to use. This is declared under `bundle.properties`.

  ```
  budget.cpu=5%
  ```

  When running individually, each bundle will use full CPU time. However, CPU time will be significantly divided when running different bundle combinations. For example,

  - When running both the "primes" and "primes-with-budget" bundles, "primes" calculates faster than "primes-with-budget" because it has no CPU budget, i.e. it uses full CPU time by default.
  - When running "primes" and "primes-lower-priority" at the same time, "primes" calculates prime numbers faster than "primes-lower-priority" because its thread priority is higher than "primes-lower-priority".

  - When running bundles “primes-with-budget” and “primes-lower-priority” at the same time, “primes-lower-priority” calculates primes faster than “primes-with-budget” because when the CPU budget is exceeded, the priority of “primes-with-budget” is reduced to the lowest possible value and is lower than the priority of “primes-lower-priority”.

  By running both bundles, the user can perceive the impact of imposing such restrictions.

- **budget.memory** JamaicaAMS allocates a fixed-sized Java heap at startup (typically between *64 MB* and *256 MB*). All memory allocations take place in this heap, that is shared by all bundles.

  Memory budgets ensure that no single bundle uses too much of the shared heap. Otherwise, starting additional bundles might not be possible, or else already running bundles may cease to operate properly due to an out-of-memory situation.

  JamaicaAMS permits memory budgets to be defined in bytes. Values for the **budget.memory** property have the format *number*, where number is a non-negative integer denoting the memory budget in bytes.

  If a bundle exceeds its memory budget, any subsequent allocation via “`new`” will result in an `OutOfMemoryError`.

  Example: In the JamaicaAMS subdirectory `example`, the bundles “memory-consumption-example” and “memory-consumption-budget-example” create 10 arrays of bytes respectively and each array consumes *1 MB* of Java heap. The size of free memory is printed after every successful array allocation. In the “memory-consumption-budget-example”, the memory budget specified in the `bundle.properties` permits the bundle to use up to *5 MB* of Java heap.

```
budget.memory = 5242880
```

  Running the “memory-consumption-example”, all 10 arrays should be allocated successfully. While running the “memory-consumption-budget-example”, the memory budget will be exceeded by attempting to allocate the fifth array and an `OutOfMemoryError` will be thrown.

```
Start eating the memory.
Amount of free memory : 38946080
Amount of free memory : 37896000
Amount of free memory : 36846336
Amount of free memory : 35796672
[Mon Feb 19 13:59:37 CET 2024] ERROR: Bundle com.aicas.jamaica.
ams.bundle.example.memory-consumption-budget-example [15]
ThreadGroup is limited to 5242880 bytes memory, used 4203136
bytes! The current allocation exceeds the budget limit and
cannot be allowed.
java.lang.OutOfMemoryError
```

- **budget.threads** It defines the thread budget as number of concurrently active threads. The format of this property is *budget.threads* = <*n*>.

  A value larger than *0* will enforce the number of active threads to be limited by this value. Any value smaller than *0* is invalid.

  If a bundle reaches its thread budget, any subsequent creation of a new thread will result in an exception.

  Example: In the JamaicaAMS subdirectory `example`, the bundles "thread-spawning-example" and "thread-spawning-budget-example" each start 15 threads and keep them active until interrupted. The threads budget is set to 10 in the "thread-spawning-budget-example", which allows the bundle to have at most 10 active threads.

  ```
  budget.threads = 10
  ```

  The "thread-spawning-example" should run successfully. However an error will occur when running the "thread-spawning-budget-example".

  ```
  javax.realtime.enforce.ThreadLimitError: ThreadGroup limit
  exceeded: 10
  ```

  A bundle may create an arbitrary number of instances of class `java.lang.Thread`, and there is no limit on the number of calls to `start()` on these threads. However, the number of threads that may be alive simultaneously is limited by this property. To be sure that a thread is no longer alive, a call to `Thread.join()` is required. Before a call to `Thread.join()`, the thread may have finished its Java execution, but it may still not have been released back to the framework's thread pool.

8.3 Thread Count

This section accounts for the threads that are created by JamaicaAMS. This information is useful for configuring the overall thread number and for configuring thread numbers for individual bundles.

In JamaicaAMS, the overall thread number can be configured through the environment variable **JAMAICA_AMS_NUM_THREADS**: this sets the maximum allowed threads in the runtime. Bundle thread numbers can be configured through the property **budget.threads**. There are in total 42 threads when JamaicaAMS runs with the default bundles.

