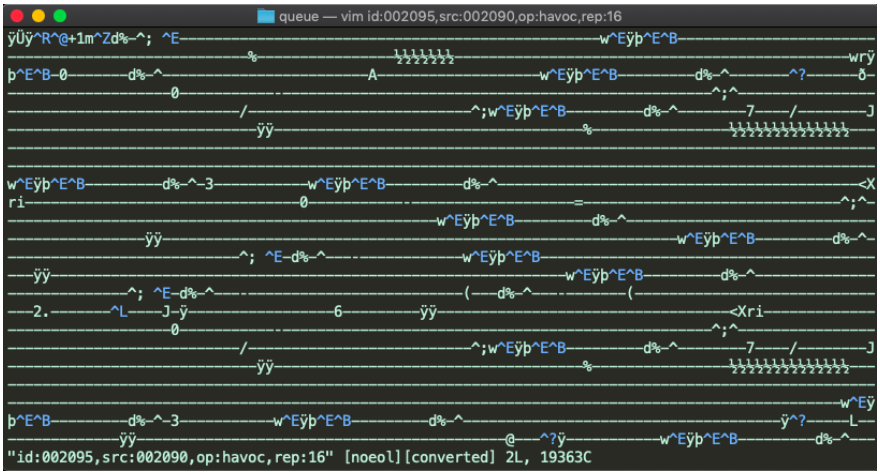
The Grand Fuzzing Challenge: A DevOps Retrospective.

Adam Hasselberg Follow

Draft · 13 min read



A Fuzzed Test-Case.

Recently the first ever Grand Fuzzing Challenge was held as part of a University course in DevOps and Automated Testing. We were 14 teams with either one or two students each, and the time from opening to the deadline was just about one month. Fuzzing is a method of testing software by generating random input and looking for faulty behavior. The Challenge was to create test-cases for maximum test-coverage of 20 of the Ubuntu Core Utilities (cd, pwd, cat, dd, etc.), *by any means necessary*.

At first glance of the name, and even of the rules, it is clear to anyone how it relates to Automated Testing. But tackling the Grand Fuzzing Challenge required a diverse skill set: creating Docker images, running automated fuzzing on Google Cloud/Kubernetes, making programs co-operate, and plenty of shell scripting.

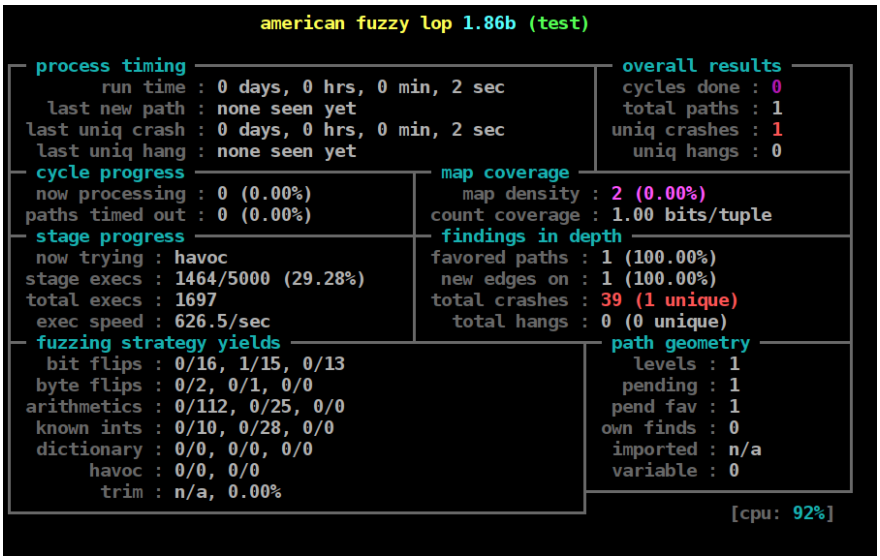
DevOps is abstract, it's often described in terms of *tools, people, processes, mindset,* and *delivery pipelines* to me it's about *bridging the gap between development and operations*. I think a useful understanding of DevOps is that it's a *doctrine* which encompasses both *tactics* and *strategies*. Tactics are the tools and specific methodologies used, such as Infrastructure as Code and Continuous Integration. Clearly, the tools aspect of DevOps was tested in The Challenge, but to win a competition

you need more than tactics; you need a strategy. This retrospective will share my story of participating and analyze the competition: what lessons I learned both tactical and strategical.

