

# Druva inSync for Mac Local Privilege Escalation

## tldr

This research details four CVEs in [Druva inSync](#) that allow local privilege escalation to the root user on MacOS. These issues have been [patched by the Druva team](#). If you're using the product, now would be a good time to check if you're up to date. Otherwise, skip down to the videos to watch it in action, or keep reading for all the details.

[CVE-2021-36665](#): An issue was discovered in Druva 6.9.0 for MacOS, allows attackers to gain escalated local privileges via the inSyncUpgradeDaemon.

[CVE-2021-36666](#): An issue was discovered in Druva 6.9.0 for MacOS, allows attackers to gain escalated local privileges via the inSyncDecommission.

[CVE-2021-36667](#): Command injection vulnerability in Driva inSync 6.9.0 for MacOS, allows attackers to execute arbitrary commands via crafted payload to the local HTTP server due to unsanitized call to the python os.system library.

[CVE-2021-36668](#): URL injection in Driva inSync 6.9.0 for MacOS, allows attackers to force a visit to an arbitrary url via the port parameter to the Electron App.

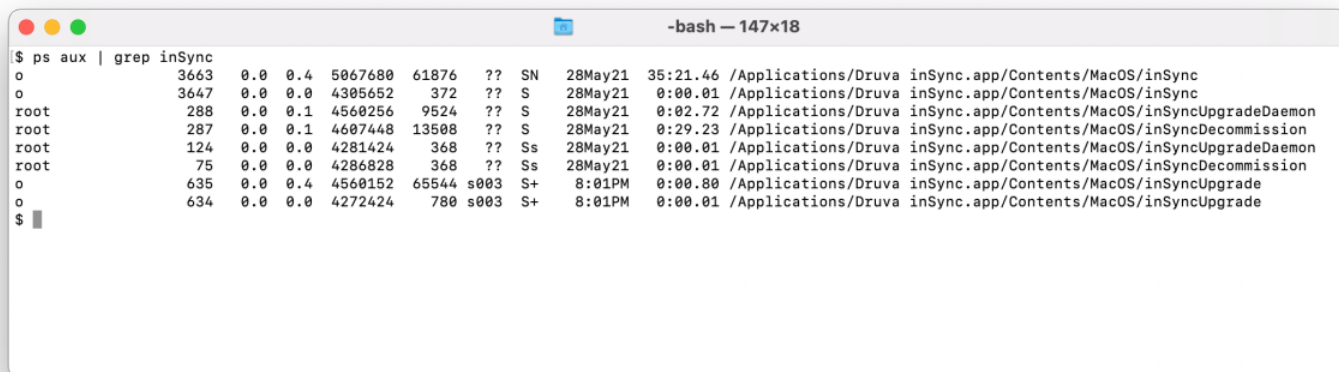
## What is Druva inSync?

[Druva inSync](#) is an enterprise solution for backing up user endpoints. The client is written in Python 2 (!) alongside an Electron app (powered by a very out of date version of Electron) that provides a UI for users.

On Mac there are several binaries bundled in the app package:

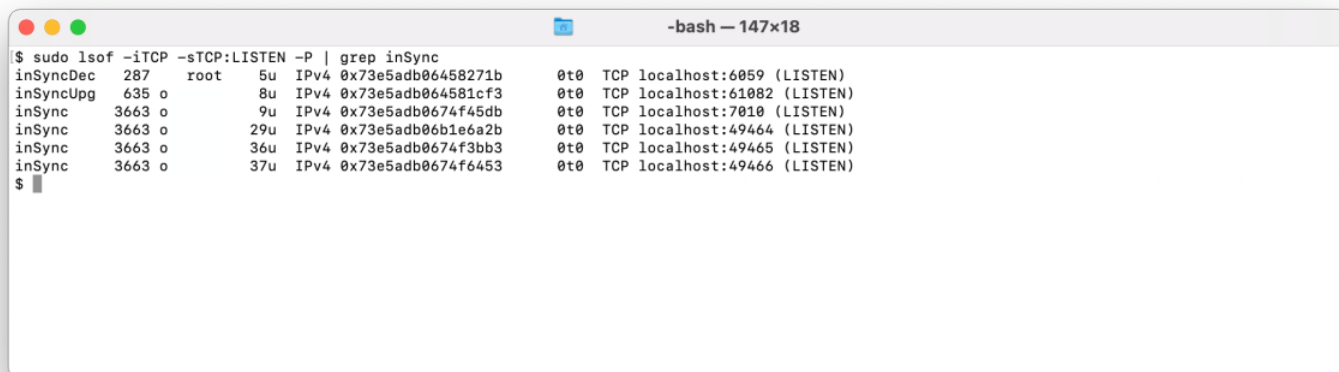
```
inSync
inSyncUpgrade
inSyncUpgradeDaemon
inSyncDecommission
```