Below is a list of JamaicaAMS threads. Note that starter threads are created when JamaicaAMS starts and do not terminate until it shuts down. To complete the information, this section also includes a list of the Jamaica Virtual Machine main threads.

JamaicaAMS Starter Threads

- Main Thread

  To invoke the construction and the initialization of JamaicaAMS and to start the initialized framework.

JamaicaAMS Framework Threads

- FrameworkWiring

  To perform asynchronous package refreshes.
- StartLevel

  To query and modify the start level information for the framework.
- FrameworkDispatchQueue

  To update all listeners (e.g., Framework, bundle, synchronous, and service listeners) associated with a specified bundle, and to dispatch the events of the bundle to the event listeners.
- FrameworkExecutorThread

  To perform special framework operations (e.g. bundle termination) asynchronously in an isolated and safe manner.

Administrative JamaicaVM Threads

- Finalizer
- MemReservation0 (idle time GC)
- Reference Handler
- PosixEventThread (heap)
- SignalPumpThread

JamaicaVM Realtime Threads

- AbstractAsyncEventHandlerThread

8.3.1 Bundle Threads

Those contain all user threads that are created by the bundle. Note that user threads often have a limited lifetime. The configurable thread numbers (**JAMAICA_AMS_NUM_THREADS** and **budget.threads**) impose bounds on the number of threads that exist *simultaneously*.

In addition to the user threads, there is an administrative thread for each bundle, which only exists while the bundle is in active state. This thread is associated to the installed bundle. There is a `BundleThreadGroup` per Bundle for managing its threads. There are in total 31 threads constructed for the default built-in bundles.

JamaicaAMS Bundle Threads

- **com.aicas.jamaica.ams.bundle.configuration-admin**
  Provides a service which allows for easy management of configuration data for configurable components. Threads **CM Event Dispatcher** and **CM Configuration Update** are part of this package.
- **com.aicas.jamaica.ams.bundle.osgi-log-writer**
  Uses the **java.util.logging** to output the log entries recorded by the logging service. The **java.util.logging** runs on a seperated thread.
- **com.aicas.jamaica.ams.bundle.policy-file-reader**
  Loads the policy file for the OSGi security layer.
- **jakarta.json**
  Provides an implementation of Jakarta JSON processing.
- **org.apache.felix.cm.json**
  Provides support for OSGi configurations specified in JSON documents.
- **org.apache.felix.configurator**
  Provides an implementation of the OSGi Configurator Service Specification.
- **org.apache.felix.gogo.command**
  Provides a set of basic commands.
- **org.apache.felix.gogo.runtime**
  Provides the core command processing functionality.
- **org.apache.felix.gogo.shell**
  Provides a simple textual user interface to interact with the command processor. Internally it starts the shell on a separate thread, **Gogo Shell**. The Gogo Shell thread continues to create threads **job controller**, **pipe-gosh –login**, and a thread pool, which incrementally constructs 10 threads. Therefore, there are in total 13 threads constructed directly and indirectly from the **org.apache.felix.gogo.shell** thread.
- **org.apache.felix.log**
  Provides a set of log utilities. Thread **FelixLogListener** is part of this package.
- **org.apache.felix.scr**
  Provides an implementation of the declarative services specification. Thread **SCR Component Actor** is part of this package.
- **org.osgi.service.component**
  API of the declarative services specification.
- **org.osgi.util.converter**
  Provides a set of basic utilities.
- **org.osgi.util.function**
  Provides a set of basic utilities.
- **org.osgi.util.promise**
  Provides a set of basic utilities.

8.4 Usage of the Java Native Interface (JNI)

In general developers are discouraged from using the Java Native Interface (JNI). Nevertheless, the usage of JNI allows them to use native code and easily interact with Java objects (e.g. get and set field values, and invoke methods without many constraints).

On the one hand, with JNI it is possible to use the safety behavior of the Java language for the ability to accomplish the tasks listed earlier; on the other hand, JNI provides powerful low-level access to the machine resources (memory, I/O, and so on), but you need to be careful because you are working without the safety net usually provided to Java developers.

JNI's flexibility and power introduce the risk of programming practices that can lead to poor performance, bugs, consume system heap and fail on a heap allocation.

