ELFINDER: THE STORY OF A REPWNING

Rédigé par Gaetan Ferry - 30/03/2022 - dans Exploit , Pentest

We recently identified a path traversal issue in the elFinder software. It is assigned CVE identifier CVE-2022-26960. While the vulnerability is pretty classical, the story of its discovery is not. Keep on reading for the details.

elFinder has a long story of being affected by severe issues. Here, at Synacktiv, we already took part in that story when, in 2019, Thomas Chauchefoin disclosed a command injection affecting that product [1].

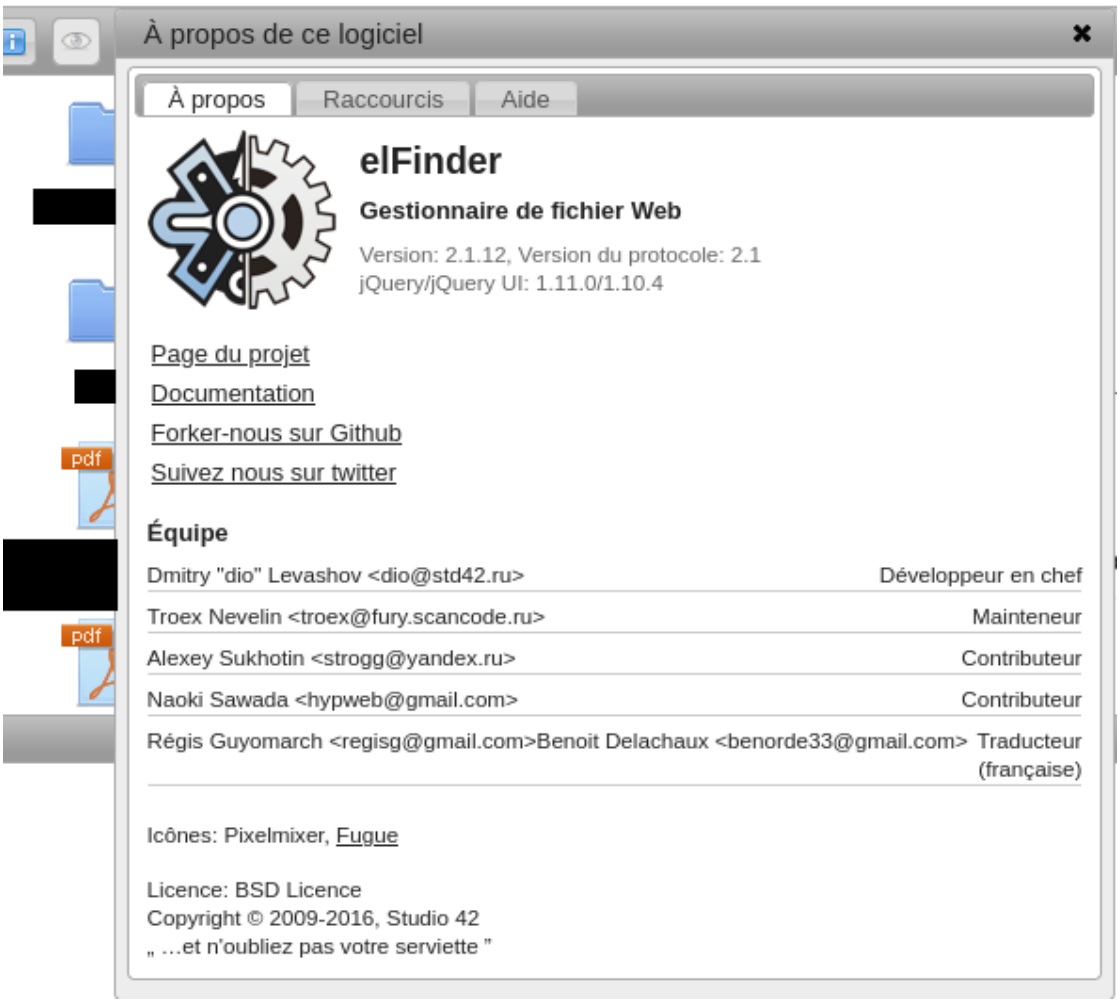
It is therefore with great pleasure that we face this product during our engagements. And, sometimes, vulnerabilities are on our way.

