OSX.XLoader hides little except its main purpose: What we learned in the installation process

blog.malwarebytes.com/mac/2021/07/osx-xloader-hides-little-except-its-main-purpose-what-we-learned-in-the-installation-process/

Thomas Reed

July 26, 2021

Last week, Check Point Research described a new Mac variant of malware they call XLoader. It was identified as being the successor of something called Formbook, a very prevalent threat in the Windows world. According to Check Point, the Mac version of the malware is being “rented” as part of a malware-as-a-service program, at the price of $49 for one month or $99 for three months.

Unfortunately, Check Point was a bit vague on the details of how the Mac version behaves, leaving folks unsure of exactly how to protect themselves against this malware. Fortunately, more details have since come to light.

How XLoader gets installed

XLoader appears to be distributed within a .jar – or Java archive – file. Such a file contains code that can be executed by Java, dropping the malware on the system. One major advantage, for the attacker, of using Java is that the “dropper” (the file responsible for installing the malware) can be cross-platform.

However, this file format has a very significant disadvantage for the attacker, which is that macOS does not, by default, include Java, and has not for quite some time. Back around 2011 to 2012, there was a flood of multiple different pieces of malware designed to infect Macs via vulnerabilities in Java, which at the time was installed on every Mac out of the box. This meant that all Macs were vulnerable, and to make matters worse, despite updates from Oracle (Java's owner), more vulnerabilities kept being found and exploited.

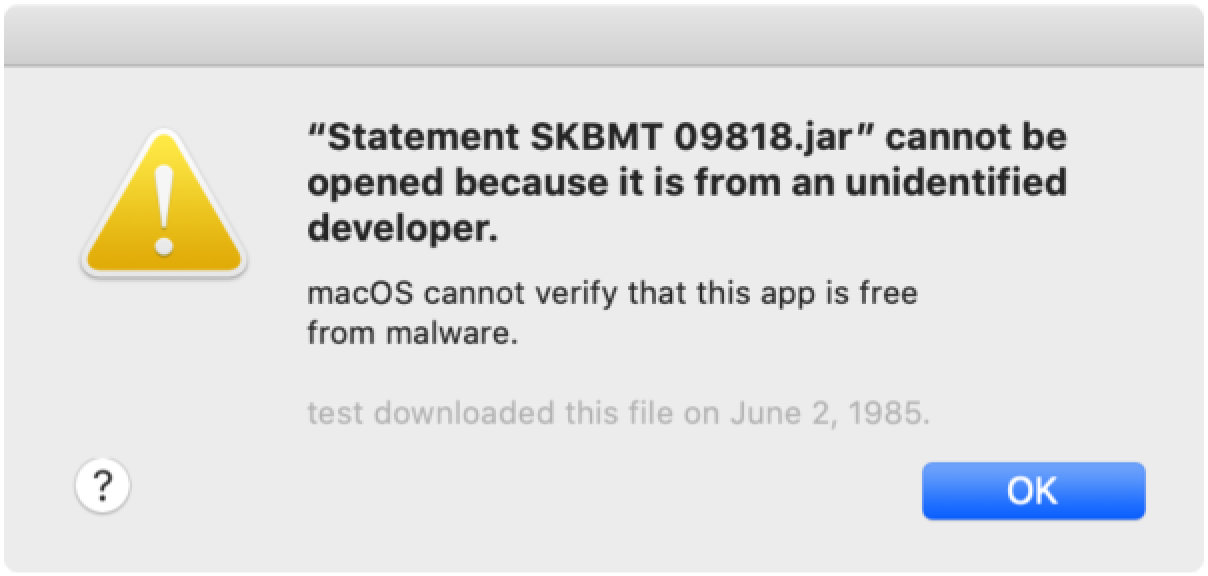
Apple responded by ripping Java out of the system. Since then, the only way Java can be on a system is if the user has installed it, which most users won't. This means that Java is no longer a very useful means of attack on modern macOS systems.

There can be a couple reasons why a JAR file might be used on macOS. One is unfamiliarity with modern macOS, from a malware developer who has Java on their system but doesn't understand this is non-standard for some reason. This is something often seen with more amateurish malware, and there are definitely some indications of that with this malware.

