ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

REPORT

EPFL SEMESTER PROJECT

Zhaori Li

25th June 2021

CHAPTER 1

MAGMA EXPANSION

1.1 INTRODUCTION

Magma is a collection of real programs and bugs meant to evaluate fuzzers in real world conditions. Fuzzers are programs that find bugs by brute-force test case generation. They are common tools for both software developers and security auditors alike. Different fuzzers employ different techniques for generating test cases and maximizing code coverage, and it's important to evaluate these techniques to identify their contribution value.

Magma serves as a benchmark for these fuzzers. It includes a large number of diverse programs and bugs. For now 7 libraries and 118 bugs in total are added to Magma, which covers image processing, file resolving, database, communication protocls and web development.

In the first part of the project, we expand magma by studying open-source libraries/programs and adding them to Magma along with their previous bugs. The main method to include a program in Magma is to first get the latest version of its code, then inspect any previous bug reports that it had, and finally forward-port (reverse fix) the bugs.

1.2 LIBSNDFILE

Libsndfile is a C library for reading and writing files containing sampled sound (such as MS Windows WAV and the Apple/SGI AIFF format) through one standard library interface. The first official release is released in 1999 and the latese release (the 31st release) is released in January 2021. For now the bugs added into Magma do not include audio processing. Therefore we choose to inspect this open-source library to make Magma evaluate fuzzers more comprehensively and objectively.

The vulnerabilies, source codes and patches for old versions are hard to find. In addition source codes usually change a lot during multiple versions, which makes it hard to port into magma. Therefore we only consider recent vulnerabilies. By recent here we mean the vulnerabilies in the last decade.

We search the CVE database[1] and find 23 vulnerabilies starting from 2015. We gather another three vulnerabilities from the github mainpage of libsndfile[2]. In total we gathered 26 vulnerabilies starting from 2011. In these 26 vulnerabilites, 12 vulnerabilies lead to Denial of Service, 8 vulnerabilies lead to overflow and 6 vulnerabilities lead to other types of bugs. All of the patches for these CVEs are gathered by github commits.

[1] https://www.cvedetails.com/product/36889/Libsndfile-Project-Libsndfile.html?vendor_id=16294
[2] https://github.com/libsndfile/libsndfile

1.3 RESULTS

Table.1.1 lists the porting status of all these 26 CVEs of libsndfile. We categorized these CVEs into three status: Ignored, Duplicated and Successful. The Ignored status indicates that we do not port this CVE into magma due to some reason which are explained in the notes column. The Duplicated status indicates that either this CVE is totally identical to another CVE or the patch for this CVE is identical to the patch for another CVE. The notes column for this kind of CVEs records its duplication. The Successful status indicates we successfully port this CVE and thus the notes column is empty.

CVE	Type	Status	Notes
CVE-2011-2696	Integer overflow	Successful	
CVE-2014-9496	Buffer read overflow	Successful	
CVE-2014-9756	Divide by zero	Successful	
CVE-2015-7805	Heap write overflow	Ignored	Unable to find the patch
CVE-2017-6892	Out-of-bounds read	Successful	
CVE-2017-7585	Stack-based buffer overflow	Successful	
CVE-2017-7586	Stack-based buffer overflow	Ignored	Too many code changes in the patch
CVE-2017-7741	Stack-based buffer overflow	Duplicated	Same patch as CVE-2017-7585
CVE-2017-7742	Stack-based buffer overflow	Duplicated	Same patch as CVE-2017-7585
CVE-2017-8361	Buffer overflow	Successful	
CVE-2017-8362	Invalid read	Duplicated	Same patch as CVE-2017-8361
CVE-2017-8363	Heap-based buffer over-read	Duplicated	Same patch as CVE-2017-8361
CVE-2017-8365	Buffer over-read	Duplicated	Same patch as CVE-2017-8361
CVE-2017-12562	Heap-based Buffer Overflow	Successful	
CVE-2017-14245	Out of bounds read	Successful	
CVE-2017-14246	Out of bounds read	Successful	
CVE-2017-14634	Divide by zero	Successful	
CVE-2017-16942	Divide by zero	Successful	
CVE-2017-17456		Duplicated	Duplicated as CVE-2017-14245
CVE-2018-13139	Stack-based buffer overflow	Successful	
CVE-2018-13419	Memory leak	Ignored	The maintainer and third parties were unable to reproduce and closed the issue.
CVE-2018-19432	NULL pointer dereference	Duplicated	Duplicated as CVE-2018-13139
CVE-2018-19661	Buffer over-read	Successful	
CVE-2018-19662	Buffer over-read	Successful	
CVE-2018-19758	Heap-based buffer over-read	Ignored	Not a complete fix and leads to CVE-2019-3832
CVE-2019-3832	Out-of-bounds read	Successful	

TABLE 1.1
Integration status of Libsndfile. Each CVE is tagged with its type and porting status. The notes column is used to explain if this CVE is not ported into magma.

