Devfile file write vulnerability in GitLab

Devfile file write vulnerability in GitLab - joern, gitlab-com.gitlab.io.

- Published: date not stated
- Original: <https://gitlab-com.gitlab.io/gl-security/security-tech-notes/security-research-tech-notes/devfile/>
- Preserved from: https://gitlab-com.gitlab.io/gl-security/security-tech-notes/security-research-tech-notes/devfile/ (live) on 2026-08-09
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

Devfile file write vulnerability in GitLab

This post is a extensive walkthrough of the process of identifying and exploiting CVE-2024-0402.

The vulnerability in GitLab was fixed on January 25th 2024 with a Critical Security Release.

Sometimes exploits for vulnerabilities are like onions: they have layers. This particular exploit had several layers and two underlying vulnerabilities which were used in combination to achieve an arbitrary file write on a GitLab instance. This file write then can be further abused to run arbitrary commands on a GitLab instance. This post will cover the general approaches to identify such issues and the technical details the exploit relies upon. However we'll spare the full end-to-end exploit as we deem the underlying techniques and approaches much more relevant in general than sharing an exploit for a patched vulnerability.

Starting point: dependencies

The adventure started by looking at the dependencies of the `GitLab` main project.

While looking for some low-hanging fruit in the project dependencies I noticed the `devfile` Gem making calls to an external helper binary by using `Open3.capture3`. Calling external binaries is a typical red flag for me, there are many things which might go sideways. Command or Argument injections for example, or other surprises from lesser known features of the called binary. Not to speak of actual vulnerabilities in the binary or its dependencies.

Notably the `devfile` Gem is written in-house at GitLab. A quick review of the repository revealed some Go based code from which the `devfile` binary called by the Ruby Gem was being created. I

didn't know much about Devfiles at this point. I only had the vague concept in the back of my head that those were YAML files used to describe the environments for GitLab's Workspaces feature.

Workspaces are isolated web-based development environments which are deployed by the GitLab application into Kubernetes clusters. Zooming out a bit we have:

- Devfiles: YAML files which are used to describe Workspaces in Kubernetes environments
- A Ruby library which parses those devfiles by calling a Go based helper binary

A quick check of the Ruby code calling the Go binary showed there were no surprises or low hanging fruits. The binary is being called directly with no shell being involved, simple command injections were not possible. Also the Go binary had no opportunity for Argument injections.

Digging deeper

Sometimes it's great to just start messing around with a software to get a feeling for potential vulnerabilities. The design of the `devfile` Gem made this easy. I could just use the included Go based binary and feed it some YAML. Digging into the documentation and looking for some sample files to use I came across the `parent` feature which allows another `devfile` to be specified. That file is then used as a base for your current `devfile`. I used the example from the documentation with the `devfile` binary in the Gem like so:

```
joern@host2:~/projects/deps/ruby/3.2.0/gems/devfile-0.0.25.pre.alpha1-x86_64-linux/bin$ ls
devfile
joern@host2:~/projects/deps/ruby/3.2.0/gems/devfile-0.0.25.pre.alpha1-x86_64-linux/bin$ ./devfile flatten 'schemaVer
metadata:
  name: my-project-dev
parent:
  uri: https://raw.githubusercontent.com/devfile/registry/main/stacks/nodejs/devfile.yaml'
failed to populateAndParseDevfile: error getting devfile info from url: failed to retrieve https://raw.githubusercontent.co
```

I was quite surprised when I did a `ls` in the directory after that command:

```
joern@host2:~/projects/deps/ruby/3.2.0/gems/devfile-0.0.25.pre.alpha1-x86_64-linux/bin$ ls
2.1.1  2.2.0  devfile  OWNERS  stack.yaml
```

A directory from the `devfile/registry` repository has been copied into the working directory where I ran the `./devfile` command. This was the moment I figured I might be on to something worth spending some more time on.

One step back and three steps forward

The `I'll copy some stuff from the Internet into your working directory when you parse a devfile` thing was very promising. I was hoping to use this vector to gain an arbitrary file write from it. Should be easy

enough, all I needed to do was to figure out the code path from within GitLab to trigger the dependency to flatten a devfile I crafted. That, and yes maybe a traversal out of the working directory, maybe not. That would depend on my current unknowns where on a GitLab instance this working directory would be and what I would control in terms of written files. But overall, this issue looked like a juicy starting place.