However, another reason is that the malware is targeted at specific individuals who are known to have Java installed. These could be Java developers, for example, at a particular company, or perhaps employees at a company that uses Java-based tools. A source at ESET reported that they had detected this malware back in January, with the JAR file being distributed via email. This points to a targeted campaign.

The installation process

The dropper – named Statement SKBMT 09818.jar in this case – would need to be opened by the user. The good news is that, if it was downloaded from an email client or browser that uses modern file system code, it will be marked with a "quarantine" flag. This means that the Gatekeeper feature of macOS will not allow it to execute by default.

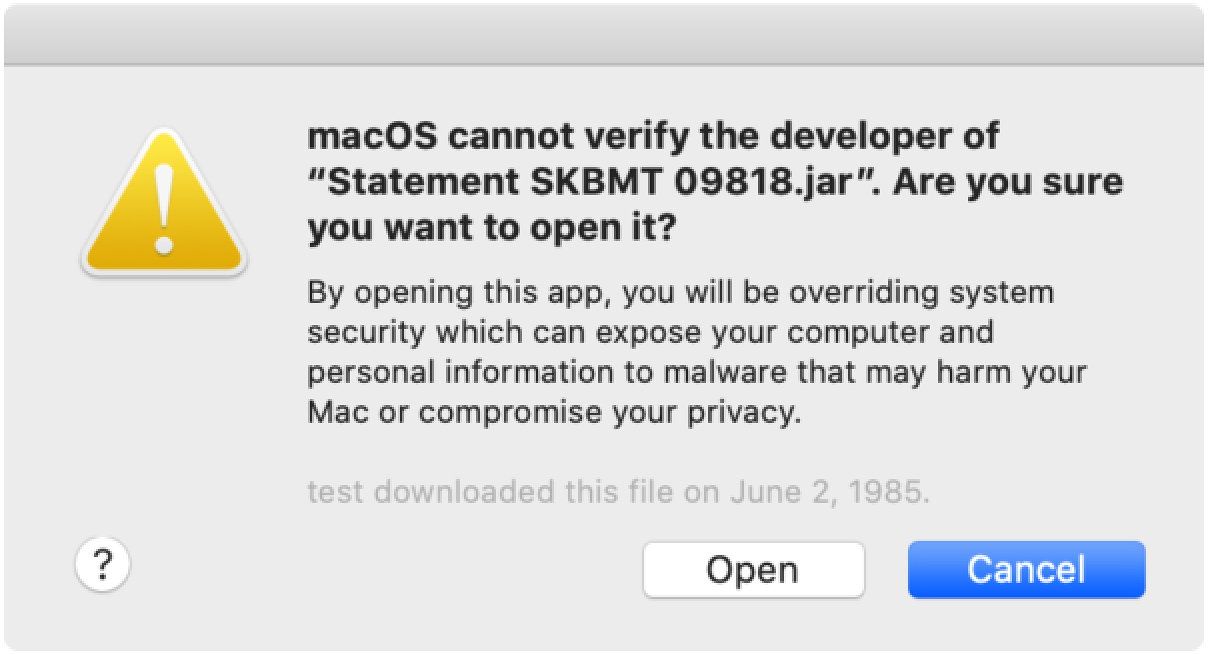


"Statement SKBMT 09818.jar" cannot be opened because it is from an unidentified developer.

macOS cannot verify that this app is free from malware.

test downloaded this file on June 2, 1985.

There are ways that Mac users can bypass this and open the file anyway, but not without seeing a similar warning first.


macOS cannot verify the developer of "Statement SKBMT 09818.jar". Are you sure you want to open it?

