snyk

Zip Slip

Zip Slip is a widespread critical archive extraction vulnerability, allowing attackers to write arbitrary files on the system, typically resulting in remote command execution. It was discovered and responsibly disclosed by the Snyk Security team ahead of a public disclosure on 5th June 2018, and affects thousands of projects, including ones from HP, Amazon, Apache, Pivotal and many more (CVEs and full list here).

The vulnerability has been found in multiple ecosystems, including JavaScript, Ruby, .NET and Go, but is especially prevalent in Java, where there is no central library offering high level processing of archive (e.g. zip) files. The lack of such a library led to vulnerable code snippets being hand crafted and shared among developer communities such as StackOverflow.

The vulnerability is exploited using a specially crafted archive that holds **directory traversal** filenames (e.g. ../../evil.sh). The Zip Slip vulnerability can affect numerous archive formats, including tar, jar, war, cpio, apk, rar and 7z.

Zip Slip is a form of directory traversal that can be exploited by extracting files from an archive. The premise of the directory traversal vulnerability is that an attacker can gain access to parts of the file system outside of the target folder in which they should reside. The attacker can then overwrite executable files and either invoke them remotely or wait for the system or user to call them, thus achieving **remote command execution** on the victim's machine. The vulnerability can also cause damage by overwriting configuration files or other sensitive resources, and can be exploited on both client (user) machines and servers.

Exploitable Application Flow

The two parts required to exploit this vulnerability is a malicious archive and extraction code that does not perform validation checking. Let's look through each of these in turn. First of all, the contents of the zip file needs to have one or more files that break out of the target directory when extracted. In the example below, we can see the contents of a zip file. It has two files, a `good.sh` file which would be extracted into the target directory and an `evil.sh` file which is trying to traverse up the directory tree to hit the root and then add a file into the tmp directory. When you run `cd ..` from the root directory, you will still find yourself in the root directory, so a malicious path could contain many levels of `../` to stand a better chance of reaching the root directory, before trying to traverse to sensitive files.

```
5 Tue Jun 5 11:04:29 BST 2018 good.sh
20 Tue Jun 5 11:04:42 BST 2018 ../../../../../../../../tmp/evil.sh
```

The contents of this zip file have to be hand crafted. Archive creation tools don't typically allow users to add files with these paths, despite the zip specification allowing it. However, with the right tools, it's easy to create files with these paths.

The second thing required in order to be exploitable is to have the functionality to extract the archive, either using your own code or a library. The vulnerability exists when the extraction code omits validation on the file paths in the archive. An example of a vulnerable code snippet (example shown in Java) can be seen below.

```
1:   Enumeration<ZipEntry> entries = zip.getEntries();
2:   while (entries.hasMoreElements()) {
3:      ZipEntry e = entries.nextElement();
4:      File f = new File(destinationDir, e.getName());
5:      InputStream input = zip.getInputStream(e);
6:      IOUtils.copy(input, write(f));
7:   }
```

You can see on line 4, `e.getName()` is concatenated with the target directory, `dir`, without being validated. At this point, when our zip archive gets to our evil.sh, it will append the full path (including every `../`) of the zip entry to the target directory resulting in evil.sh being written outside of the target directory.

To see Zip Slip in action, watch us exploit the vulnerable java-goof application, a sample application used to show many known vulnerabilities.

Are you Vulnerable?

You are vulnerable if you are either using a library which contains the Zip Slip vulnerability or your project directly contains vulnerable code, which extracts files from an archive without the necessary directory traversal validation. Snyk is maintaining a GitHub repository listing all projects that have been found vulnerable to Zip Slip and have been responsibly disclosed to, including fix dates and versions. The repository is open to contributions from the wider community to ensure it holds the most up to date status.

What action should you take?

Here are some steps you can take to check if your project's dependencies of code contain the Zip Slip vulnerability:

1. Search through your projects for vulnerable code.

In each ecosystem section, you'll see example snippets of code highlighting the specific vulnerability. The accompanying validation code can be added to the vulnerable snippet to test for directory traversal. You should search through your code for similar extract patterns, and ensure you're on the fixed versions of the archive processing libraries that we have found to be vulnerable.