They all run as part of normal operation:



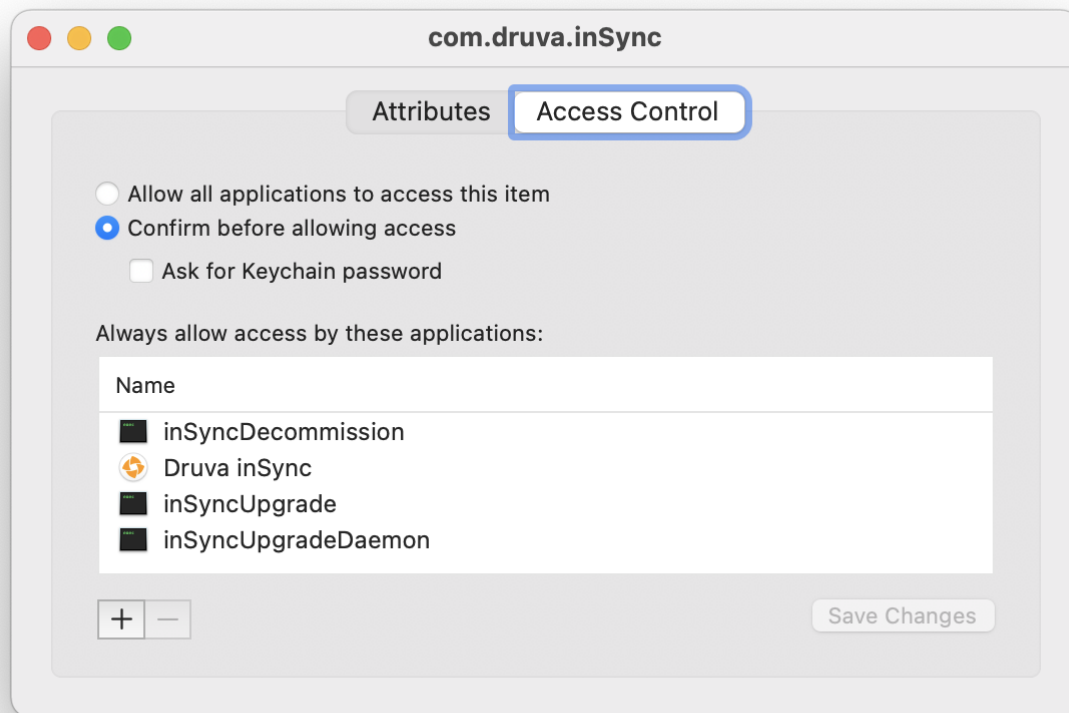
```
-bash — 147x18
$ ps aux | grep inSync
o        3663  0.0  0.4  5067680  61876  ??  SN   28May21   35:21.46  /Applications/Druva inSync.app/Contents/MacOS/inSync
o        3647  0.0  0.0  4305652    372  ??   S    28May21   0:00.01  /Applications/Druva inSync.app/Contents/MacOS/inSync
root     288   0.0  0.1  4560256    9524  ??   S    28May21   0:02.72  /Applications/Druva inSync.app/Contents/MacOS/inSyncUpgradeDaemon
root     287   0.0  0.1  4607448   13508  ??   S    28May21   0:29.23  /Applications/Druva inSync.app/Contents/MacOS/inSyncDecommission
root     124   0.0  0.0  4281424    368   ??  Ss   28May21   0:00.01  /Applications/Druva inSync.app/Contents/MacOS/inSyncUpgradeDaemon
root      75   0.0  0.0  4286828    368   ??  Ss   28May21   0:00.01  /Applications/Druva inSync.app/Contents/MacOS/inSyncDecommission
o        635   0.0  0.4  4560152   65544  s003  S+   8:01PM   0:00.80  /Applications/Druva inSync.app/Contents/MacOS/inSyncUpgrade
o        634   0.0  0.0  4272424    780  s003  S+   8:01PM   0:00.01  /Applications/Druva inSync.app/Contents/MacOS/inSyncUpgrade
$
```