THE SITUATION

A few months ago, during regular intrusion tests of ours, a fuzz of the root of our target application raised an interesting result (thank the world for raft wordlists).

```
000017052:   301        9 L      28 W       321 Ch      "elfinder"
```

Reaching the discovered directory led to a shiny *elFinder* deployment. A few *click click* later the penalty strikes: version is 2.1.12. **AGES** old! (2016).



A 5 years old version!

The issue we (Thomas) found in 2019 affected all versions prior to 2.1.48. Our instance should therefore be affected, and we should be close to a remote code execution. However, if you read the advisory for CVE-2019-9194 you might notice a subtle detail:

> The implementation of $volume->resize() can be found in elFinder/php/elFinderVolumeDriver.class.php and will perform various operations to ensure that resizing was not explicitly disabled,

In our case, the resize feature was disabled, preventing the exploitation. However, with such an old version, one could expect other issues to be available. And, indeed, multiple ones exist.

CVE-2018-9109, for example, is a path traversal that allows reading arbitrary files. But it also deletes them, which was not that great on our production environment.

Then, you have CVE-2021-32682 and CVE-2021-23394 discovered at SonarSource, and by Thomas Chauchefoin again [2]. If you have not read this post yet, I invite you to do so. The TL;DR is a list of interesting vulnerabilities with critical impacts:

- File deletion
- File move
- PHP file upload (to remote code execution)
- Argument injection in the zip CLI (to remote code execution)

- Race condition (to remote code execution)

The fact is, our target, if not good with software updates, had a top-notch server hardening:

- No write permissions in the web root
- Disabled PHP handler in the elFinder upload directory
- Minimum deployment (no zip CLI)

We got ourselves without an easy go-to exploit.

> **Note:**
>
> If you ever doubt investing in hardening is necessary, this proves you wrong.

PWNING ANYWAY

What happened next is an insane scam, a vulnerability robbery, an unexpected stroke of luck.

BACK TO ELFINDER FEATURES

If you are not familiar with the software we are talking about, you only need to know it is nothing more than a file manager for the web. It as features like uploading and downloading files, zipping things, previewing doohickeys and so on.

For that, it exposes a single server entry point, named the connector, with a JSON like API. All API calls include a `cmd` parameter, with the action to perform, and often one or more `target` arguments representing the files to manipulate.

The `target` parameter is of the form `source_path`, where `source` is the storage in which the file is, and `path` the base64 encoded path of the file in the storage. We'll ignore the source part in our case study as it won't play any role. Just consider our source is always the file system root directory.

```
GET /elfinder/php/connector.minimal.php?cmd=file&target=l1_YWFhLnR4dA&download=1&reqid=17fbb7e
Host: 192.168.122.113
[...]
```

For example, in the previous request, the `file` action (AKA download) is called on the *aaa.txt* file. Relative paths are taken from the configured exposed directory.

