

JAVA ANTIDECOMPILER API README

Source code security is an important part of security in general. Indeed the many attacks are directed to the source code. On the other hand, the source code is the only valued thing you have after a few years of hard work.

It's also clear that all your support, marketing, and customer relations will be rendered useless if your code falls into the hands of a competitor or is stolen by an attacker.

Java developers are familiar with the scenario where a sold jar is recompiled and, with minor changes to circumvent patent or intellectual property protection, appears on the markets and takes away a significant part of the profit.

There are many source code protection tools for Java and Java-based languages. Among them, the most modern and reliable means is byte code level protection. This means that during the protection process, special transformations are carried out with the byte code, after which the original functionality of the recompiled Java app can't be restored, just as text encrypted, for example, using AES-256 cannot be decrypted, even if there is the source of AES-256 algorithm.

We use the term API in the Apache sense, i.e. as a non-specialized library jar [+ dependencies] that can be used in various projects. Therefore it can be sold independently.

Java Antidecompiler API was successfully tested on the most popular Java APIs such as Log5j, JUnit, commons-email, commons-cli, commons-net, commons-io, commons-lang3, commons-text, BCEL, ASM, and Jakarta.mail.

As for increasing the size of the protected API, then

- First, free cheese only in a mousetrap, and
- Secondly, you can consider the additional size as the thickness of the protective wall for your

API or as third-party software, the size of which is usually not taken into account even if only a small part is used.

Good luck,
BIS Guard Team

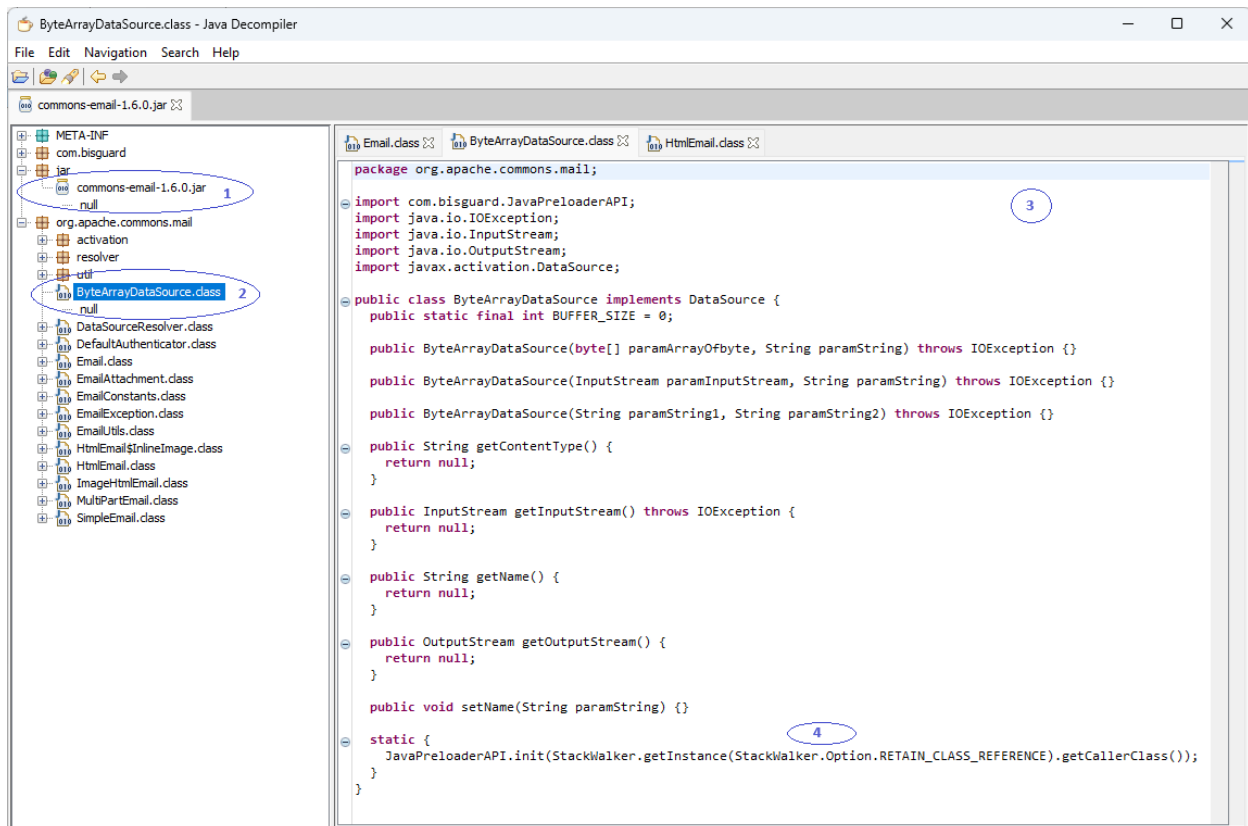


The dependencies screen is self-explained. We add them for the first time only.