Java

As previously mentioned, the Java ecosystem doesn't offer a central library containing high level processing of archive files. The popular Oracle and Apache commons-compress APIs that are heavily used do offer some archiving support but do not publically provide the full extract capability. This has contributed to there being more instances of users hand crafting the archive processing code themselves. We observed that the Java ecosystem had many more archive libraries than other ecosystems, many of which were found to be vulnerable.

Example Vulnerable code:

```
1:   Enumeration<ZipEntry> entries = zip.getEntries();
2:   while (entries.hasMoreElements()) {
3:      ZipEntry e = entries.nextElement();
4:      File f = new File(destinationDir, e.getName());
5:      InputStream input = zip.getInputStream(e);
6:      IOUtils.copy(input, write(f));
7:   }
```

Example Validation Code:

```
1:   String canonicalDestinationDirPath = destinationDir.getCanonicalPath();
2:   File destinationfile = new File(destinationDir, e.getName());
3:   String canonicalDestinationFile = destinationfile.getCanonicalPath();
4:   if (!canonicalDestinationFile.startsWith(canonicalDestinationDirPath + File.separator)) {
5:       throw new ArchiverException("Entry is outside of the target dir: " + e.getName());
6:   }
```

Groovy

Like Java, Groovy also has vulnerable snippets in various project codebases, as well as making use of all the vulnerable Java archive processing libraries.

Example Vulnerable code:

```
1:   final zipInput = new ZipInputStream(new FileInputStream(self))
2:   zipInput.withStream {
3:    def entry
4:    while(entry = zipInput.nextEntry)  {
5:      final file = new File(dest, entry.name)
6:      file.parentFile?.mkdirs()
7:      def output = new FileOutputStream(file)
8:      output.withStream {
9:        output << zipInput
10:     }
11:     unzippedFiles << file
12:   }
13: }
```

Example Validation Code:

```
1:   final canonicalDestinationDirPath = destinationDir.getCanonicalPath()
2:   final destinationfile = new File(destinationDir, e.name)
3:   final canonicalDestinationFile = destinationfile.getCanonicalPath()
4:   if (!canonicalDestinationFile.startsWith(canonicalDestinationDirPath + File.separator))
{
5:       throw new ArchiverException("Entry is outside of the target dir: ${e.name}")
6:   }
```

JavaScript

JavaScript has benefitted from having more central libraries that provide the functionality to extract from archives and the vulnerable libraries we found before public disclosure were fixed. Note that the join command concatenates the two path parameters and returns the shortest path possible after being resolved.

Example Vulnerable code:

```
1:   self.on('entry', function(entry) {
2:     entry.pipe(Writer({
3:       path: path.join(opts.path,entry.path)
4:   }))
```

Example Validation Code:

```
1:   var filePath = path.join(targetFolder, entry.path);
2:   if (filePath.indexOf(targetFolder) != 0) {
3:     return;
4:   }
```

.Net

The .Net ecosystem also has central libraries that perform the extraction functionality. In fact the code in the core .Net library that checks for the Zip Slip vulnerability was so neat, we used the implementation as the example reference solution to other libraries and ecosystems.

Example Vulnerable code:

```
1:   public static void WriteToDirectory(IArchiveEntry entry,
2:                                       string destDirectory,
3:                                       ExtractionOptions options){
4:      string file = Path.GetFileName(entry.Key);
5:      string destFileName = Path.Combine(destDirectory, file);
6:      entry.WriteToFile(destFileName, options);
7:   }
```

Example Validation Code:

```
1:   destFileName = Path.GetFullPath(Path.Combine(destDirecory, entry.Key));
2:   string fullDestDirPath = Path.GetFullPath(destDirectory + Path.DirectorySeparatorChar);
3:   if (!destFileName.StartsWith(fullDestDirPath)) {
4:      throw new ExtractionException("Entry is outside of the target dir: " + destFileName);
5:   }
```

Go

The Go ecosystem only has one vulnerable library that we found which was fixed within two days of us disclosing the issue. Note that the Join command concatenates the two path parameters and returns the shortest path possible after being resolved.