So I took a deeper dive into the main Ruby on Rails codebase of GitLab, just to find out that there's a validation to prevent the usage of that `parent` feature in a `devfile`. This was really inconvenient, I almost saw that awesome file write vanish in that line of code:

```
return err(_("Inheriting from 'parent' is not yet supported")) if devfile['parent']
```

But it wasn't really that bad, this roadblock was something for which I was prepared. In May 2023 I was "messing" with YAML files. This was inspired by a blog post from Jake Miller about parser differentials in JSON parsers, as JSON and YAML are somewhat similar in their use cases, but YAML is a bit more complex I thought it might be worthwhile looking at YAML from a parser differential perspective. Back then I was able to craft a YAML file which would parse differently in Ruby and Python. However it would parse the same in Ruby and in Go, so I "just" needed to find a similar parser differential in Go and in Ruby. Let's first look at the initial YAML file targeting Ruby/Go vs. Python:

```
joern@host2:~/projects/devfile$ cat 1.yaml
test: python
!!binary dGVzdA==: ruby & go
joern@host2:~/projects/devfile$ python -c 'import yaml;x = yaml.safe_load(open("1.yaml"));print(x["test"])'
python
joern@host2:~/projects/devfile$ ruby -ryaml -e 'x = YAML.safe_load(File.read("1.yaml"));puts x["test"]'
ruby & go
joern@host2:~/projects/devfile$ cat g.go
package main

import (
	"fmt"
	"log"
	"os"
	"gopkg.in/yaml.v3"
)

func main() {
   data, _ := os.ReadFile(os.Args[1])
	unmarshalled := &yaml.Node{}

	err := yaml.Unmarshal([]byte(data), unmarshalled)
	if err != nil {
		log.Fatalf("error: %v", err)
	}
	var expanded interface{}
	err = unmarshalled.Content[0].Decode(&expanded)
	if err != nil {
		log.Fatalf("error: %v", err)
	}

	d, err := yaml.Marshal(expanded)
	if err != nil {
		log.Fatalf("error: %v", err)
	}
	fmt.Printf("%s\n", string(d))
}

joern@host2:~/projects/devfile$ go run g.go 1.yaml
test: ruby & go
```

The very simple "secret sauce" here is using the `!!binary` notation to introduce a Base64 encoded key:

```
test: python
!!binary dGVzdA==: ruby & go
```

The Base64 (`!!binary`) string `dGVzdA==` becomes `test` when being decoded. In Ruby and in Go this will overwrite the previously defined `test: python` value. But in Python the following will happen:

```
python -c 'import yaml;y = yaml.safe_load(open("1.yaml"));print(y)'
{'test': 'python', b'test': 'ruby & go'}
```

The `!!binary` notation will create a Binary Sequence (`b'test'`) in Python which is different from the string `test` . So we'll have two keys, `test` and `b'test'` here, instead of one overwriting the other like it happens in Ruby and Go.

This behavior was the base I had ready, would GitLab have been a Python code base with the same Go parser backend this would've been ready to use right away to bypass the `parent` key filtering. Unfortunately this isn't the case so I had to find a different way to bypass the key filter.

I spent a bit of time reading about Tags in YAML and noticed the line `Local tags start with “!”` . Well OK then I thought, let's just try and see what happens when I use `!binary` instead of `!!binary` :

```
joern@host2:~/projects/devfile$ cat 2.yaml
test: non-binary
!binary dGVzdA==: binary
joern@host2:~/projects/devfile$ ruby -ryaml -e 'x = YAML.safe_load(File.read("2.yaml"));puts x'
{"test"=>"binary"}
joern@host2:~/projects/devfile$ go run g.go 2.yaml
dGVzdA==: binary
test: non-binary

joern@host2:~/projects/devfile$
```

Yes, it really was that "easy". Having this `!binary` behavior difference in Ruby and Go we can construct a YAML file like:

```
whatever: is here
!binary parent: hehehe injected
```

Now when we look at it through the eyes of Ruby it will appear as:

```
joern@host2:~/projects/devfile$ ruby -ryaml -e 'x = YAML.safe_load(File.read("what.yaml"));puts x'
{"whatever"=>"is here", "\xA5\xAA\xDE\x9E"=>"hehehe injected"}
```

The `!binary` value has been decoded to a binary key `"\xA5\xAA\xDE\x9E"` and now, **drumroll** for the Go parser:

```
joern@host2:~/projects/devfile$ go run g.go what.yaml
parent: hehehe injected
whatever: is here
```

The `!binary` tag has just been silently dropped and the resulting YAML key is called `parent`. These two behaviors combined are exactly what we need to bypass the Ruby validation for the `parent` key in the Devfile YAML.