For these reasons, developers must be careful about the code they include in their software and use good practices to safeguard its integrity. Please note the additional information on best practices:

- A common JNI programming error is to call a JNI method and to proceed without checking for exceptions once the call is complete. This can lead to buggy code and crashes. For this reason it is important to return to the Java side in case of a pending exception. The pending exception will then be raised as a regular Java exception.
- The memory allocated on the native side of JNI is not accounted for in the memory limits.
- Many JNI methods have a return value that indicates whether the call succeeded or not. A common error, similar to not checking for exceptions, is to fail to check the return value and for the code to proceed on the assumption that the call was successful.
- Native methods can create global references so that objects are not garbage collected until they are no longer needed. Common errors are forgetting to delete global references that have been created or losing track of them completely. Not freeing global references is an issue not only because they keep the object itself alive but also because they keep all objects that can be reached through the object alive. In some cases this can add up to a significant memory leak.

Note: In its Core document for the Release 8, the OSGi Alliance advises developers about the use of the Java Native Interface.

Considerations Using Native Libraries

There are some restrictions on loading native libraries due to the nature of class loaders. In order to preserve namespace separation in class loaders, only one class loader can load a native library as specified by an absolute path. Loading of a native library file by multiple class loaders (from multiple bundles, for example) will result in a linkage error.

Care should be taken to use multiple libraries with the same file name but in a different directory in the JAR. For example, foo/http.dll and bar/http.dll. The Framework must only use the first library and ignore later defined libraries with the same name. In the example, only foo/http.dll will be visible.

A native library is unloaded only when the class loader that loaded it has been garbage collected. When a bundle is uninstalled or updated, any native libraries loaded by the bundle remain in memory until the bundle's class loader is garbage collected. The garbage collection will not happen until all references to objects in the bundle have been garbage collected, and all bundles importing packages from the updated or uninstalled bundle are refreshed. This implies that native libraries loaded from the system class loader always remain in memory because the system class loader is never garbage collected.

It is not uncommon that native code libraries have dependencies on other native code libraries. This specification does not support these dependencies, it is assumed that native libraries delivered in bundles should not rely on other native libraries.

Source: `https://docs.osgi.org/download/r8/osgi.core-8.0.0.pdf`.

8.5 Usage of the two-level cache strategy

JamaicaAMS supports a two-level cache strategy, which extends the default cache mechanism by introducing an additional read-only cache. This approach is particularly useful in deployment scenarios where some bundles are stable and unlikely to change across framework restarts.

In many cases, part of the bundles in the framework remain unchanged for long periods. However, maintaining a clean framework environment often requires deleting the entire cache directory, which also removes the stable bundles unnecessarily. By leveraging a two-level cache strategy, stable bundles can be stored in a read-only cache, while dynamic or frequently updated bundles continue to reside in the normal writable cache. JamaicaAMS loads bundles first from the read-only cache, and then the normal cache. This separation provides the following benefits:

- **Improved startup performance**: The state of bundles in the read-only cache is preserved and does not need to be recomputed.
- **Easier cache management**: The normal cache directory can be safely cleared without affecting the read-only bundles.
- **Enhanced reliability**: Stable bundles in the read-only cache are protected from accidental modification or deletion.

To enable this feature, follow the steps below:

1. Preparing the read-only cache
   - Start JamaicaAMS normally.
   - Install all the bundles that are planned to be kept in the read-only cache.
   - Stop JamaicaAMS. This will generate the standard cache directory, which defaults to `jamaica-ams-cache` in the current working directory, containing the framework state and cache data.
   - Move or rename this cache directory to a suitable location and set it to read-only to prevent further modifications.
2. Delete or clear the auto-deploy directories to avoid duplicate bundle installation errors, since these directories contain the bundles that have already been installed and kept in the read-only cache. By default, these directories are named `bundle.n` and are located in the working directory.
3. Enable the read-only cache in JamaicaAMS, you can choose either of the methods:
   - Set the configuration property in the configuration files `config.properties`

     ```
     jamaica-ams.readonly.cache = <absolute path to read-only cache>
     ```

   - Alternatively, you can use the command line option **-r <bundle-readonly-cache-dir>**.

When security is enabled, the bundles stored in the read-only cache are assumed to have been verified at the time of cache creation and remain unchanged afterward. Therefore, these bundles are considered trusted and can be exempted from further verification at runtime. By using the two-level cache strategy, you can grant full permissions to these trusted bundles by adding the following lines to the `osgi.global.policy`

```
ALLOW {
 [org.osgi.service.condpermadmin.BundleLocationCondition "preinstalled:*"]
 (java.security.AllPermission)
} "all_allow_location_bundle.preinstalled"
```

Note that...

