CASE STUDY

Capsicum Case Study: Got

BY YANG ZHONG

I've been working with Capsicum for some time as part of my internship with the FreeBSD Foundation. This article details my process of applying the Capsicum sandboxing framework to a large program called Got. Along the way, I'll give a simple and concrete introduction to Capsicum: what problems it deals with, the reasoning behind its solutions, and how to use it. We will find that Got is particularly well-suited to Capsicum, and I'll discuss how Got's structures make the program Capsicum-friendly.

Capsicum Concepts

Capsicum exists to fix a straightforward problem with computer programs: they have too much power. I like to think of it like this. In a world without Capsicum, if I log in to my computer and run some program, *there is nothing stopping that program from deleting my entire home directory without warning*. Of course the program probably will never do that intentionally, but it has enough power to do so. This becomes a real concern when thinking about security: if someone finds a vulnerability in a program, they could exploit it to do anything the program can do. Therefore, if the program is wielding an excessive amount of power, the attacker can use it to do an excessive amount of damage.

In comes Capsicum[1], which provides tools to control a program's power. One important concept Capsicum deals with is that of a global namespace. Essentially, a *global namespace* is a group ("space") of objects, each of which have an identifier ("name") that uniquely identifies it among all objects ("global"). An easy example of a global namespace is the file system: the space is the group of all files, and each file's absolute path is its unique 'name'. The FreeBSD operating system has many global namespaces[2], but the file system is ubiquitous and very important; I'll be talking about it a lot from now on.

Capsicum-less programs deal frequently with global namespaces. When these programs want to `open()` a file, they can pass in a file's absolute path. While this seems normal enough, the program is actually using its tremendous power here: "Out of every single file on this computer, I want this one!" While file permissions and such will prevent the program from accessing every file, this amount of power is certainly in the 'delete-your-whole-home-directory' range.

Even worse, when the program is exercising its power like this, it's exercising the power implicitly. It does not say: "I would like to exercise my power to access every file, and here is a 'key' to verify that I have this power"; the program just always has the power. This power to do things 'by default' is called *ambient authority*[3]. When accessing these global namespaces, programs are always exercising their ambient authority.

Therefore, Capsicum recognizes global namespaces as basically an uncontrollable source of power, and so introduces capability mode — a state in which a program cannot use glob-

al namespaces at all, and therefore also have no ambient authority. Programs 'enter' *capability mode* by calling `cap_enter()`[4], and cannot ever leave. In capability mode, a program cannot `open()` new files, and is therefore restricted to file descriptors that it `open()`-ed before calling `cap_enter()`, or file descriptors provided through a connection to a different process. This environment that limits the resources a program has access to is called a *sandbox*, which is why Capsicum is described as a *sandboxing technique*.

Capsicum does more than this. Another important concept is that of capabilities, which in Capsicum's case are objects that extend file descriptors; they let you finely limit what any file descriptor is capable of doing, and serve the more hidden role of making capability mode actually work. There is also the Casper library, which provide common services for Capsicumized programs.

The Target: Got

So, with these tools, we'd like to adapt some existing programs to use them. I was tasked with adapting to Capsicum the version-control system Game of Trees, or Got for short. It's being developed by and for OpenBSD developers, but the FreeBSD project is considering adding Got to the base system. So as part of this effort, it was my job to figure out how to tweak Got to be more amenable to Capsicum, without drastically changing anything: Ideally, we would make structural changes clean enough to incorporate into the upstream version of Got, so that the FreeBSD Capsicum version of Got is as similar to it as possible.

Capsicumizing Got

On a high level, a common pattern for Capsicumized programs has the program be separated into two parts. In the first part, the program acquires the resources it needs; in the second, the program does its 'work' of reading from and writing to those resources. The program enters capability mode right after the first part. Since the program is stuck in capability mode after this, its power is limited in the dangerous second part, in which it works with the external and untrustworthy resources it acquired in the first.

Many programs are not separated in this way. Often, they acquire resources wherever it's convenient, resulting in the two parts being delicately interleaved. Before Got, I dealt with this issue in the program sort(). In these situations, helper libraries like Casper are invaluable, as they exist to solve common Capsicumization problems that would otherwise take a lot of set-up work to fix. You can see an simple example of the interleaving issue in *Case studies of sandboxing base system with Capsicum, by Mariusz Zaborski (EuroBSDcon 2017)* on Youtube, part of which describes the process of Capsicumizing the program `bspatch()`.