They communicate with each other via HTTP or Unix socket, depending on the binary. Both inSyncDecommission and inSyncUpgradeDaemon are configured as LaunchDaemons and run as root. This means that if we can talk to either of them then we may be able to use them to perform our own actions with root privileges.



```
-bash — 147x18
$ sudo lsof -iTCP -sTCP:LISTEN -P | grep inSync
inSyncDec  287  root    5u  IPv4  0x73e5adb06458271b  0t0  TCP localhost:6059 (LISTEN)
inSyncUpg  635  o      8u  IPv4  0x73e5adb064581cf3  0t0  TCP localhost:61082 (LISTEN)
inSync     3663  o      9u  IPv4  0x73e5adb0674f45db  0t0  TCP localhost:7010 (LISTEN)
inSync     3663  o     29u  IPv4  0x73e5adb06b1e6a2b  0t0  TCP localhost:49464 (LISTEN)
inSync     3663  o     36u  IPv4  0x73e5adb0674f3bb3  0t0  TCP localhost:49465 (LISTEN)
inSync     3663  o     37u  IPv4  0x73e5adb0674f6453  0t0  TCP localhost:49466 (LISTEN)
$
```

Luckily, the communications between them are not fully unauthenticated. The user level binaries use a variety of keys stored in the Keychain to communicate with the root daemons. Those keys are accessed via a custom native Python module (packaged as `keychainAPI.so`) and the access is configured to allow each binary to read the secret without prompting for a password.



What does that mean for us? If we can convince one of the usermode programs to give up the keychain secret to us, we can use that ourselves to talk to the root daemons!

## Decompiling

To get anywhere, it will be helpful to inspect the Druva binaries further. On Mac the Druva applications are packaged using PyInstaller. We can use existing tooling to extract the original pyc, and then py files from the binaries. We also have to de-obfuscate the opcodes, which we can do using the [same technique](#) outlined in the last public vulnerability research into inSync.

Once we've decompiled the apps we can start poking around. `inSyncDecommission` and `inSyncUpgradeDaemon` are both running RPC servers that are authenticated (as expected) with keys from the keychain. Let's see what we can do.

## Exploit 1: inSyncUpgradeDaemon

Decompiling `inSyncUpgradeDaemonRPC.py` we can see the `daemon_install_package` method runs a shell script and blindly executes as root whatever package has been handed to it by the usermode `inSyncUpgrade`.

```
def daemon_install_package(ctx, installer, uname):
    try:
        import stat
        os.chmod(installer, stat.S_IXUSR)
        if not os.path.exists(inSyncParam.CLIENT_GLOBAL_ROOT):
            try:
                os.makedirs(inSyncParam.CLIENT_GLOBAL_ROOT)
            except Exception as fault:
                SyncLog.error('Could not create directory %s', inSyncParam.CLIENT_GLOBAL_ROOT)

        installfilepath = os.path.join(inSyncParam.CLIENT_GLOBAL_ROOT, 'install.sh')
        if os.path.exists('/Volumes/inSync'):
            ret, out = commands.getstatusoutput('hdiutil unmount /Volumes/inSync')
            ret, out = commands.getstatusoutput('hdiutil mount "%s"' % installer)
            if ret != 0:
                SyncLog.error('An error occurred while mounting dmg file')
            SyncLog.info('Creating %s', installfilepath)
            FILE = open(installfilepath, 'w')
            FILE.write('#!/bin/sh\n')
            FILE.write('\n')
            FILE.write('echo `date` "install.sh execution started"\n')
            FILE.write('cd /Volumes/inSync/\n')
            FILE.write('pkg_name=$(ls | grep -i pkg | head -n1)\n')
            FILE.write('if [ -z "$pkg_name" ]; then pkg_name="Install inSync.mpkg"; fi\n')
            FILE.write('installer -package "/Volumes/inSync/${pkg_name}" -target /\n')
            FILE.write('hdiutil unmount /Volumes/inSync -force\n')
            FILE.write('rm "%s"\n' % installer)
            user_home = '/Users/' + uname + inSyncParam.CLIENT_GLOBAL_ROOT
            ipkgfile = os.path.join(user_home, 'auto-upgrade', uname, 'binary')
            FILE.write('rm "%s"\n' % ipkgfile)
            FILE.write('echo `date` "install.sh execution ended"\n')
            FILE.write('() >>/tmp/inSyncInstall 2>&1')
            os.chmod(installfilepath, stat.S_IXUSR)
            cmdline = ['/bin/sh', installfilepath]
            try:
                pid = os.fork()
            except OSError as fault:
                SyncLog.error('Unable to install new package Error: %s', fault)
```

```
        raise Exception('%s [%d]' % (fault.strerror, fault.errno))

    if 0 == pid:
        os.setsid()
        os.execvp(cmdline[0], cmdline)
    except Exception as fault:
        SyncLog.error('Failure in daemon install. Fault %s', fault)
        SyncLog.traceback(fault)
```