1. Bundles placed in the read-only cache are assumed to be stable and immutable. They are not expected to be updated or uninstalled at runtime. Any attempt to do so will be ignored and may result in warnings.
2. If you intend to store accelerated bundles in the read-only cache, and reuse their extracted libraries,
   - Ensure that compiled code loading is enabled at the time the read-only cache is prepared.
   - The extraction directory must remain consistent across runs.

   Otherwise, the framework will re-extract the bundle on every startup, negating the benefit of the read-only cache.

Chapter 9

Information for Specific Targets

Generally all JamaicaVM target systems should be also feasible to run JamaicaAMS, and this chapter aims at documenting information referring to specific platforms.

9.1 Linux

JamaicaAMS can be run on variant Linux distributions for variant target, e.g., Intel x86_64 architecture, ARM architecture (v7 and v8), and RISC-V. The following section describes several configuration issues which might need to be taken care of to run the JamaicaAMS.

9.1.1 Shared Libraries

Different libraries are required on different targets. In order to run JamaicaAMS, at least the following shared libraries should exist in the library search paths.

- `libm.so`
- `libpthread.so`
- `libdl.so`
- `librt.so`
- `libstdc++.so`

Some Board Support Packages (BSPs) might not have `libstdc++.so` integrated. In this case, the `libstdc++.so` has to be copied to the target file system where the library loader can discover it. For example, if the `libstdc++.so` is copied in a folder `<path to JamaicaAMS>/setup/lib`, the environment variable `LD_LIBRARY_PATH` should be set with following command:

```
> export LD_LIBRARY_PATH=<path to JamaicaAMS>/setup/lib:$LD_LIBRARY_PATH
```

On VxWorks the dynamic library `libc.so` must be renamed to `libc.so.1` and added to the `bin` directory containing the executable binaries. This library is located inside the VxWorks Source Build (VSB) project. When using elliptic curve cryptography the following additional libraries are needed: `libcplusplus.so.1`, `libllvmcplus.so.1` and `libllvm.so.1`. They can also be found inside the VSB project. For more information, see the Shared Library Location and Loading at Run-time paragraph in the VxWorks Application Programmer's Guide.

9.1.2 Random Number Generator

JamaicaAMS requires a random number generator been initialized on the target system. If the BSP does not initialize the random number generator at boot time, it has to be done prior to starting the JamaicaAMS. For example, on a Raspberry Pi, the application processor contains a Hardware Random Number Generator. The BSP has been configured with the device (`/dev/random` and `/dev/hwrng`) existing in the system. Therefore, no additional steps need to be done. For other distribution or targets, please consult with the document of the specific distributions or the targets.

Bibliography

[1] Apache Felix Gogo Shell. https://felix.apache.org/documentation/subprojects/apache-felix-gogo.html.

[2] Introducing the Bndtools. https://bnd.bndtools.org.

[3] JamaicaVM 8.10 User Manual. https://www.aicas.com/download/manuals/aicas-JamaicaVM-8.11-Manual.pdf.

[4] Java Properties File Format. https://www.aicas.com/jamaica/8.11/doc/jamaica_api/java/util/Properties.html#load-java.io.Reader.

[5] JDK 8 PKCS#11 Reference Guide. https://docs.oracle.com/javase/8/docs/technotes/guides/security/p11guide.html.

[6] OSGi Core Release 8 Specification. https://docs.osgi.org/download/r8/osgi.core-8.0.0.pdf.

[7] PMD Information. https://pmd.github.io/.

[8] RTSJ Standards. https://www.aicas.com/wp/standards/rtsj/.

[9] SpotBugs Information. https://spotbugs.github.io/.

[10] The Eclipse Download page. https://www.eclipse.org/downloads/.

[11] The Eclipse project. https://www.eclipse.org/.

[12] The Plugin-in Development Environment (PDE) for Eclipse. https://www.eclipse.org/pde/.

[13] Wikipedia site "Computer Security". https://en.wikipedia.org/w/index.php?title=Computer_security&oldid=1171684823.

[14] Morrie Gasser. *Building a Secure Computer System*. Van Nostrand Reinhold Co., New York, NY, USA, 1988.

[15] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java Virtual Machine Specification, Java SE 8 Edition*. Pearson Education, 2014.