Fortunately for me, Got is structured in how it gets its files. Got works with two main directories: a repository and a worktree. If you know Git, these are quite similar to Git repositories and worktrees[5]. Got then has the functions `got_repository_open()` and `got_worktree_open()`, responsible for looking for the repository/worktree and returning a `struct - struct got_repo` and `struct got_worktree` respectively — containing information about the two directories[6].

After this point, Got exclusively works within these two directories (and /tmp), which means that it never tries to acquire anything 'new'. This avoids the interleaving problem discussed earlier, but Got still uses the global file system namespace to actually open new files — for example, the `got_repo` struct contains the absolute path to its associated repository, and so Got would open the directory using that path whenever it needs to. This is not compatible with ca-

pability mode.

In that case, must I pre-open every single file inside the two directories, so that I can use them in capability mode? Thankfully not. When you `open()` file, you get its file decriptor. For non-directory files, its descriptor lets you access just that file. However, a directory's file descriptor allows you to access everything inside that directory.

For this purpose, FreeBSD, by way of POSIX, provides the `*at()-family` of system calls. Where the normal calls take in absolute paths, the `*at()` calls take in a file descriptor and a relative path. If I wish to open the file "`/dir/subdir/a`", and I have a file descriptor `fd` for dir, I can call `openat(fd, "subdir/a")`. This form of access is allowed in capability mode, barring some exceptions[7], since we are no longer searching through the global namespace of all files.

It's easy to see how this helps us with Got, as we know that Got will always work within two specific directories! If we pre-open the repository and worktree directories and store their file descriptors inside the `got_repo` and `got_worktree` structs, we can later use those descriptors to open files inside those directories, even in capability mode. In Got, functions that operate on files inside the repository or worktree will take in a `got_repository` or `got_worktree` as a parameter, meaning that the file descriptor we added will be easily accessible there.

```
static const struct got_error
update_blob(struct got_worktree *worktree,
    struct got_fileindex *fileindex, struct got_fileindex_entry *ie,
    struct got_tree_entry *te, const char *path,
    struct got_repository *repo, got_worktree_checkout_cb progress_cb,
    void *progress_arg)
{
    const struct got_error *err = NULL;
    struct got_blob_object *blob = NULL;
    char *ondisk_path;
    unsigned char status = GOT_STATUS_NO_CHANGE;

    truct stat sb;
    if (asprintf(&ondisk_path, "%s/%s", worktree->root_path, path) == -1)
    return got_error_from_errno("asprintf");

    // example of usage
    int opened_file_fd = open(ondisk_path, 0);
```

The above snippet of Got's code shows a function that takes in a `got_worktree struct`, and uses it to construct a path to a file in that directory. I've added an example of how the function would typically use the new path.

Below is the same code, converted to use our new file descriptor strategy.

```
static const struct got_error
update_blob(struct got_worktree *worktree,
    struct got_fileindex *fileindex, struct got_fileindex_entry *ie,
    struct got_tree_entry *te, const char *path,
    struct got_repository *repo, got_worktree_checkout_cb progress_cb,
```

```
    void *progress_arg)
{
    const struct got_error *err = NULL;
    struct got_blob_object *blob = NULL;
    char *ondisk_path;
    unsigned char status = GOT_STATUS_NO_CHANGE;

    struct stat sb;
    int path_fd_part = worktree->root_fd;
    char *path_relative_part = path;

    // example of usage
    int opened_file_fd = openat(path_fd_part, path_relative_part, 0)
```

It's quite simple! Simpler than the first one, even, since the asprintf() call is no longer needed. In these types of situations, adapting Got to support Capsicum is easy.

Some functions don't take in these structs, but instead take in an absolute path that they operate on. Adapting these functions to be Capsicum-compatible takes more work, as we must change their parameters from an absolute path to a pair of (relative path, directory file descriptor), in order for the function to be able to access the file in capability mode.

