Building interactive SSH applications

Originalartikel

Backup

<html> After the announcement of shell access for builds.sr.ht jobs, a few people sent me some questions, wondering how this sort of thing is done. Writing interactive SSH applications is actually pretty easy, but it does require some knowledge of the pieces involved and a little bit of general Unix literacy. On the server, there are three steps which you can meddle with using OpenSSH: authentication, the shell session, and the command. The shell is pretty easily manipulated. For example, if you set the user’s login shell to

```
/usr/bin/nethack
```

, then nethack will run when they log in. Editing this is pretty straightforward, just pop open

```
/etc/passwd
```

as root and set their shell to your desired binary. If the user SSHes into your server with a TTY allocated (which is done by default), then you’ll be able to run a curses application or something interactive. <noscript readability=„3“> This article includes third-party JavaScript content from asciinema.org, a free- and open-source platform that I trust. </noscript> However, a downside to this is that, if you choose a “shell” which does not behave like a shell, it will break when the user passes additional command line arguments, such as

```
ssh user@host ls -a
```

. To address this, instead of overriding the shell, we can override the command which is run. The best place to do this is in the user’s

```
authorized_keys
```

file. Before each line, you can add options which apply to users who log in with that key. One of these options is the “command” option. If you add this to

```
/home/user/.ssh/authorized_keys
```

instead: <div class=„highlighter-rouge highlight“ readability=„6“> <pre>command=„/usr/bin/nethack“ ssh-rsa … user </pre> Then it’ll use the user’s shell (which should probably be

```
/bin/sh
```

) to run

```
nethack
```

, which will work regardless of the command supplied by the user (which is stored into

```
SSH_ORIGINAL_COMMAND
```

in the environment, should you need it). There are probably some other options you want to set here, as well, for security reasons: <div class=„highlighter-rouge highlight“ readability=„8“> <pre>restrict,pty,command=„…“ ssh-rsa … user </pre> The full list of options you can set here is available in the

```
sshd(8)
```

man page.

```
restrict
```

just turns off most stuff by default, and

```
pty
```

explicitly re-enables TTY allocation, so that we can do things like curses. This will work if you want to explicitly authorize specific people, one at a time, in your

```
authorized_keys
```

file, to use your SSH-driven application. However, there’s one more place where we can meddle: the

```
AuthorizedKeysCommand
```

in

```
/etc/ssh/sshd_config
```

. Instead of having OpenSSH read from the

```
authorized_keys
```

file in the user’s home directory, it can execute an arbitrary program and read the

```
authorized_keys
```

file from its stdout. For example, on Sourcehut we use something like this: <div class=„highlighter-rouge highlight“ readability=„6“> <pre>AuthorizedKeysCommand /usr/bin/gitsrht-dispatch „%u“ „%h“ „%t“ „%k“ AuthorizedKeysUser root </pre> Respectively, these format strings will supply the command with the username attempting login, the user’s home directory, the type of key in use (e.g.

```
ssh-rsa
```

), and the base64-encoded key itself. More options are available - see

```
TOKENS
```

, in the

```
sshd_config(8)
```

man page. The key supplied here can be used to identify the user - on Sourcehut we look up their SSH key in the database. Then you can choose whether or not to admit the user based on any logic of your choosing, and print an appropriate

```
authorized_keys
```

to stdout. You can also take this opportunity to forward this information along to the command that gets executed, by appending them to the command option or by using the environment options. We use a somewhat complex system for incoming SSH connections, which I won’t go into here - it’s only necessary to support multiple SSH applications on the same server, like git.sr.ht and builds.sr.ht. For builds.sr.ht, we accept all connections and authenticate later on. This means our AuthorizedKeysCommand is quite simple: <div class=„language-python highlighter-rouge highlight“ readability=„12“> <pre>#!/usr/bin/env python3 # We just let everyone in at this stage, authentication is done later on. import sys key_type = sys.argv[3] b64key = sys.argv[4] keys = (f„command=\“buildsrht-shell '{b64key}'\„,restrict,pty “ +

```
  f"{key_type} {b64key} somebody\n")
```

print(keys) sys.exit(0) </pre> The command,

```
buildsrht-shell
```

, does some more interesting stuff. First, the user is told to connect with a command like

```
ssh builds@buildhost connect &lt;job ID&gt;
```