The Grand Fuzzing Challenge

The rules of the challenge are simple: Fuzz 20 of the Core Utilities (coreutils) by any means necessary. The submission should be a folder containing one text file per util, where each line is a string of arguments used to launch the util, a test case. The judge would then allow for the test cases to run sequentially for a maximum of one hour, the team achieving the highest test coverage would win.

The competition was opened during a lecture, introducing everyone in the room to what fuzzing actually is, and what is going to be fuzzed. The professor briefly explained the history of fuzzing, talked about the absolute top tier of fuzzing software, American Fuzzy Lop (AFL), and gave little to no hints on how to fuzz.



AFL's hip, retro-style UI.

Playing to Learn

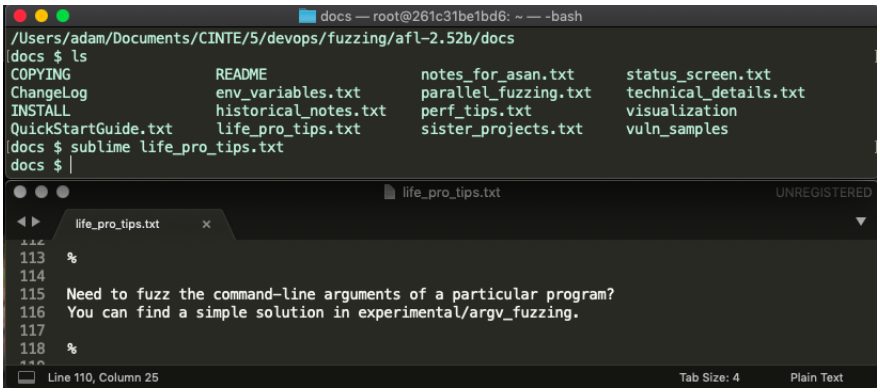
From the very beginning of learning of the challenge, I had low hopes of winning the competition. I know that I'm a competent coder but I also knew, from the discussions that had taken place in the classroom, that the other students had much more experience in software testing than I. But that didn't discourage me from trying to compete, after all, I'm a competitive person who enjoys a challenge, especially when it's a challenge I can learn from.

Going from no knowledge of fuzzing to some knowledge of fuzzing is impossible without stumbling over mentions of AFL several times. The Wikipedia entry for fuzzing mentions AFL four times, and it is at the top of the ”See also” list. I decided that AFL was going to be my choice of weapon for the challenge, even if I fail at winning I’ll at least learn something about a popular piece of software. Luckily my teammate had come to the same conclusion and shared my attitude of learning something interesting. We both had the idea that fuzzing in the cloud would be preferable to running on our own machines, and he was as happy as I was to use the challenge as an excuse to familiarize ourselves with Kubernetes as well. Thus we had *an agreed upon goal* and *a preliminary plan*: Run one instance of AFL per pod, one pod per coreutil.

Arg-v Fuzzing with AFL

Due to the tight schedule of the course and our other obligations outside of the course we couldn’t get started with any actual work until there was less than two weeks to deadline, with Easter holidays in the middle. On our first day of working on the project we were quickly able to make AFL run on a recompiled binary following the basic AFL instructions, but we found that AFL made progress for about 1 minute and then stopped discovering any new paths. We were able to see that it did run several different test cases which we knew should trigger different paths in the program but still it didn’t make any progress regardless of how many initial test-cases we added.

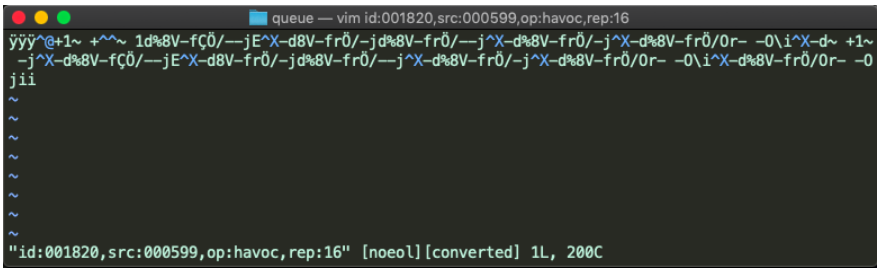
AFL feeds its test-cases to specified binaries through standard input. But the competition is not about submitting test-cases which are going to be fed into standard input. This means that when AFL is trying to feed our test-cases into a coreutil it is not launching the coreutil with our input as arguments, it’s launching the util with zero arguments. Reading the AFL-documentation thoroughly we found a life_pro_tip mentioning the experimental feature called argv-fuzzing. It’s a .c file, which when included in the source code of a c program together with one line of code makes the program read its arguments from standard-input rather than from it’s launching argument string. Thereby allowing AFL to fuzz arguments of programs, exactly what we needed to do