In practice, this is usually only a small problem. The absolute path the function takes in doesn't come from nowhere — it must have been created by using the `got_repo` or `got_worktree structs`, and therefore the file descriptor we need won't be far away. Below is a function whose parameters needed to be changed as a part of Capsicumization:

```
const struct got_error *
got_fileindex_entry_update(struct got_fileindex_entry *ie,
-    const char *ondisk_path, uint8_t *blob_sha1, uint8_t *commit_sha1,
-    int update_timestamps)
+    int wt_fd, const char *ondisk_path, uint8_t *blob_sha1,
+    uint8_t *commit_sha1, int update_timestamps)
{
    struct stat sb;
-    if (lstat(ondisk_path, &sb) != 0) {
+    if (fstatat(wt_fd, ondisk_path, &sb, AT_SYMLINK_NOFOLLOW) != 0) {
        if (!((ie->flags & GOT_FILEIDX_F_NO_FILE_ON_DISK) &&
        errno == ENOENT))
-            return got_error_from_errno2("lstat", ondisk_path);
+            return got_error_from_errno2("fstatat", ondisk_path);
        sb.st_mode = GOT_DEFAULT_FILE_MODE;
        } else {
...
```

Since the parameters of the function changed, we also need to alter all the places where it was called. In some places, the calling function created the path, using the `got_worktree`

struct, that it passes into `got_fileindex_entry_update()`; for these, we already have the necessary file descriptor, and so adapting to the new parameters is easy:

```
...
        * Preserve the working file and change the deleted blob's
        * entry into a schedule-add entry.
        */
-       err = got_fileindex_entry_update(ie, ondisk_path, NULL, NULL,
-       0);
+       err = got_fileindex_entry_update(ie, worktree->root_fd,
+       ie->path, NULL, NULL, 0);
    } else {
...
```

Some calling functions took in the path as a parameter as well, simply passing it through to `got_fileindex_entry_update()`. For these, we must similarly change the calling function's parameters:

```
 static const struct got_error *
-sync_timestamps(char *ondisk_path, unsigned char status,
+sync_timestamps(int wt_fd, const char *path, unsigned char status,
 struct got_fileindex_entry *ie, struct stat *sb)
 {
      if (status == GOT_STATUS_NO_CHANGE && stat_info_differs(ie, sb))
-         return got_fileindex_entry_update(ie, ondisk_path,
+         return got_fileindex_entry_update(ie, wt_fd, path,
          ie->blob_sha1, ie->commit_sha1, 1);
...
```

Ultimately, the path must originate from a function that has access to `got_worktree`, and so the file descriptor can always be threaded through the calls. It's certainly not a clean solution, especially if the thread gets long, but I've yet to find a thread longer than two calls.

Wrap-up

I hope you've been convinced at this point that making Got work with capability mode is simple. While I've only committed to Got the very beginnings of the work needed, I suspect that most of the necessary changes will be similar to what you've seen.

Of course, not every program is so amenable to Capsicum. Fundamentally, capability mode works well with programs that deliberately manage their resources. Got makes its main resources — the worktree and repository directories — into structs in the code. If a function wants to operate on one of these resources, it needs the struct to do so.

In this way, the code is explicitly saying: "This function will need this resource". Additionally, since Got works with few other resources, the code is saying "This function will need this resource only". This explicitness is opposite to ambient authority, and is exactly what capability mode wants! The rest of the work lies in just enforcing these limitations.

1. ...Along with other frameworks, with similar goals but different designs, such as seccomp for Linux and pledge/unveil for OpenBSD. Much has already been written comparing these frameworks; Jonathan Anderson's "A comparison of Unix sandboxing techniques" takes a detailed look.
2. You can find a comprehensive list in "Capsicum: practical capabilities for UNIX" by Robert N.M. Watson et. al.
3. "Capability Myths Demolished" by Mark S. Miller et. al. gives a clear description of ambient authority.
4. Practically, you'll be using the 'Capsicum helpers' and calling caph_enter(), but it's essentially the same thing.
5. In fact, Got can be used with normal Git repositories, hence the similar name.
6. One of the big mistakes I made here was that I tried to enter capability mode *before* the got_worktree_open and got_repo_open functions — It did work after a lot of hacking, but it left a huge mess, and later the lead developer of Got helpfully told me that those functions weren't doing anything dangerous anyway so it's okay to call them before entering capability mode; From this I realized that it's very important to *understand the code before trying to apply Capsicum*. It sounds obvious, but I learned it the hard way.
7. The path can't be absolute, the path can't use ".." components to 'escape' out of the directory, and the file descriptor can't be AT_FDCWD.

YANG ZHONG is studying Computer Science at the University of Waterloo. He worked as an intern with the FreeBSD Foundation for the Fall 2020 term as part of the University's co-operative education program and returned for the Spring 2021 term. In his spare time he enjoys writing for *mathNEWS*, the University of Waterloo math faculty's student publication.