There are 4 CVEs with ignored status, CVE-2015-7805, CVE-2017-7586, CVE-2018-13419 and CVE-2018-19758. For CVE-2015-7805, which does not affect the earliest version we can find, we could not find the patch for it and thus have to ignore it. The patch for CVE-2017-7586 involves too many code changes and it is thus difficult to be ported. The patch for CVE-2018-19758 does not fully fix the bug and, in addition, leads to CVE-2019-3832. Therefore we only port CVE-2019-3832 and ignore CVE-2018-19758.

For CVE-2018-13149, the maintainer and third parties were unable to reproduce the bug. No patch was made and thus we ignore it too.

In total 7 CVEs are categorized into Duplicated status. CVE-2017-17456 is completely identical to CVE-2017-14245. CVE-2017-7741, CVE-2017-7742 and CVE-2017-7585 are fixed by the same patch. CVE-2017-8361, CVE-2017-8362, CVE-2017-8363 and CVE-2017-8365 are fixed by the same patch too. CVE-2018-19432 and CVE-2018-13139 are actually different ways to trigger the same bug.

For the CVEs with Successful status, there exists some bugs which are triggered by serveral coniditions together. In such cases we generate a magma bug for each condition violated. In total we successfully port 20 additional magma bugs, which increases the original database of magma by 16.9%.

CHAPTER 2

ORCHESTRATION

2.1 INTRODUCTION

Currently magma only supports single machine and does not scale on multiple machines. If users want to parallelly launch magma, they need to manually set up the environment for each machine and assign tasks to these machines. In this part of the project we modify the magma so that it can automatically assign different tasks on different machines. In addition we expect that the orchestration is load balanced in case too many tasks running on the same machine makes parallelization meaningless.

The remaining parts of this chapter is organized as following: Section 2.2 explains why we choose docker swarm to achieve our goal. An overview of the structure docekr swarm and its load balancing algorithm is illustrated in Section 2.3. In Section 2.4 we describe some challenges when transferring to automatic orchestration and how to solve them. Results of our modificaion are presented in Section 2.5. We discuss the results in Section 2.6.

2.2 CHOICES OF TOOLS

There are multiple frameworks for clusters, Ray, Parallel Programming in Python, Spark, etc. However these frameworks are intended for parallel computing rather than managing and orchestrating docker images. Most of the frameworks do not provide support for docker. Even those frameworks which support docker only provide experimental supports[1] which are removed in the official release. In addition using these frameworks requires a lot of code changes in the original magma, which introduces a lot of overhead. Therefore we decide to use container orchestration tools.

There are two mainstream container orchestration tools today, docker swarm and kubernetes. Kubernetes is developed by the community with the intent of addressing container scalability and management needs. Docker swarm - or more accurately, swarm mode - is Docker's native support for orchestrating clusters of Docker engines. Both of them are effective solution of massive scale application deployment, implementation and management. While kubernetes provide richer functionalities like service discovery, ingress and batch execution, it is too heavyweight for individual developers to set up for simplistic apps and infrequent deployments. Our main goal is to automatically assigning tasks on different machines to parallelly running magma and most of the rich functionalities provided by kubernetes are not needed. Thus we decide to use docker swarm to for our docker images orchestration.

[1] https://docs.ray.io/en/ray-0.3.0/using-ray-and-docker-on-a-cluster.html

2.3 OVERVIEW OF DOCKER SWARM

2.3.1 STRUCTURE

A swarm cluster[1] consists of multiple Docker hosts which run in swarm mode and act as managers (to manage membership and delegation) and workers (which run swarm services). A given Docker host can be a manager, a worker, or perform both roles.

Swarm mode names such docker host as node. To deploy an application on a swarm cluster, users need to submit the corresponding service definition to a manager node and then the manager node will dispatch units of work called tasks to worker nodes. Manager nodes also perform the orchestration and cluster management functions required to maintain the desired state of the swarm. Manager nodes elect a single leader to conduct orchestration tasks.

Worker nodes will receive and execute tasks dispatched from manager nodes. By default manager nodes also run services as worker nodes. An agent runs on each worker node and reports on the tasks assigned to it. The worker node notifies the manager node of the current state of its assigned tasks so that the manager can maintain the desired state of each worker.