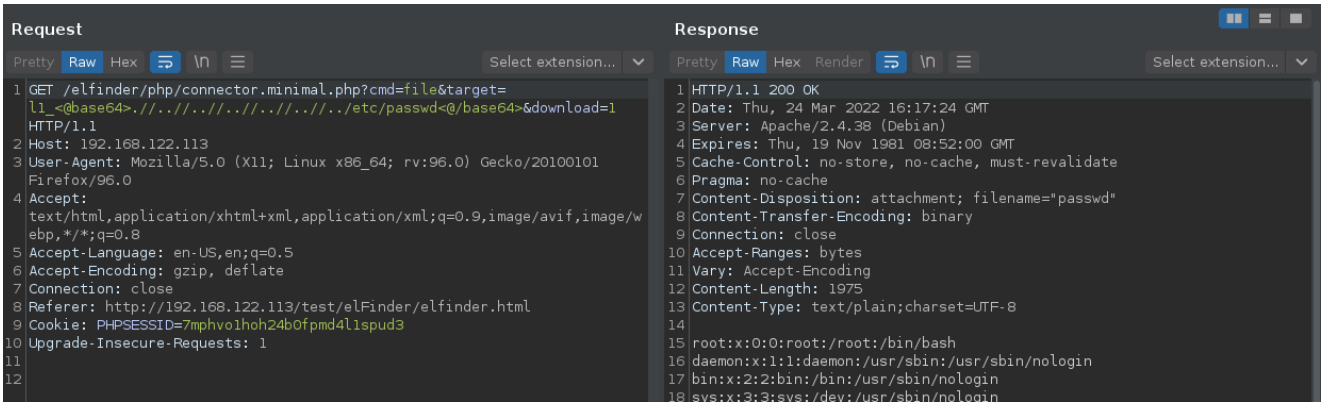
AND THEN BACK TO BUSINESS

Let's imagine we are in 2010 working on a software that has not been extensively reviewed. Let's say we face a software that handles file paths. To finish, imagine we want to test for path traversal issues. The first instinctive action would be to provide the application with a long *../* chain and to try reaching */etc/passwd* that way.

If that does not work we would probably try some bypasses, adding a few slashes, encoding on key characters and so on.

But this should not succeed in our case right? Not on a 6 years old version. Not on a software that has been thoroughly reviewed by talented people.

The fact is we tried.



Wait, WHAT!?

The fact is it passed.

EMOTIONAL ROLLERCOASTER

DOWN THE RABBIT HOLE

We found ourselves in a situation where we had an unexpected vulnerability, and no idea about why it is here and went unnoticed for such a long time. However, if we analyze our payload, what appears is that the application is probably messing up looking for ../and trying to delete them. This is often seen in beginner CTF web challenges.

Building upon our luck, let's grep for *"..".*

```
$ find . -name "*.php" -exec grep -HF ".." {} \;
./elFinderVolumeDriver.class.php:    * System Root path (Unix like: '/', Windows: '\', 'C:\' o
./elFinderVolumeDriver.class.php:                           $name = preg_replace('/^(.*?)(\..*
./elFinderVolumeDriver.class.php:                           $name != '.' && $name != '..' && $
./elFinderVolumeDriver.class.php:          if (substr($path, 0, 3) === '..' . $separator) {
./elFinderVolumeDriver.class.php:          // normalize `/../`
./elFinderVolumeDriver.class.php:                                    if ($file === '.' || $file
./elFinderVolumeDriver.class.php:                           if ($file !== '.' && $file !== '..
./elFinderVolumeDriver.class.php:                                                                if
./elFinderVolumeDriver.class.php:    * Ususaly used for images, but can be realize for video e
./elFinderVolumeFTP.class.php:              if (count($info) < 9 || $info[8] == '.' || $info[8
./elFinderVolumeFTP.class.php:                      if (($comp != '..')
./elFinderVolumeFTP.class.php:                      || ($new_comps && (end($new_comps) == '..'
./elFinderVolumeFTP.class.php:                      if ($name && $name !== '.' && $name !== '.
./elFinderVolumeFTP.class.php:       * Ususaly used for images, but can be realize for video e
./elFinderVolumeFTP.class.php:                      if ($name !== '.' && $name !== '..' && (!$
./elFinderVolumeFTP.class.php:              $excludes = array(".","..");

./elFinderVolumeLocalFileSystem.class.php:                  if (($comp != '..')
./elFinderVolumeLocalFileSystem.class.php:                  || ($new_comps && (end($new_comps)
./elFinderVolumeLocalFileSystem.class.php:   * Usualy used for images, but can be realize for
./elFinderVolumeDropbox.class.php:   * Ususaly used for images, but can be realize for video e
./elFinderVolumeMySQL.class.php:     * Usualy used for images, but can be realize for video et
```

There are not that many results. Even less if we suppose `elFinderVolumeFTP`, `elFinderVolumeDropbox` and `elFinderVolumeMySQL` probably have nothing to do with our issue. Among the results, a code comment might catch the eye:

> ./elFinderVolumeDriver.class.php: // normalize `/../`

Following that path leads to the `getFullPath` function from the `elFinderVolumeDriver` class. This abstract class implements common components that are used by what are called *drivers* that abstract the file operations for the different source types. We said we would only consider the local file system case, so let's not bother with that.

```
/**
     * Resolve relative / (Unix-like)absolute path
     *
     * @param string $path  target path
     * @param string $base  base path
     * @return string
     */
protected function getFullPath($path, $base) {
    [...]
    // normalize `/../`
    $normreg = '#('.$sepquoted.')[^'.$sepquoted.']+'.$sepquoted.'\.\.'.$sepquoted.'#'; // '#(/)[^\/]+/
\.\./#'
    while(preg_match($normreg, $path)) {
        $path = preg_replace($normreg, '$1', $path, 1);
    }

    [...]

    return $path;
}
```

We skipped irrelevant parts of the function to get to the heart of the question, and to the pattern we were looking for. And what we see is actually a regex that is being used to "normalize `/../`". We could ask: hey, what could go wrong?, but we already know the answer.

Thanks to the code comments, we know that the pattern being looking for is `#(/)[^\/]+/\.\./#` and all the matches are replaced with a */*. This process is repeated until no match is found.

But this pattern is wrong. The `//../` sequence will never be matched, allowing the path traversal we highlighted.

Let's fire up the "go to reference" feature of our IDE to validate this function is part of the call chain for the *file* action:

- Maj+F12: elFinderVolumeLocalFileSystem::_inpath
- Maj+F12: elFinderVolumeLocalFileSystem::_stat
- Maj+F12: elFinderVolumeDriver::stat
- Maj+F12: WOWOWOW

`elFinderVolumeDriver` calls the `stat` method more than 50 times in tenth of functions, including one called `file` (the name of the action we called on the API). Most of those functions are controllers for actions which mean our issue might have a broader impact than just reading arbitrary files.

BACK TO THE FUTURE

At the time of our engagements, this discovery was enough to get RCE. Our target was running a *Laravel* instance for which we leaked the cookie signing secret. You know the sequel.

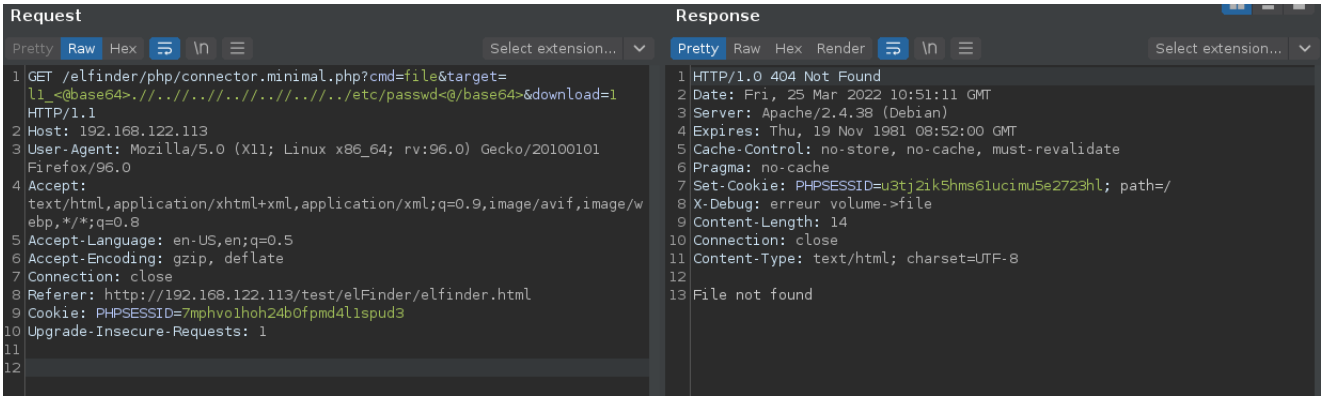
The next question that arose was: is this issue still present in the latest release? That question was legitimate as about 50 minor versions were released since the one we tested against.

```
$ git diff --ignore-cr-at-eol --ignore-space-at-eol -w --ignore-blank-lines 2.1.12 2.1.60 -- p
[...]
-       protected function getFullPath($path, $base) {
+    protected function getFullPath($path, $base)
+    {
        $separator = $this->separator;
        $systemroot = $this->systemRoot;
+        $base = (string)$base;

-               if ($base[0] === $separator && strpos($base, 0, strlen($systemroot)) !== $syst
+        if ($base[0] === $separator && substr($base, 0, strlen($systemroot)) !== $systemroot)
            $base = $systemroot . substr($base, 1);
        }
+        if ($base !== $systemroot) {
+            $base = rtrim($base, $separator);
+        }

        // 'Here'
        if ($path === '' || $path === '.' . $separator) return $base;
@@ -5259,6 +6807,9 @@ abstract class elFinderVolumeDriver {
        while (preg_match($normreg, $path)) {
            $path = preg_replace($normreg, '$1', $path, 1);
        }
+        if ($path !== $systemroot) {
+            $path = rtrim($path, $separator);
+        }
```