Brief How It Works

We decompile all classes selected for protection from the source API jar and extract the headers only from them. That is, the developer can see only field names without values, method headers without bodies, and so on something like the following



where

- 1 – encrypted API jar invisible for SystemClassLoader and decompilers
- 2 – class after extraction headers for development time
- 3 – extracted class text
- 4 – callback method for switching ClassLoader's

We differentiate between design time, where the user only sees skeletons of the classes, and runtime, where the custom ClassLoader decipheres the API when running the actual classes.

EXAMPLE. API commons-email-1.6.0.jar

```
package user;

// from https://commons.apache.org/proper/commons-email/userguide.html

import org.apache.commons.mail.DefaultAuthenticator;
import org.apache.commons.mail.Email;
import org.apache.commons.mail.EmailException;
import org.apache.commons.mail.SimpleEmail;

public class User_EMAIL {

    public static Object[] values;

    public static void main(String[] args) throws EmailException {
        values = args;

        if (values.length < 3) throw new EmailException("Wrong argument number");

        Email email = new SimpleEmail();
        email.setHostName(args[0]);
        email.setSmtpPort(465);
        email.setAuthenticator(new DefaultAuthenticator(args[1], args[2]));
        email.setSSLonConnect(true);
        try {
            email.setFrom("taas@bisguard.com");
            System.out.println("setFrom: success");
        } catch (EmailException e) {
            e.printStackTrace();
        }
        email.setSubject("Test Mail");
        try {
            email.setMsg("This is a test mail ... :-)");
            System.out.println("setMsg: success");
        } catch (EmailException e) {
            e.printStackTrace();
        }
        try {
            email.addTo("sales@bisguard.com");
            System.out.println("addTo: success");
        } catch (EmailException e) {
            e.printStackTrace();
        }
        try {
            email.send();
            System.out.println("send: success");
        } catch (EmailException e) {
            e.printStackTrace();
        }
    }
}
```

The variable `values` are needed iff we want to pass parameters through a command line or from another class.

Command line class

```
package user;

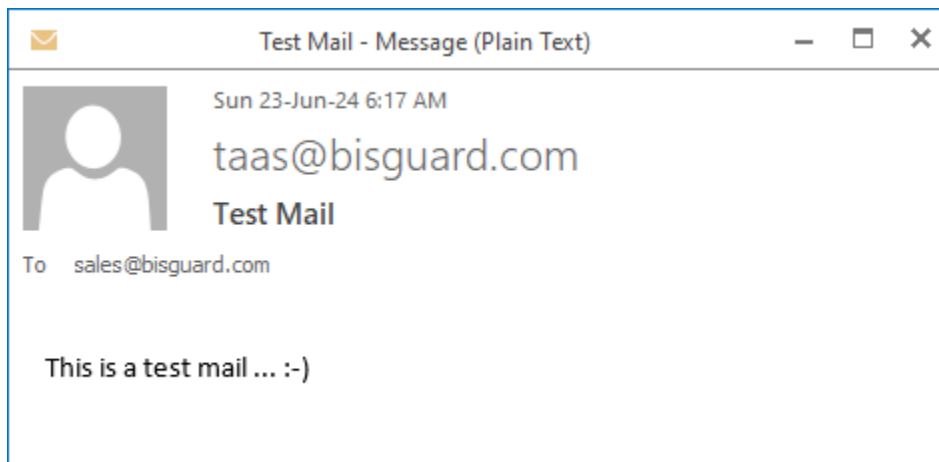
import org.apache.commons.mail.EmailException;

public class User_EMAIL_CMD {

    public static void main(String[] args) {
        String[] values = new String[3];
        values[0] = hostname, e.g. "smtp.hostname.com";
        values[1] = user;
        values[2] = password;
        try {
            User_EMAIL.main(values);
        } catch (EmailException e) {
            e.printStackTrace();
        }
    }
}
```

Output:

```
Jun 23, 2024 4:37:57 PM com.bisguard.JavaPreloaderAPI init
INFO: com.bisguard.JavaPreloaderAPI TRIAL VERSION STARTED
Jun 23, 2024 4:37:57 PM com.bisguard.JavaPreloaderAPI init
INFO: user.User_EMAIL STARTED
setFrom: success
setMsg: success
addTo: success
send: success
Jun 23, 2024 4:38:06 PM com.bisguard.JavaPreloaderAPI init
INFO: user.User_EMAIL FINISHED
Jun 23, 2024 4:38:06 PM com.bisguard.JavaPreloaderAPI init
INFO: com.bisguard.JavaPreloaderAPI TRIAL VERSION FINISHED
```



THE END