Where does inSyncUpgrade get the package information from? It reads a URL out of `$HOME/Library/Application Support/inSync/inSync.cfg` and then asks that URL for a package over a REST API. `inSync.cfg` is user-writable, so we can replace that server with our own, and serve up a malicious binary. There is some home grown package signature verification in `inSyncPackage.py` but critically it calls `pickle.loads()` on the untrusted data:

```
def load(self, fpath, quick=False):
    f = file(fpath, 'rb')
    magic = f.read(len(PKG_MAGIC))
    if magic != PKG_MAGIC:
        raise PackageException('Not an inSync Package')
    ssize = struct.calcsize('<I')
    sbytes = f.read(ssize)
    if len(sbytes) < ssize:
        raise PackageException('Package file corrupt')
    sbytes = struct.unpack('<I', sbytes)[0]
    header = f.read(sbytes)
    if len(header) != sbytes:
        raise PackageException('Package file corrupt')
    headerBlk = None
    md5Header, md5Signature = Pickle.loads(header)
    ...
```

This is essentially game over. We can supply a malicious pickle that runs whatever code we want. Here we talk to the `UpgradeDaemon` and supply it a malicious filename that exploits a command injection vulnerability in the shell script it runs ( `FILE.write('rm "%s\n' % installer)` ), giving us a reverse shell as root. As part of this exploit we also take advantage of a URL injection vulnerability in the app to make this attack only require one line on the victim's system - if port starts with an `@` sign the ip address will be interpreted as a username and password:

```
inSyncURL = 'http://127.0.0.1:' + port + '/index.html';
```

Full Exploit Code

0:00 / 0:57

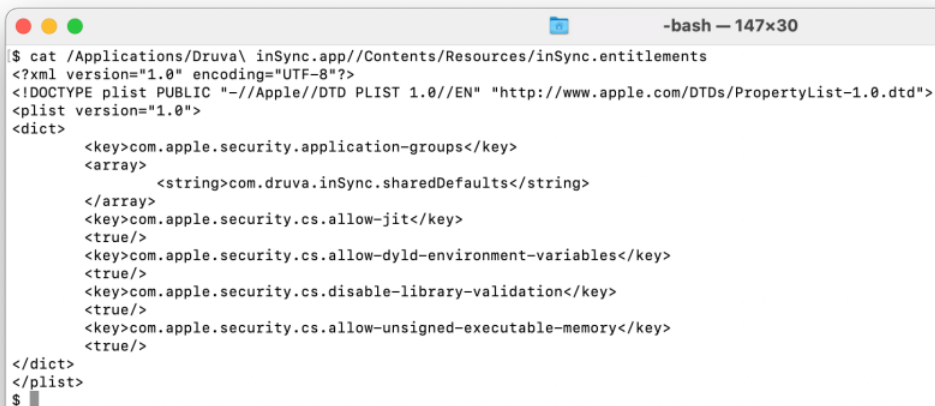
Exploit 2: inSyncDecommission

While we could use the same pickle vulnerability to talk to `inSyncDecommission` , let's take a different route to privilege escalation. `InSyncDecommission` runs an RPC server ( `inSyncDaemonRPC.py` ) that basically lets us do whatever we want on the system:

```
def register(server):
    server.register_auth_validator(validate_session)
    server.register_function(daemon_authenticate, 'daemon.authenticate', autoauth=False)
    server.register_function(daemon_mkdir, 'daemon.mkdir')
    server.register_function(daemon_makedirs, 'daemon.makedirs')
    server.register_function(daemon_open_file, 'daemon.open_file')
    server.register_function(daemon_write, 'daemon.write')
    server.register_function(daemon_read, 'daemon.read')
    server.register_function(daemon_close_file, 'daemon.close_file')
    server.register_function(daemon_set_file_acl, 'daemon.set_file_acl')
    server.register_function(daemon_set_acls, 'daemon.set_acls')
    server.register_function(daemon_set_file_time, 'daemon.set_file_time')
    server.register_function(daemon_set_xattr, 'daemon.set_xattr')
    server.register_function(daemon_remove_file, 'daemon.remove_file')
    server.register_function(daemon_rename_file, 'daemon.rename_file')
    server.register_function(daemon_autoupdate, 'daemon.autoupdate')
    server.register_function(daemon_system_restart, 'daemon.system_restart')
    server.register_function(daemon_renice, 'daemon.renice')
    server.register_function(daemon_set_secrets, 'daemon.set_secrets')
    server.register_function(daemon_delete_secrets, 'daemon.delete_secrets')
    server.register_function(daemon_delete_file, 'daemon.delete_file')
    server.register_function(daemon_symlink, 'daemon.symlink')
    server.register_function(daemon_mknod, 'daemon.mknod')
    server.register_function(daemon_mkfifo, 'daemon.mkfifo')
    server.register_function(daemon_move, 'daemon.move')
    server.register_function(daemon_create_device_refresh_path, 'daemon.create_device_refresh_path')
```