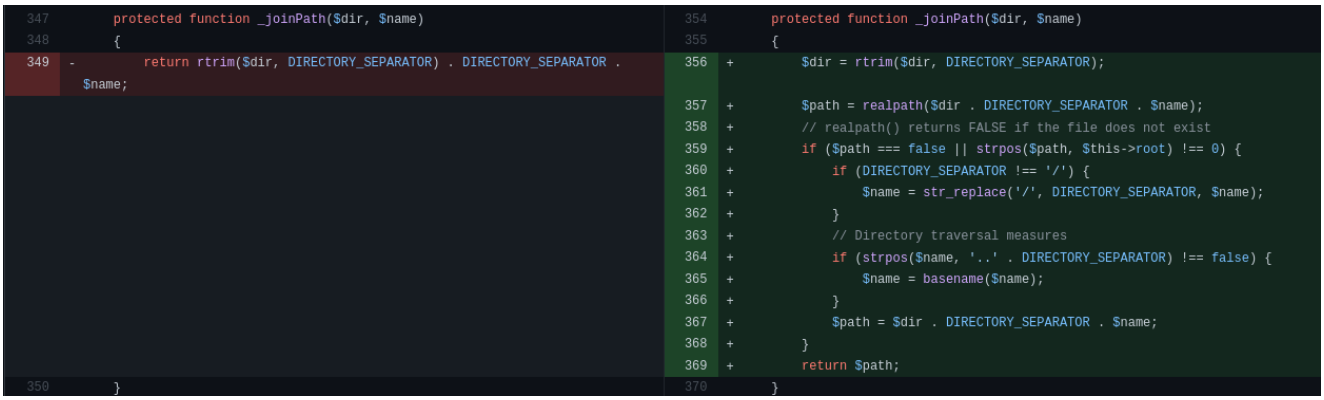
The diff between the 2.1.12 and the latest version speaks for itself. No significant change was made to the vulnerable `getFullPath` function. Right before cracking open the champagne, we threw the exploit payload to a test deployment of the latest version.



Wait, WHAT!?

So our vulnerable function did not change. Yet, our exploit payload does not work anymore. Sure, another change in the software must prevent the directory traversal.

Among the vulnerabilities reported by SonarSource [2], two actually are path traversal related. The `_joinPath` method of the `elFinderVolumeLocalFileSystem` was the root cause for this issue. It concatenated two paths without any checks and was patched the following way:



```
protected function _joinPath($dir, $name)
{
-       return rtrim($dir, DIRECTORY_SEPARATOR) . DIRECTORY_SEPARATOR . $name;
}
```

```
protected function _joinPath($dir, $name)
{
+       $dir = rtrim($dir, DIRECTORY_SEPARATOR);
+
+       $path = realpath($dir . DIRECTORY_SEPARATOR . $name);
+       // realpath() returns FALSE if the file does not exist
+       if ($path === false || strpos($path, $this->root) !== 0) {
+           if (DIRECTORY_SEPARATOR !== '/') {
+               $name = str_replace('/', DIRECTORY_SEPARATOR, $name);
+           }
+           // Directory traversal measures
+           if (strpos($name, '..' . DIRECTORY_SEPARATOR) !== false) {
+               $name = basename($name);
+           }
+           $path = $dir . DIRECTORY_SEPARATOR . $name;
+       }
+       return $path;
}
```

The patch for the _joinPath function directory traversal

While the function was previously just returning a concatenation of two paths, without further validation or sanitation, the new version contains a few security measures. In particular, a call to `realpath` might be responsible for our problems if it happens that the `_joinPath` function is part of the call chain for the *file* action.