Writing files where they don't belong

Now that we're able to sneak the `parent` key past Ruby and into the Go code it is time to dig deeper into the devfile library and the odd file writing behavior to the working directory I noticed earlier.

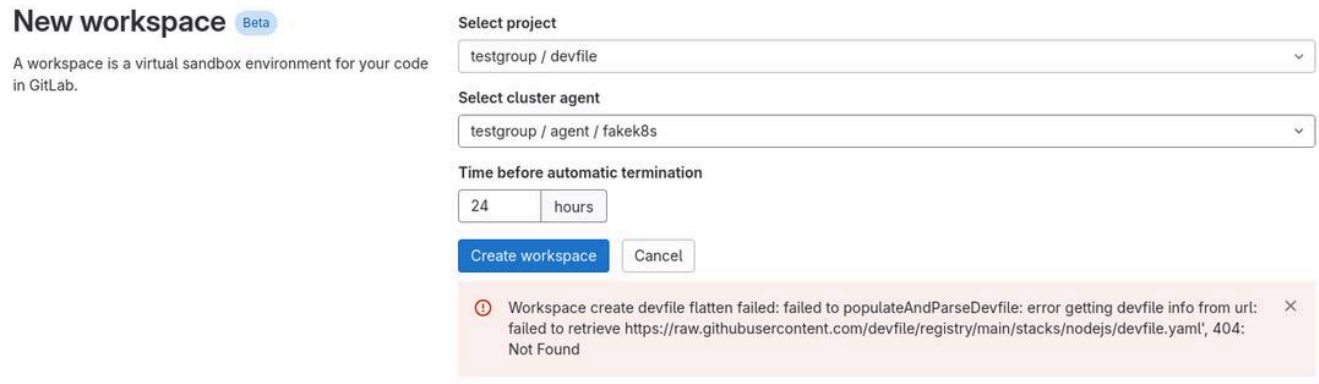
First of all, I wanted to know where that working directory was when the `devfile` binary from the Gem was called on a GitLab instance. I was hoping that it would be somewhat useful directory from an exploitation perspective.

To find this out I looked at the documentation howto set up GitLab Workspaces. There's quite a lot of requirements here to set up those Workspaces properly, a Kubernetes Cluster and a GitLab Agent for it, as well as certain configuration projects on the GitLab instance to set up and tie everything together. The TL;DR of my test setup was to run everything locally. GitLab was run using Docker and the Kubernetes side was set up with `minikube`.

With those two pieces in place we can connect the `minkube` cluster with the GitLab Agent to a project in a group on the GitLab instance. In another within the same group we can then create a `.devfile.yaml` with the following content:

```
schemaVersion: 2.2.0
!binary parent:
  uri: https://raw.githubusercontent.com/devfile/registry/main/stacks/nodejs/devfile.yaml'
```

To trigger the Devfile parsing we now just need to create a workspace for that project:



After this step I logged into the GitLab Docker container and searched for the file `stack.yaml` which was present when I parsed the same Devfile earlier when I initially observed the file writing behavior.

```
root@localhost:/# find . -name stack.yaml 2> /dev/null
./var/opt/gitlab/gitlab-rails/working/stack.yaml
root@localhost:/# cd /var/opt/gitlab/gitlab-rails/working/
root@localhost:/var/opt/gitlab/gitlab-rails/working# ls -lart
total 46
drwxr-xr-x 9 git root  12 Apr 11 12:59 ..
-rw-r--r-- 1 git git  283 Apr 12 13:09 stack.yaml
-rw-r--r-- 1 git git   73 Apr 12 13:09 OWNERS
drwxr-xr-x 2 git git    2 Apr 12 13:09 2.2.0
drwxr-xr-x 2 git git    2 Apr 12 13:09 2.1.1
drwx------ 4 git root   6 Apr 12 13:09 .
```

This result was not very promising, the directory was empty besides the files written via the Devfile parser. While there might be some race conditions with other parts of GitLab where we could write into temporary directories I decided to not go down this route and instead dig deeper into the Dev file library.