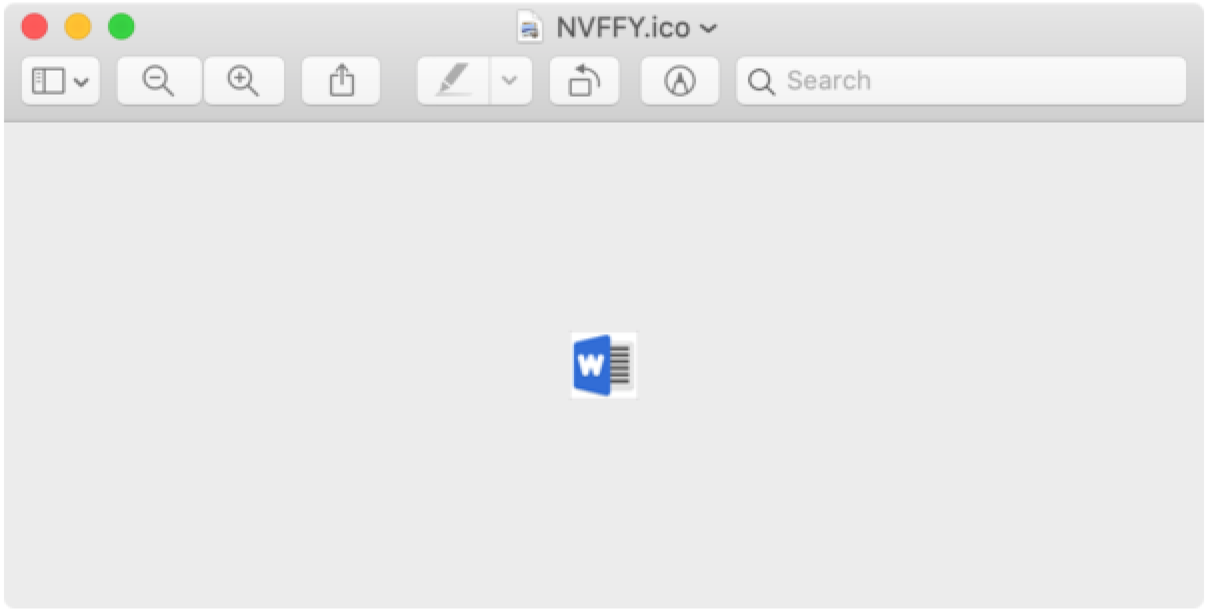
By opening this app, you will be overriding system security which can expose your computer and personal information to malware that may harm your Mac or compromise your privacy.

test downloaded this file on June 2, 1985.

Still, a significant amount of Mac malware droppers in the last year or so have been unsigned, and have given users instructions on what to expect and how to open the file. In such cases, users can and do bypass these warnings and open the malicious installers successfully.

In the event that the user downloads the JAR file using an email client that does not use the right file system code, and thus does not set a quarantine flag, the file will immediately open when double-clicked, without any complaints. The same will also be true if the file is copied onto a non-Mac drive before being opened, such as a Windows network share, where the quarantine flag will be lost.

Once opened, the JAR file will infect the system, and strangely, will also open a .ico (icon) file containing a Microsoft word icon image.



It’s unknown why this is done. It’s not uncommon for malware to open a “decoy document.” In such cases, when the malware pretends to be a document (as in this case, where the malware is pretending to be a statement of some kind), it will then open a document for the user to look at, to assuage suspicions the user would have if no document ever opened, while it’s doing bad stuff behind the scenes.

Is this a really badly botched attempt to open a decoy document? Or is it possible that this wasn’t meant as a public release, and the file being opened is a placeholder? Either would be a reasonable explanation, but we don’t know which is true.

While the user is looking in confusion at this wonderful icon, the JAR code will install the malware in the background. On my test machine, the malware installed the following items:

```
~/._p1pxXl0Fz4/I8ppUnip.app
~/kIbwf02l
~/NVFFY.ico
~/LaunchAgents/com._p1pxXl0Fz4.I8ppUnip.plist
```

The launch agent .plist file is used to load the app from the hidden folder (`._p1pxXl0Fz4`) found in the user folder. The `kIbwf02l` file is an exact copy of the Mac mach-o executable file found inside the app, but it’s unclear why this is left there, as it isn’t actually used. It’s a suspiciously-named file that will be visible to the user and thus may raise suspicions, so its presence is odd.

The `NVFFY.ico` file is the Microsoft Word icon file opened by the malware as a “decoy.”

A closer look at the Java code

Extracting the Java code from the JAR file was a painless task, and the code is not obfuscated in any way. The code is quite simple, but is able to drop a payload on either Windows or Mac. If you’re not interested in looking at code, feel free to skip ahead.