- Maj+F12: Direct usages of this function are found in few controllers not related to the file action.
  Also `elFinderVolumeLocaleFileSystem::_abspath`.
- Maj+F12: `elFinderVolumeDriver::abspathCE`
- Maj+F12: `elFinderVolumeDriver::decode`
- Maj+F12: pretty much everywhere

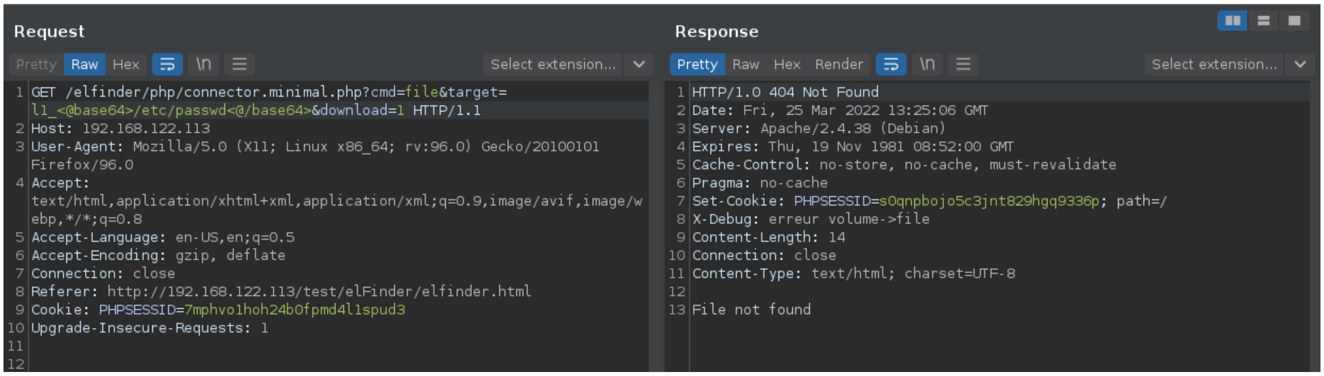
The `decode` method from `elFinderVolumeDriver` is actually the one that is used to decode the *target* kind of arguments (AKA from a l1_AAAA parameter, it extracts the actual file pointer). It is therefore called in most action controllers, including the one for *file* calls. There is no way you can go around the `decode` call.

However, if we go back down the call chain to `_joinPath` and stop on the `elFinderVolumeLocaleFileSystem::_abspath`, it appears it will not always call the doomed `_joinPath` function.

```
protected function _abspath($path)
{
    if ($path === DIRECTORY_SEPARATOR) {
        return $this->root;
    } else {
        if (strpos($path, $this->systemRoot) === 0) {
            return $path;
        } else if (DIRECTORY_SEPARATOR !== '/' && preg_match('/^[a-zA-Z]:' . preg_quote(DIRECTORY_SEPARATOR, '/') . '/', $path)) {
            return $path;
        } else {
            return $this->_joinPath($this->root, $path);
        }
    }
}
```

In fact, `_joinPath` is only called when the path being queried is a relative path. Using an absolute path should then be enough to exploit the vulnerability on the latest version.

Of course, just throwing the absolute path to */etc/passwd* at the application won't do the trick.



Would have been to easy.

The answer to the question "what happens if the path is absolute?" is in fact in the call chain to the `getFullPath` function. If you look back at it, you will see that the chain goes through a function named `_inpath`. This one is said to prevent directory traversal.