The main logic which parses the `parent` key in the Devfile was quickly identified. It starts in `parsePa rentAndPlugin()`. The name of the function already indicates another similar feature comparable to `parent`, namely the `plugin`. As both features, `parent` and `plugin` had pretty much the same underlying logic in a switch statement for `plugin` and for `parent`:

```
switch {
case parent.Uri != "":
    parentDevfileObj, err = parseFromURI(parent.ImportReference, d.Ctx, resolveCtx, tool)
case parent.Id != "":
    parentDevfileObj, err = parseFromRegistry(parent.ImportReference, resolveCtx, tool)
case parent.Kubernetes != nil:
    parentDevfileObj, err = parseFromKubeCRD(parent.ImportReference, resolveCtx, tool)
default:
    return fmt.Errorf("devfile parent does not define any resources")
}
```

I digged deeper into the `parseFrom*` methods. At first I looked into `parseFromURI`, a URI to download a Devfile from I thought, should be easy enough. Surprisingly it was not that easy. The `p arseFromURI` function had quite some logic involved about local and remote URLs. What caught my attention was:

```
if tool.downloadGitResources {
    destDir := path.Dir(curDevfileCtx.GetAbsPath())
    err = tool.devfileUtilsClient.DownloadGitRepoResources(newUri, destDir, token)
    if err != nil {
        return DevfileObj{}, err
    }
```

Litte did I know back when I examined this code about CVE-2023-49569:

>

CVE-2023-49569 A path traversal vulnerability was discovered in go-git versions prior to v5.11. This vulnerability allows an attacker to create and amend files across the filesystem. In the worse case scenario, remote code execution could be achieved.

This vulnerability was published on January 12th, and could have been very useful for my proposed attack as the Devfile library utilizes `go-git` for the underlying Git operations. However when I was looking at the Devfile library this vulnerability was not public yet. I *might* have found it if I had decided to dig deeper into the Git operations, but I didn't ;). Instead when I was realizing that a simple URL pointing to one of either `gitlab.com`, `github.com`, `raw.githubusercontent.com` or `bitbucket.org`, the Devfile library would do its magic and try to `git clone` the according repository to get the referenced files.

The relevant implementation parts are in pkg/devfile/parser/util/utils.go:

```go
// DownloadGitRepoResources downloads the git repository resources
func (c DevfileUtilsClient) DownloadGitRepoResources(url string, destDir string, token string) error {
    var returnedErr error
    if util.IsGitProviderRepo(url) {
        gitUrl, err := util.NewGitURL(url, token)
        if err != nil {
            return err
        }

        if !gitUrl.IsFile || gitUrl.Revision == "" || !ValidateDevfileExistence((gitUrl.Path)) {
            return fmt.Errorf("error getting devfile from url: failed to retrieve %s", url)
        }

        stackDir, err := os.MkdirTemp("", "git-resources")
        if err != nil {
            return fmt.Errorf("failed to create dir: %s, error: %v", stackDir, err)
        }

        defer func(path string) {
            err := os.RemoveAll(path)
            if err != nil {
                returnedErr = multierror.Append(returnedErr, err)
            }
        }(stackDir)

        gitUrl.Token = token

        err = gitUrl.CloneGitRepo(stackDir)
        if err != nil {
            returnedErr = multierror.Append(returnedErr, err)
            return returnedErr
        }

        dir := path.Dir(path.Join(stackDir, gitUrl.Path))
        err = util.CopyAllDirFiles(dir, destDir)
        if err != nil {
            returnedErr = multierror.Append(returnedErr, err)
            return returnedErr
        }
    } else {
        return fmt.Errorf("failed to download resources from parent devfile.  Unsupported Git Provider for %s ", url)
    }
```

```
    return nil
}
```

and in /pkg/util/util.go:

```
// IsGitProviderRepo checks if the url matches a repo from a supported git provider
func IsGitProviderRepo(url string) bool {
    if strings.Contains(url, RawGitHubHost) || strings.Contains(url, GitHubHost) ||
        strings.Contains(url, GitLabHost) || strings.Contains(url, BitbucketHost) {
        return true
    }
    return false
}
```

So instead of digging into the `go-git` path here I performed some simple checks using symbolic links in repositories to see if that would bring me any further. That wasn't the case, so next I started looking into `parseFromRegistry`. A registry for Devfiles is based on the Open Container Initiative (OCI) Specification and pretty much behaves like for instance a Docker registry.