Example Vulnerable code:

```
1:   func (rarFormat) Read(input io.Reader, dest string) {
2:     rr := rardecode.NewReader(input, "")
3:     for {
4:       header := rr.Next()
5:       writeNewFile(filepath.Join(dest, header.Name), rr, header.Mode())
6:     }
7:   }
```

Example Validation Code:

```
1:   func sanitizeExtractPath(filePath string, destination string) error {
2:      destpath := filepath.Join(destination, filePath)
3:      if !strings.HasPrefix(destpath, destination) {
4:        return fmt.Errorf("%s: illegal file path", filePath)
5:     }
6:     return nil
7:   }
```

Ruby & Python

We also vetted the Ruby and Python ecosystems and couldn't find any vulnerable code snippets or libraries. In fact, Python's zipfile was vulnerable until fixed in 2014. Ruby has a number of existing vulnerabilities that have been fixed in previous versions here, here and here, and has a few more libraries that are prone to be used incorrectly because of the lack of a high level extraction API.

2. Add Zip Slip Security Testing to your application build pipeline

If you'd prefer not to search through your direct and transitive dependencies (of which you likely have hundreds) to determine if you're using a vulnerable library, you can choose a dependency vulnerability scanning tool, like Snyk. It's a good practice to add security testing into your development lifecycle stages, such as during development, CI, deployment and production. You can test your own projects (all the ecosystems mentioned above are supported) to determine if they are vulnerable to Zip Slip.

Other Vulnerable projects

Vulnerable projects include projects in various ecosystems that either use the libraries mentioned above or directly include vulnerable code. Of the many thousands of projects that have contained similar vulnerable code samples or accessed vulnerable libraries, the most significant include: Oracle, Amazon, Spring/Pivotal, Linkedin, Twitter, Alibaba, Jenkinsci, Eclipse, OWASP, SonarCube, OpenTable, Arduino, ElasticSearch, Selenium, Gradle and JetBrains.

Thank you!

The Snyk security team would like for thank all the vendors, project owners and the community members that helped raise awareness, find and fix vulnerabilities in projects across many ecosystems.

Zip Slip Disclosure Timeline

This disclosure timeline details our actions from the first private disclosure on April 15th 2018.

Date	Event
15/04/2018	Initial private disclosure to codehous/plexus-archiver, zip4j, adm-zip, unzipper, mholt/archiver
15/04/2018	Confirmed codehous/plexus-archiver, zip4j, mholt/archiver
17/04/2018	Confirmed adm-zip
17/04/2018	unzipper fix released, v0.8.13, (CVE-2018-1002203)
17/04/2018	Snyk submitted a fix to mholt/archiver and unzipper
17/04/2018	mholt/archiver fix released (CVE-2018-1002207)
18/04/2018	Private disclosure to ZeroTurnaround (zt-zip)
19/04/2018	Private disclosure to Apache (multiple projects affected)
20/04/2018	Apache confirmed the issue, collaborated triage began
21/04/2018	Apache Ant fix released, v1.9.12
22/04/2018	Snyk submitted a fix to adm-fix
23/04/2018	adm-zip fix released, v0.4.9 (CVE-2018-1002204)
25/04/2018	Dislocsed to DotNetZip.Semverd, SharpCompress
26/04/2018	zt-zip confirmed and fixed, v1.13 (CVE-2018-1002201)
27/04/2018	HP Fortify Cloud Scan Jenkins Plugin fix released, v1.5.1
02/05/2018	Snyk submitted a fix to SharpCompress
02/05/2018	Apache Storm confirmed vulnerable and fixed (CVE-2018-8008)
03/05/2018	Private disclosure to OWASP DependencyCheck
03/05/2018	Private disclosure to SonarQube
04/05/2018	OWASP DependencyCheck fix released (3.2.0)
04/05/2018	SonarQube fixed
06/05/2018	Snyk submitted a fix to plexus-archiver, fix released, v3.6.0 (CVE-2018-1002200)
07/05/2018	DotNetZip.Semverd fix released, 1.11.0 (CVE-2018-1002205)
09/05/2018	Private disclosure to Pivotal Security Team
09/05/2018	Private disclosure to Oracle Security Team
09/05/2018	Pivotal spring-zip-integration fixed (CVE-2018-1261)

09/05/2018	SharpCompress fix released, v0.21.0 (CVE-2018-1002206)
12/05/2018	Apache Commons Compress documentation fixed
14/05/2018	Pivotal eclipse-integration-gradle fixed, v3.9.4
22/05/2018	LinkedIn removed vulnerable implementation from Pinot project
23/05/2018	Apache Hadoop and Hive confirmed vulnerable and fixed (CVE-2018-8009)
26/05/2018	Oracle fixed documentation
31/05/2018	Amazon aws-toolkit-eclipse fix released, v201805311643