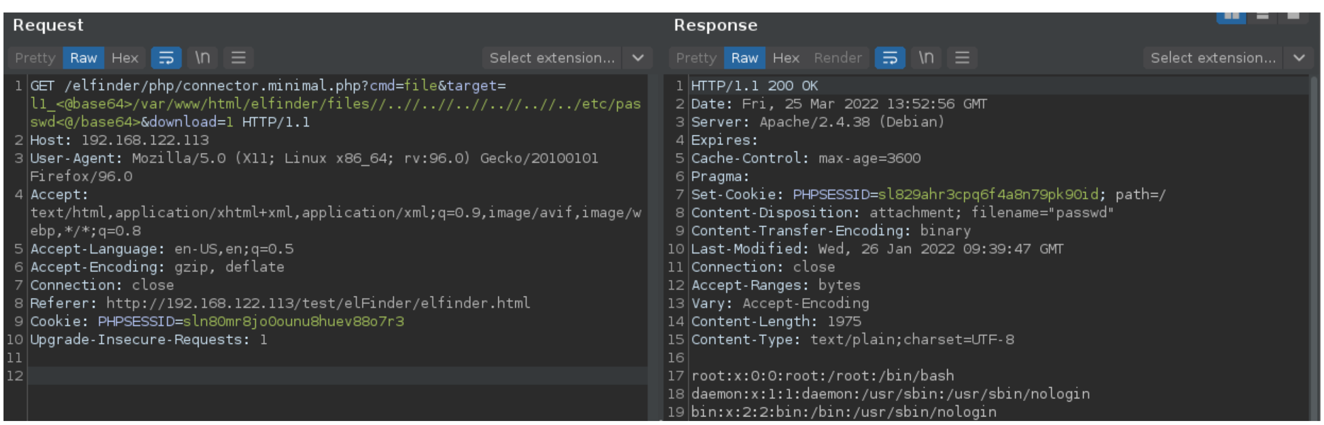
```
protected function _stat($path)
{
    $stat = array();

    if (!file_exists($path) && !is_link($path)) {
        return $stat;
    }

    //Verifies the given path is the root or is inside the root. Prevents directory traveral.
    if (!$this->_inpath($path, $this->root)) {
        return $stat;
    }
```

As it is explained in the comments, the `_inpath` is used to validate that the requested path is inside the configured web root directory. Even if we know this part of the software fails at handling *dot dot slashes* there is no chance that */etc/passwd* is considered inside the *root*. For that to happen, the provided path needs to start with the configured root path as a prefix.

Well, that's nothing like impossible. If we build a path that actually starts with that prefix, and exploit the double slash trick to inject unsanitized *../* sequences after it, we should be able to pass all the checks and sanitation.



Repwned!

WRAPPING UP

If you got this far in the post, you probably know the issue is a path traversal vulnerability in the *elFinder* software. The impact depends a lot on the context the thing is deployed in. At the very least, an arbitrary file read is possible. It can be sufficient to get remote code execution in some cases.

As we said earlier, the security function we bypassed is used in multiple action controllers. It should be able to call actions like:

- search
- upload
- parents
- subdirs

With correct permissions, this is equivalent to reading, writing and browsing the server's file system.

elFinder implements other controls, on file extensions in particular, that might prevent writing arbitrary PHP scripts or so. However, this is not necessary to get code execution on a host. You might just replace *authorized_keys* files, crontabs, etc.

To finish, yes, exploiting this issue on the latest version requires knowing the file system path to the *elFinder* root directory. This one might be guessed (/var/www/html? /srv/php?), brute forced, or leaked via another vulnerability. Left as an exercise for the reader.

This is not required on all versions before 2.1.59. But in those, you might as well exploit the RCE reported by Thomas from Sonar. Unless you are as lucky as we are...

TIMELINE

Date	Action
2022-03-08	Vulnerability reported
2022-03-09	Patch proposed by elFinder team
2022-03-14	CVE-2022-26960 assigned
2022-03-14	elFinder 2.1.61 released with fix

REFERENCES

[1] Command Injection in elFinder < 2.1.48 https://www.synacktiv.com/ressources/advisories/elFinder_2.1.47_Command_Injection.pdf

[2] elFinder - A Case Study of Web File Manager Vulnerabilities https://blog.sonarsource.com/elfinder-case-study-of-web-file-manager-vulnerabilities