Fine-combing the docs folder

At this point, we had spent the entire afternoon trying to make AFL fuzz our chosen coreutil, date, and we finally saw it making progress. We decided that we would let it run over-night and if it was able to achieve decent coverage we knew that we would have a plan to achieve coverage on all the other utils. Throughout the troubleshooting and discussions about the cryptic coverage reports AFL told us it was achieved through its sleek retro design we were both in agreement that we didn't actually have any clue what map coverage meant or what would be a good percentage.

The next day I got a text from my teammate that he'd been able to run AFL-cov and gotten a decent measurement of our test-run on date. We had already prepared dictionaries and inputs for all the utils, so all we had to do was instrument them with the argument fuzzing and deploy the Kubernetes nodes.



A test case for **date**.

Interpreting Outputs

Continuing my day after working on the project I realized that the experimental argv fuzzing was going to be a problem. The output files looked nothing like what we should submit, and they were not even recognizable as valid parameter inputs for the utils. AFL does fuzzing by, among other things, flipping bits. The output of fuzzing is one file per test case, where the content of the file is raw unencoded binary

data. How can I convert the raw un-encoded data to strings? I spent the rest of the weekend thinking about the problem and trying to understand what I was missing. I knew that we needed a way to compile our outputs to the submission folder, and ideally a way to test run the compiled outputs. When Monday came around my spontaneous weekend googling had not yielded any results for how to compile outputs to something which we could submit.

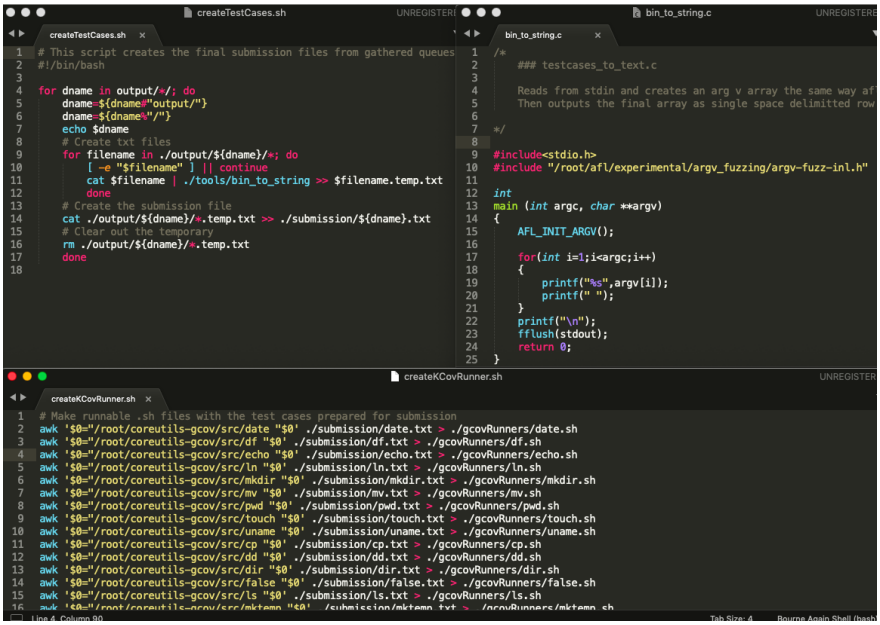
After many hours of investigating how I can get the AFL output to a workable format, I realized I needed some clarification. I submitted an issue to the course GitHub repo, asking for clarification on exactly what should be handed in, and quickly received the answer that the files should be UTF-8 formatted. It was the answer I feared since I had already tried to search for a way to convert binary files to UTF-8 and I had come to the conclusion that it's **literally an impossible task**. We can't just force any arbitrary byte to be interpreted as UTF-8.