, so we use the

```
SSH_ORIGINAL_COMMAND
```

variable to grab the command line they included: <div class=„language-python highlighter-rouge highlight“ readability=„10“> <pre>cmd = os.environ.get(„SSH_ORIGINAL_COMMAND“) or „“ cmd = shlex.split(cmd) if len(cmd) != 2:

```
  fail("Usage: ssh ... connect &lt;job ID&gt;")
```

op = cmd[0] if op not in [„connect“, „tail“]:

```
  fail("Usage: ssh ... connect &lt;job ID&gt;")
```

job_id = int(cmd[1]) </pre> Then we do some authentication, fetching the job info from the local job runner and checking their key against meta.sr.ht (the authentication service). <div class=„language-python highlighter-rouge highlight“ readability=„13“> <pre>b64key = sys.argv[1] def get_info(job_id):

```
  r = requests.get(f"http://localhost:8080/job/{job_id}/info")
  if r.status_code != 200:
      return None
  return r.json()
```

info = get_info(job_id) if not info:

```
  fail("No such job found.")
```

meta_origin = get_origin(„meta.sr.ht“) r = requests.get(f„{meta_origin}/api/ssh-key/{b64key}“) if r.status_code == 200:

```
  username = r.json()["owner"]["name"]
```

elif r.status_code == 404:

```
  fail("We don't recognize your SSH key. Make sure you've added it to " +
      f"your account.\n{get_origin('meta.sr.ht', external=True)}/keys")
```

else:

```
  fail("Temporary authentication failure. Try again later.")
```

if username != info[„username“]:

```
  fail("You are not permitted to connect to this job.")
```

</pre> There are two modes from here on out: connecting and tailing. The former logs into the local build VM, and the latter prints the logs to the terminal. Connecting looks like this: <div class=„language-python highlighter-rouge highlight“ readability=„26“> <pre>def connect(job_id, info):

```
  """Opens a shell on the build VM"""
  limit = naturaltime(datetime.utcnow() - deadline)
  print(f"Your VM will be terminated {limit}, or when you log out.")
  print()
  requests.post(f"http://localhost:8080/job/{job_id}/claim")
  sys.stdout.flush()
  sys.stderr.flush()
  tty = os.open("/dev/tty", os.O_RDWR)
  os.dup2(0, tty)
  subprocess.call([
      "ssh", "-qt",
```

```
      "-p", str(info["port"]),
      "-o", "UserKnownHostsFile=/dev/null",
      "-o", "StrictHostKeyChecking=no",
      "-o", "LogLevel=quiet",
      "build@localhost", "bash"
  ])
  requests.post(f"http://localhost:8080/job/{job_id}/terminate")
```

</pre> This is pretty self explanatory, except perhaps for the dup2 - we just open

```
/dev/tty
```

and make

```
stdin
```

a copy of it. Some interactive applications misbehave if stdin is not a tty, and this mimics the normal behavior of SSH. Then we log into the build VM over SSH, which with stdin/stdout/stderr rigged up like so will allow the user to interact with the build VM. After that completes, we terminate the VM. This is mostly plumbing work that just serves to get the user from point A to point B. The tail functionality is more application-like: <div class=„language-python highlighter-rouge highlight“ readability=„24“> <pre>def tail(job_id, info):

```
  """Tails the build logs to stdout"""
  logs = os.path.join(cfg("builds.sr.ht::worker", "buildlogs"), str(job_id))
  p = subprocess.Popen(["tail", "-f", os.path.join(logs, "log")])
  tasks = set()
  procs = [p]
  # holy bejeezus this is hacky
  while True:
      for task in manifest.tasks:
          if task.name in tasks:
              continue
          path = os.path.join(logs, task.name, "log")
          if os.path.exists(path):
              procs.append(subprocess.Popen(
                  f"tail -f {shlex.quote(path)} | " +
                  "awk '{ print \"[" + shlex.quote(task.name) + "] \" $0
}'",
                  shell=True))
              tasks.update({ task.name })
      info = get_info(job_id)
      if not info:
          break
      if info["task"] == info["tasks"]:
          for p in procs:
              p.kill()
          break
      time.sleep(3)
```

if op == „connect“:

```
  if info["task"] != info["tasks"] and info["status"] == "running":
      tail(job_id, info)
  connect(job_id, info)
```

elif op == „tail“:

```
  tail(job_id, info)
```

</pre> This… I… let’s just pretend you never saw this. And that’s how SSH access to builds.sr.ht works! Have a comment on one of my posts? Start a discussion in my public inbox by sending an email to ~sircmpwn/public-inbox@lists.sr.ht <small class=„c1“>[mailing list etiquette]</small> <hr/><section class=„webring“ readability=„12“><h3>Articles from blogs I follow around the net</h3> <section class=„articles“ readability=„27“> <h4 class=„title“>Go 1.13 is released</h4> Today the Go team is very happy to announce the release of Go 1.13. You can get it from the download page. <small class=„source“>via The Go Programming Language Blog</small>   <h4 class=„title“>Updates in July 2019</h4> This post gives an overview of the recent updates to the Writing an OS in Rust blog and the used libraries and tools. Since I'm still very busy with my master thesis, I haven't had the time to work on a new post. But there were quite a few maintena… <small class=„source“>via Writing an OS in Rust</small>   <h4 class=„title“>What is RISC?</h4> This is an archive of a series of comp.arch USENET posts by John Mashey in the early to mid 90s, on the defnition of reduced instruction set computer (RISC). Contrary to popular belief, RISC isn't about the number of instructions! This is archived her… <small class=„source“>via Dan Luu</small> </section> Generated by openring </section> </html>