For this attack we're going to target `daemon_read` and `daemon_write`. This will let us read and write any file on the file system (as root). For a proof of concept we can write to `/etc/sudoers` allowing any user to run sudo without requiring a password.

inSync is compiled with the [Hardened Runtime](#) enabled. However, poking around the app package we find the entitlements file for the app which disables some critical protections:



```
-bash — 147x30
$ cat /Applications/Druva\ inSync.app//Contents/Resources/inSync.entitlements
<?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>com.apple.security.application-groups</key>
  <array>
    <string>com.druva.inSync.sharedDefaults</string>
  </array>
  <key>com.apple.security.cs.allow-jit</key>
  <true/>
  <key>com.apple.security.cs.allow-dyld-environment-variables</key>
  <true/>
  <key>com.apple.security.cs.disable-library-validation</key>
  <true/>
  <key>com.apple.security.cs.allow-unsigned-executable-memory</key>
  <true/>
</dict>
</plist>
$
```

This means that the hardened runtime protections do not apply to any libraries, and we can use the `DYLD_` environment variables to inject whatever libraries we want. Therefore we can simply inject a malicious dylib into the app and use the app's privilege to read out secrets from the keychain. From there it's simple enough to write a python script to talk to `inSyncDecommission` and own the system:

[Full Exploit Code](#)

0:00 / 0:19

## Bonus security hygiene issues

There are several other ports open by `inSync` and `inSyncUpgrade`. Since they're running with user level privileges I didn't look into them too deeply, but they probably shouldn't be running. Both services are running a debug server which allows (by design) arbitrary Python code execution. They're saved by needing to write to `/etc/inSyncServer` which is not possible by default. Once that directory is created, code execution is easy via the `runCmd` RPC command. The same script can be used as for exploiting Decommission, the socket merely needs to be wrapped in an SSL connection with `ssl.wrap_socket`.

`inSync` also listens on three other ports. One is an http server that seems to simple proxy to Druva's cloud service. The Electron app talks to this server (ignoring TLS validation errors) for operations such as restore.

There is also another http server running that the main Electron UI talks to. This also has a simple command injection vulnerability via the `filename` variable:

```
@cherry.py.expose
@post_activation_api_wrapper
def copy_to_clipboard(self, *args, **kwargs):
    try:
        cherry.py.response.headers['Access-Control-Allow-Origin'] = '*'
        cherry.py.response.headers['Access-Control-Allow-Methods'] = 'POST, GET'
        filename = kwargs.get('data', '')
        if sys.platform == 'win32':
            import pyperclip
            pyperclip.copy(filename)
        else:
            os.system("echo '%s' | pbcopy" % filename)
        return {'status': 0, 'msg': '', 'data': {}}
    except Exception as fault:
        return {'status': -1, 'msg': _(fault), 'data': {}}
```

The final server is another RPC service used by the Syncer.

Beyond the URL injection vulnerability in the Electron app already discussed, it is also several versions out of date and has none of the important security mitigations introduced in recent years such as [Context Isolation](#).

## Fix

These issues are fixed in `inSync Client 7.0.1`. Thanks to the Druva security team for their quick response to my report and resolving the issues.

## References

<https://medium.com/tenable-techblog/getting-root-on-macos-via-3rd-party-backup-software-b804085f0c9>

<https://medium.com/tenable-techblog/remapping-python-opcodes-67d79586bfd5>

<https://wojciechregula.blog/post/stealing-macos-apps-keychain-entries/>

## Timeline

| Date     | Action  |
|----------|---|
| 06/10/21 | Issue reported to Druva   |
| 06/16/21 | Followed up with Druva after receiving no reply                 |
| 06/16/21 | Confirmation from Druva that they have received the report      |
| 06/22/21 | Confirmation from Druva of repro of three out of four issues    |
| 06/30/21 | Confirmation from Druva of repro of the fourth issue            |
| 07/08/21 | CVEs requested from MITRE                                       |
| 07/24/21 | inSync Client v7.0.0 released that fixes the bulk of the issues |
| 09/22/21 | inSync Client v7.0.1 released with an additional fix            |
| 10/11/21 | inSync Client v7.0.1 re-released with a further fix             |
| 05/11/22 | CVE assignment finally received from MITRE                      |