A Fool-Proof Plan

I realized that the argv-fuzzing file we used to instrument the coreutils for AFL was something we could use. I didn't know exactly what it did beyond splitting the output file up by null byte instead of space, the way an argument string is typically split. I decided to try to write my own C program that could be fed a binary file through standard-input and then include the argv-fuzzing method from AFL's experimental feature to convert the binary file to argv parameters, and then print the parameters to standard output with space in between. I figured that would be a foolproof plan, as the arguments would then be printed exactly the way our coreutils had interpreted the arguments when the test-case was created by AFL. The program was appropriately named bin_to_string, and since my C skills are lacking I kept it simple and did most of the work by piping the program in the command line and leveraging the included code.

```
1  #include<stdio.h>
2  #include "/root/afl/experimental/argv_fuzzing/argv-fuz
3  int main (int argc, char **argv) {
4      AFL_INIT_ARGV();
5      for(int i=1;i<argc;i++) {
6          printf("%s",argv[i]);
7          printf(" ");
8      }
9      printf("\n");
```

With a few more shell scripts I could easily create submission files from all the output directories, and they got somewhat more readable. Test running the scripts were still a problem though. Generating a runnable shell-script from the submission files was easy, but I still didn't know how to measure coverage of running the scripts.



From AFL Output to Runnable Test-Cases in One-screenshot.

Good Enough is Better Than Nothing

After having spent almost two full days mainlining raw binary data and still not really understanding what I'm doing, I only had two full days of work left to wrap my head around how to get coverage reports *without* AFL-cov before the deadline. It was a wednesday before the easter-holidays, which we'd agreed to take time off for, the tuesday after would be our last day. We had two choices, either we try to get 3rd party coverage/simulation of the competition run going, or we try to get better coverage by tweaking our AFL inputs/dictionaries so we can let them run over the holidays.

We decided to go for the coverage measurement rather than the coverage chase, we recognized that handing in a submission which won't run is worse than handing in a submission with low coverage. By the end of the day I hadn't made much progress but decided that we had a nice enough toolkit that we would easily be able create the submission files and would at least be able to test-run them without coverage reports. It felt like a failure, I had gotten stuck on this issue for way longer than I had wanted to and it had eaten up all the time I would've preferred spending on actually improving our coverage.

Manually Achieving Coverage

When tuesday came around, the last day, me and my teammate met up early and were well rested after the holidays. We got to work and he told me that he'd been able to use gcov successfully. I don't know how or why we had both missed reading up on that tool from the get go but as soon as we got started with using it I realized that we should've been using it from day 1. Thanks to the scripts I'd been working on we were able to get decent coverage reports within less than two hours of working. This was the first time we saw what our actual score was going to be.

With gcov we were able to test run our submission, then test run any parameter we would like and see if we got better line coverage. Our submissions had varying results, some were very high from AFL fuzzing, while some were very low. The first thing we tried to do was to run all the coreutils with—help and—version and see if those would improve our coverage. They did so we knew that we could just add them to the output. There was a hard-limit we knew that we should pass of an average of 50% line coverage of the twenty coreutils. We decided to give manual fuzzing a try for all utils which were below 50%. We spent several hours reading through man-pages, checking lcov reports for what lines we were missing and trying different input files as arguments.

Checking a manually written test-case for **dd**.

By the end of the day we were above 50% on all utils and we were happy to know that we'd at least achieved a decent competitive entry, even though we had spent the last day doing, which we realized in hindsight, what we should've been doing on the first day.

The Other Teams

The next lecture of the course was the day when the results would be presented. Before the results were presented, each team was given a chance to present their method. Before the presentations began I was

able to talk to the other teams and I realized almost all had tried to use AFL but failed, either due to not getting passed the argv-fuzzing, or due to not being able to convert the output. We presented first and got a lot of interest to my solution to convert the output. After us, the other teams presented their solution and I was surprised that no one seemed particularly happy with their results.

Of all the teams who submitted test-cases from AFL, I believe ours was the only one that was able to run properly. The other teams who didn't get disqualified had all changed their strategy when they realized that AFL didn't do argument passing out-of-the-box, or when they saw the outputs of AFL fuzzing. **It turned out that our submission, which was essentially a one-shot fuzzing approach, not iterating over multiple versions of inputs or dictionaries won.** Simply completing the pipeline from the most popular fuzzing software, combined with some manual labor to fill the gaps was enough. We had landed at a whopping 75% coverage, where the second and third place had landed at about 70% and 65% respectively.

. . .

Lessons Learned

Complete the Pipeline

My number one lesson that I got out of this experience was the importance of considering the full path from input to submission. The competition is decided by a single submission, so to win the competition you don't need necessarily mean a pipeline for continuous preparation, but it turned out to be very useful. The other teams who put too much effort into getting good results before completing the full pipeline ended up at the bottom. Other teams spent more time on getting the pipeline working but were unable to get results. We were able to snag the win by being the only team to successfully complete the pipeline from AFL to submission, with the ability to test-run our own submission and fill the gaps. The other teams got too focused on optimizing their fuzzing or spent their effort on finding a different way to fuzz than using AFL.