2.3.2 LOAD BALANCING

The load balancing algorithm of docker swarm consists of two parts: external load balancing and internal load balancing. External load balancing uses ingress load balancing to expose the services users want to make available externally to the swarm. For internal load balancing, the swarm manager distributes requests among services within the cluster based upon the DNS name of the service. In magma we do not need to balance any data flow from outside, which means the whole external load balancing and some parts of the internal load balancing are irrelevant to us. Thus we only consider how the manager node decides where to run a container.

The scheduling process of docker swarm works as following[2]: First, the manager node selects some worker nodes based on pre-defined filters. Once eligible nodes are selected, the next process is to choose one of them every time a container is started based on its ranking strategies.

There are two types of filters, those based on the properties of the node or that of the containers. Node filters are constraints such as operating system of the host, health of the node, or containerslot which limits the number of containers on a given host, stopped containers inclusive. Container filters are affinity, dependency and port. Affinity helps run new containers in proximity to existing containers using identifiers. Dependency filters are used for creating containers next to other containersthey depend on. The port filter maps containers with nodes on the matching port number on the host.

For the ranking strategies, docker swarm currently only support spread strategy. Spread strategy spread prefers nodes with less containers on them, stopped containers inclusive.

2.4 CHALLENGES

2.4.1 LOCAL IMAGE

One main challenge for transferring to automatic orchestration is the local docker image. Since we do not want to set up environment for each machine, we only build the docker image once on the manager node where we submit our task. Therefore how the other nodes get access to the local image becomes a problem. To solve this problem, we use a local docker registry. The Registry is a stateless, highly scalable server side application that stores and lets you distribute Docker images. In the swarm mode, this local

registry is viewed running on the localhost for every node. Docker will automatically sends the requests of accessing the registry to the node which is running the service.

We first create a registry service before we build any image. Then each time after building the image, we push them onto the local registry. Finally we let the service to use the image locating in the local registry instead of the loacal image so that the other node know where to find the image and can get the image from this local registry.

2.4.2 RESOURCE LIMITATIONS AND RESERVATIONS

Another problem is that by default each container's access to the host machine's CPU cycles is unlimited. For single machine the docker solve this problem by setingt various constraints to limit a given container's access to the host machine's CPU cycles. For the cluster things are little different. Not all nodes in the cluster may be equal (some nodes could have a different number of CPUs/cores). The scheduler has to take that into account. In addition resources limitation cannot prevent a node from overwhelmed.

In this case we add soft limitation and reservation for the service. Instead of specifying excatly what cpu cores are used for this container, we ask the number of cpu cores that the service needs and the maximum number of cpu cores that the service can use. The docker swarm will use these as constraints when filtering nodes in the scheduling part. If no node satisfies the requirements, the task will remain pending state until some node finish its task and release enough resources.

2.4.3 STORAGE

By default all files created inside a container are stored on a writable container layer. The data doesn't persist when that container no longer exists, and it can be difficult to get the data out of the container if another process needs it.[3]. Docker provides two options for containers to store files in the host machine, so that the files are persisted even after the container stops: volumes, and bind mounts. However in cluster, the storage is more complicated due to the uncertainty of which node will run the task. Docker itself does not deal with storage seperately for swarm mode. It runs any volume mount command on the node where the container is running. For volume option, the swarm mode will create different volume for different nodes even for the same service. For bind option, the swarm mode requires the path of the file exists on every node, which makes creating new binding directory much more complicated.

To address this problem, we use a distributed file system glusterfs. GlusterFS is a scale-out network-attached storage file system which uses ssh and tcp to communicate. We create a gluster file system before we build the docker cluster. Then we use binding mount, which directly mount the gluster file system on the host machine into the container. The docker swarm will write the result on the node where the service is running on and the Gluster file system will sync up between the nodes.

2.4.4 OTHER

There are some other trivial problems when transferring to swarm mode:

- By default a service will repeatedly start the task until it is deleted by users. So we use replicated job mode to avoid this. When a task belonging to a job exits successfully (return value 0), the task is marked as "Completed", and is not run again.
- The use of lock in the original magma is replaced by the constraints done by the swarm mode. There is no need to wait until the previous task finish (actually the swarm mode does not support waiting since cross-node communication is quite expensive). In addition the service is not automatically deleted. Therefore we need to repeatedly inspecting whether the service is finished or not. After the service finish we record the log and remove them.

```
project@ubuntu-s-2vcpu-4gb-fra1-01:~/magma/tools/captain/workdir/ar/afl/libpng/libpng_read_fuzzer/0/monitor$
AAH003_R, AAH003_T, AAH007_R, AAH007_T, AAH001_R, AAH001_T, AAH005_R, AAH005_T, AAH008_R, AAH008_T, AAH004_R, AAH004_T
4595016, 697, 24266189, 0, 4784558, 0, 4535924, 0, 3554417, 0, 9474897, 0
project@ubuntu-s-4vcpu-8gb-fra1-01:~/workdir/cache/afl/libpng/libpng_read_fuzzer/1/monitor$ cat 895
AAH003_R, AAH003_T, AAH007_R, AAH007_T, AAH001_R, AAH001_T, AAH005_R, AAH005_T, AAH008_R, AAH008_T, AAH004_R, AAH004_T
1677116, 172, 7492929, 0, 1506333, 0, 1435762, 0, 1326522, 0, 814253, 0
```

(A) The result above is the original magma and the result below is the modified magma. Both are the result of using afl to fuzz libpng by 15 minutes

```
project@ubuntu-s-2vcpu-4gb-fra1-01:~/magma/tools/captain/workdir/ar/fairfuzz/libxml2/xmllint/0/monitor$ cat 895
AAH037_R, AAH037_T, AAH041_R, AAH041_T, AAH024_R, AAH024_T, AAH029_R, AAH029_T, AAH035_R, AAH035_T, AAH032_R, AAH032_T, AAH026_R, AAH026_T, AAH034_R, AAH034_T, AAH031_R, AAH031_T
11233611, 0, 226520, 33292, 4423, 0, 6340, 0, 63178, 0, 180816, 0, 456, 0, 488, 0, 593, 0
project@ubuntu-s-4vcpu-8gb-fra1-01:~/workdir/cache/fairfuzz/libxml2/xmllint/0/monitor$ cat 895
AAH037_R, AAH037_T, AAH041_R, AAH041_T, AAH024_R, AAH024_T, AAH029_R, AAH029_T, AAH035_R, AAH035_T, AAH032_R, AAH032_T, AAH026_R, AAH026_T, AAH034_R, AAH034_T, AAH031_R, AAH031_T
972087, 0, 18005, 9674, 989, 0, 1719, 0, 14848, 0, 888, 0, 456, 0, 488, 0, 1975, 0
```

(B) The result above is the original magma and the result below is the modified magma. Both are the result of using fairfuzz to fuzz xmllint by 15 minutes

FIGURE 2.1
The results of modified magma and the original magma on two fuzzers and two programs

2.5 RESULTS

Limited by the physical computing resources we owned, we have to use virtual machines to test how is the modification of magma. We tried Azure's virtual machines first but the configuration of network is more complicated than we expected. Therefore we finally create three virtual machines on Digital Ocean to test the modified magma. All of these machines are equipped with Ubuntu 20.04. Due to the expensive cost of these virtual machines, the performances of three virtual machines are limited. The actual virtual machine we used are: one machine with 1 processor, 1 virtual cpu and 2GB memory, one machine with 1 processor, 2 virtual cpus and 4 GB memory and one machine with 3 processors, 4 virtual cpus and 8 GB memory. All of these processors are 2.3 MHz.

We run this modified magma to evaluate two fuzzers, afl and fairfuzz, by three targets: libpng, libtiff and libxml2 (5 programs in total). For simplicity we set the repeat times to 2 (each fuzzer run 2 times for each program) and timeout to 15 minutes. In total we run $2 * 2 * 5 = 20$ campaigns (services) simutaneously. To utilize the resources of machines, we set the cpu limitation of services to 1 cpu and the cpu reservation of services to 0.8 cpu. We do not add limitation to memories since fuzzers do not consume much memory.

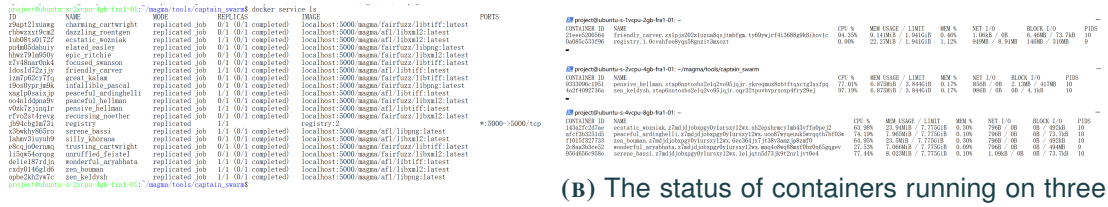
2.5.1 CORRECTNESS

We need to ensure that the modified magma will not crash in normal cases and provide the evaluation of fuzzers the same as the original magma. We first check all the logs of the docker services. All the fuzzers exit normally.