The filenames are hard-coded in the JAR file, as seen here.

```
private static String get_crypted_filename(final int pt) {
    final String exe_ = "fI4sWHkeeeee";
    final String mach_o = "kIbwf02ldddd";
    final String display = "NVFFYfffffff";
```

It’s a bit of a stretch to call these filenames “encrypted,” as the only thing that has been done to them is that a specific letter has been added to the end, repeating a varying number of times. (What letter is used depends on the string in question, and is also hard-coded.) These characters are stripped off to get the filenames, resulting in the mach-o filename of `kIbwf02l` and the “display” document filename of `NVFFY` .

The malware has quite simple code for determining the system it’s running on:

```
public static int _GetOS() {
    final String OS = System.getProperty("os.name").toLowerCase();
    if (OS.contains("mac")) {
        return 1;
    }
    if (OS.contains("win")) {
        return 2;
    }
    return 0;
}
```

From there, the malware reads encrypted data from within the JAR file and writes it out to the desired location on the system (in this case, the `kIbwf02l` file).

```
private byte[] getFileFromResource(final String name) throws Exception {
    try (final InputStream in = this.getClass().getResourceAsStream("/resources/" +
name)) {
        final byte[] data = new byte[16384];
        final ByteArrayOutputStream buffer = new ByteArrayOutputStream();
        int nRead;
        while ((nRead = in.read(data, 0, data.length)) != -1) {
            buffer.write(data, 0, nRead);
        }
        return buffer.toByteArray();
    }
}
```

From there, the malware launches the malicious process and opens the decoy document (aka “displayFile”).

```
        if (osFile != null && osFile.length != 0) {
            final String absolutePath = userPath + osFilename + ((os == 1) ? "" :
".exe");
            stubClass.writeBufferToFile(decrpt_data(osFile), absolutePath);
            if (os == 1) {
                final File file = new File(absolutePath);
                final Set<PosixFilePermission> perms = new
HashSet<PosixFilePermission>();
                perms.add(PosixFilePermission.OWNER_READ);
                perms.add(PosixFilePermission.OWNER_WRITE);
                perms.add(PosixFilePermission.OWNER_EXECUTE);
                Files.setPosixFilePermissions(file.toPath(), perms);
            }
            processBuilder.command(absolutePath);
            processBuilder.start();
        }
        final byte[] displayFile = stubClass.getFileFromResource(displayFilename);
        if (displayFile != null && displayFile.length != 0) {
            final String absolutePath2 = userPath + displayFilename +
getDisplayExt();
            stubClass.writeBufferToFile(decrpt_data(displayFile), absolutePath2);
            final File f = new File(absolutePath2);
            Desktop.getDesktop().open(f);
        }
```

The malicious application

The malicious Mac application, dropped and executed by the JAR file, is heavily obfuscated, making it hard to learn more about what it does. According to analysis done by SentinelOne, one of the app's main goals appears to be harvesting credentials.

The app itself is not code signed in any way. However, since it was created by the JAR and not downloaded from anywhere, it can be executed without Gatekeeper examining it or asking for user consent to run it. The launch agent .plist file is used to ensure the app is launched at startup, but explicitly does not try to keep the process alive. This means that if anything terminates the malicious process, it will not re-open until the next reboot.

The app itself has been marked as an LSUIElement, which is done to prevent its icon from showing on the Dock whenever it is running. This is a feature intended to be used by apps responsible for managing some user interface element – such as a menu bar icon – but that do not have a user interface of their own and thus can't be interacted with directly. This prevents the Dock from being littered with these kinds of apps, but is a common technique used to prevent malicious apps from appearing in the Dock.

TL;DR

To sum up, this malware is likely to be used for targeted attacks against intended victims who are known to have Java installed. Attackers may also have knowledge that something in the victims' environment will enable users to easily open a JAR file without being blocked by Gatekeeper.

The dropper itself is completely unsophisticated, with barely an attempt to hide anything, while the mach-o executable used in the malicious application installed on the system is quite well protected against prying eyes. This may be an indication that the two components of the malware were developed by different individuals.

This malware will be detected by Malwarebytes for Mac as OSX.XLoader. However, as of yet, data shows that Malwarebytes has not detected a single instance of this malware in the wild.