AFL is a Powerful Tool

Although I haven't had a chance to analyze the difference between the results of the submissions in detail, I don't think it was a coincidence that the only team who managed to use AFL won. Even though we

barely knew what we were doing when we started fuzzing with AFL, just going by the most obvious and simple plan possible took us really far. If we had started off by doing the manual coverage inspection of different commands I believe we could have gotten a much higher coverage. Using dictionaries works just fine for this purpose. We didn't get a chance to experiment with different inputs and comparing what they yield. The author of the program recommends small inputs to start from, but interpreting small is hard. If I were to go at fuzzing again I would spend the manual labour of achieving high line coverage first, then start fuzzing using the high-line coverage corpus as input.

Kubernetes Makes Orchestration Easy

My teammate did a lot of the heavy lifting with getting us started with a Kubernetes engine to work on. Using Google Cloud we could get a cluster up and running, and create nodes on the fly within an hour. Interacting through bash scripts run locally was just fine and allowed us to manage our fuzzing quickly without having to spend time reading through manuals. As we were working with very limited amount of time available for the task, we couldn't develop a nice workflow. What we lacked was a decent way of collecting the output files.

Name	Status	Type	Pods	Namespace	Cluster
cat	OK	Deployment	1/1	default	fuzzing-cluster
cp	OK	Deployment	1/1	default	fuzzing-cluster
date	OK	Deployment	1/1	default	fuzzing-cluster
dd	OK	Deployment	1/1	default	fuzzing-cluster
df	OK	Deployment	1/1	default	fuzzing-cluster
dir	OK	Deployment	1/1	default	fuzzing-cluster
echo	OK	Deployment	1/1	default	fuzzing-cluster
false	OK	Deployment	1/1	default	fuzzing-cluster
ln	OK	Deployment	1/1	default	fuzzing-cluster
ls	OK	Deployment	1/1	default	fuzzing-cluster
mkdir	OK	Deployment	1/1	default	fuzzing-cluster
mktemp	OK	Deployment	1/1	default	fuzzing-cluster
mv	OK	Deployment	1/1	default	fuzzing-cluster
printf	OK	Deployment	1/1	default	fuzzing-cluster
pwd	OK	Deployment	1/1	default	fuzzing-cluster
sleep	OK	Deployment	1/1	default	fuzzing-cluster
touch	OK	Deployment	1/1	default	fuzzing-cluster
true	OK	Deployment	1/1	default	fuzzing-cluster
uname	OK	Deployment	1/1	default	fuzzing-cluster
vdir	OK	Deployment	1/1	default	fuzzing-cluster

Kubernetes Status Screen

GitHub for Storage

The thought of using github to store the output first felt wrong to me, I'm used to thinking of github as a place for your code/config, not binary files. But technically the output are test-cases and it would make sense to push them to a repo, they'll be used to run tests so surely they count as code, even if they are unreadable binary blobs? I'm not sure how a git repo could or should be used to push files from a node but I believe it would've been the easiest way to remove our own local machines from the pipeline we ended up using, where we collected the files from the nodes through shell scripts.

Rapid Development

Even before the competition was started I knew that it would take a lot of time. Fuzzing takes time in itself, you want the process running, sometimes for weeks. And getting started with several pieces of

technology, and then combining them can take time. We worked as a team and were able to complement each other's strengths and efficiently split up the work by communicating continuously on what we were doing and having the mindset of working together.

Most of all we were able to settle for good enough solutions, we didn't spend time automating processes that were easier to do manually, and we didn't spend time doing things manually that would be faster to automate. The tools we used, primarily docker and Kubernetes, are a godsend for doing rapid development. We could run experiments and set up new nodes so fast that it felt like cheating.

The drawback of this rapid development became very obvious by the chaos of my own local branch of the repo. It would be completely unsustainable to acquire as much technical debt as we did during the competition run. Since this was a one-shot competition and we didn't have any plans on continuing the project afterward, that was fine. But had the timeframe been a bit longer we would've needed more solid DevOps practices for both managing our code/scripts, docker images, Kubernetes orchestration and collecting the outputs.

. . .

Thank you for reading my story. If you want to learn more about fuzzing the coreutils you can check-out our Github repository for the project.

Shoutouts to my amazing teammate Anders Sjöbom, and the people behind the competition: the brilliant Professor Monperrus and his fantastic Ph.D. student Gluck Zhang.