Diving into `parseFromRegistry` quickly escalated as I was confronted with just another dependency of the dependency. `parseFromRegistry` calls `getResourcesFromRegistry` which itself leaves the heavy-lifting to `registryLibrary`. This library, `registry-support` is also developed by the Devfile project and I decided to take a look at it. After following the code flow, I arrived at the `PullStackFromRegistry` function, which calls the `decompress` function, which takes a `tar.gz` archive from the registry library and extracts the files inside that archive. Let's have a look at that `decompress` function:

```
// decompress extracts the archive file
func decompress(targetDir string, tarFile string, excludeFiles []string) error {
   var returnedErr error

   reader, err := os.Open(filepath.Clean(tarFile))
...
   gzReader, err := gzip.NewReader(reader)
...
   tarReader := tar.NewReader(gzReader)
   for {
...

      target := path.Join(targetDir, filepath.Clean(header.Name))
      switch header.Typeflag {
...
      case tar.TypeReg:
         /* #nosec G304 -- target is produced using path.Join which cleans the dir path */
         w, err := os.OpenFile(target, os.O_CREATE|os.O_RDWR, os.FileMode(header.Mode))
         if err != nil {
            returnedErr = multierror.Append(returnedErr, err)
            return returnedErr
         }
         /* #nosec G110 -- starter projects are vetted before they are added to a registry.  Their contents can be seen before th
         _, err = io.Copy(w, tarReader)
         if err != nil {
            returnedErr = multierror.Append(returnedErr, err)
            return returnedErr
         }
         err = w.Close()
         if err != nil {
            returnedErr = multierror.Append(returnedErr, err)
            return returnedErr
         }
      default:
         log.Printf("Unsupported type: %v", header.Typeflag)
      }
   }

   return nil
}
```

This looked promising: the line

```
target := path.Join(targetDir, filepath.Clean(header.Name))
```

followed by:

```
/* #nosec G304 -- target is produced using path.Join which cleans the dir path */
w, err := os.OpenFile(target, os.O_CREATE|os.O_RDWR, os.FileMode(header.Mode))
```

The gosec rule 304, `File path provided as taint input`, had alerted here and the developer had instructed the gosec scanner to ignore the finding. The comment even gives us the reasoning for this: `target is produced using path.Join which cleans the dir path`, which references how the `filepath.Clean(header.Name)` is used a little earlier in the code flow.

However `filepath.Clean` does not work as expected here. It's not really obvious from the documentation either, but when supplying a relative path to `filepath.Clean` the `Clean()`ed path will stay relative.

Consider the following example:

```
package main

import (
  "fmt"
  "path/filepath"
)

func main() {
  fmt.Println(filepath.Clean("/../../../../../../../tmp/test")) // absolute path
  fmt.Println(filepath.Clean("../../../../../../../tmp/test"))  // relative path
}
```

The output of this program is:

```
/tmp/test
../../../../../../../tmp/test
```

And this was the missing puzzle piece for a successful exploit. Tar files can contain `/`es and `.`s in their entry names. So we can traverse out of the intended directory and decompress and write files to arbitrary locations on disk when including a `parent` from a registry in the Devfile.

After this path traversal was identified a lot of time actually went into setting up a fake registry server and delivering a proper payload. In total this vulnerability, from starting to look into the Devfile Gem to having the a working exploit ready took about two working days where a lot of time was spent in setting up both the Workspaces feature and the fake registry.

Conclusions

There are several takeaways more or less hidden between the lines in this post I would like to highlight here.

Parser differentials

Parser differentials can be a very powerful tool when it comes to exploitation. They are very context dependent though and hard to generalize in their use for exploiting software.

For once the SAST scanner was right. The path traversal was not stopped by `filepath.Clean`, but the comment's author thought it was. They explicitly turned off the gosec warning. The whole point in software exploitation is to let some software do what it wasn't intended to do by the authors. This means when reading comments in source code they should be taken as an inspiration to think `how can I falsify this comment?`.

`../` keeps on giving

The character sequence `../` is really a gift which keeps on giving. Path traversals most of the time are simple and reliably to exploit. They've been around 30+ years, still this vulnerability class has not yet been solved.

Can't find all the bugs

The go-git vulnerability (CVE-2023-49569) was disclosed only a few days after I internally reported the file write issue based on the registry parser. This vulnerability used in combination with the parser differential would have been another way to write files where they don't belong. The message here is kind of two fold: while it might not be possible to find all the bugs there's often enough bugs to reach your goal in a sufficiently big code base. ;)

Keep digging everyone

Finally, I'd like to highlight that to find vulnerabilities it's always worth digging into source code, reading it and trying to understand the assumptions under which it has been developed. The real hard part is to "know" where to look and when to stop looking.