Next we compare the result provided by the original magma and the modified magma. Here I only show two results, one is the result of afl and libpng_read_fuzzer and the other is the result of fairfuzz and xmllint. The other results are similar.

We can see from Figure.2.1 that the relative relationship between how many times the fuzzer reach and trigger these bugs is the same. The differences between the two results mainly come from two aspects: one is the randomness of fuzzing and the other is the different cpu usage. In the original magma the fuzzer uses almost 95% of the cpus while in the modified magma the cpu usage of a fuzzer fluctuates especially when multiple fuzzers share the same cpu resources.

Through these two experimental results we prove that the modification for automatic orchestration does not affect the correctness of the magma.



(A) The status of the services

(B) The status of containers running on three machines

FIGURE 2.2
The status of the 20 services launched by the modified magma

2.5.2 LOAD BALANCING

We also need to test the load balancing for swarm mode to avoid assigning too many tasks on a single machine. Figure. 2.2 shows the status of these 20 services (registry should be ignored). We can see from Figure. 2.2a that among these 20 services, only 8 services are marked 1/1 in the replicas field while the other 12 services are marked 0/1 in this field. This means that the swarm cluster is running 8 services now. The rest 12 services, though already started, stay in pending state since no suitable nodes satisfy the resource constraints. This is reasonable if we consider each container exactly uses 1 cpu and the cluster should run $1 + 2 + 4 = 7$ containers at the same time. The reason for the additional running container is that sometimes the container uses less than 1 cpu and the remaining cpu resources satisfy the reservation constraint. Therefore a new task is assigned to this node.

Figure.2.2b lists the status of docker container on three virtual machines. We can see that the number of tasks running on a machine is basically linear with the number of cpus this machine owns, which is the same as the original magma. In addition we can see most of the tasks use 70%+ cpus. This shows that the resources of the cluster are fully utilized. We also counted the number of services that each machine runs. The machine with 2 virtual cpus runs 5 services in total, the machine with 1 virtual cpu runs 3 services intotal and the machine with 4 cpus runs 12 services in total. The number of running services is also basically linear with the number of cpus this machine owns.

One thing needs to be mentioned is that the overhead of scheduling is very low. The service only takes seconds before it is truly running on some machine. These results illustrate that the load balancing in the swarm mode works exactly as we expect. The containers work basically the same as they do in the original magma.

2.6 DISCUSSIONS

The resource reservation constraint is only applied when choosing nodes. After the container is deployed, there does not have any constraints except maximum resource usage limitations. The fuzzers will consume most of the cpu resources that it can use. Therefore it is common that the real resources usage is less than the expected reservations. Whether the differences in cpu usage by different containers will impact the evaluation of fuzzers is not clear now but this can be solved by simply applying a bigger reservation constraint. There also exists a downside for reserving resources. Reserving resources does not guarantee that you make optimum use of the resources available on the node. Hence if the reservation is set too large, the risk of resources being wasted increases.

Though gluster provides a good distributed file system for our cluster, there are lots of overhead on communicating, especially in our experiments that three machines are connected through the Internet. This overhead is a great problem for the . The service has to wait for I/O operations and the CPU usage thus becomes low. The swarm won't notice this. It just believes that enough cpu resources are availabe

and assigns more tasks to this machine, which makes the things evne worse. One potential solution for this is that we can find a way to sync up the data after all the services are finished since we do not need real-time data for the evaluation.

2.7 FUTURE WORK

The original magma schedules the cpu more fine-grained. It refines the cpu into the logical CPU core (like in simultaneous multi-threading, e.g., HyperThreading), the physical core (which disregard the presence of SMT and would allocate an entire core to the worker) and the socket (which would allocate an entire CPU chip to a worker). This feature is not supported by swarm mode whereas it is supported in kubernetes. We may change to kubernetes for container orchestration for better functionalities.

There are also some corner cases and locking problems, which are not taken into account for this project. In addtion the swarm mode dose not have any monitor. We may use some third-party monitoring tools for better monitoring the status of swarm cluster in the future.

BIBLIOGRAPHY

[1] Docker. *Swarm mode key concepts*. 2021. URL: https://docs.docker.com/engine/swarm/key-concepts/.
[2] Anwar Hassen. 'A survey of Docker Swarm scheduling strategies'. In: ().
[3] Docker. *Manage data in Docker*. 2021. URL: https://docs.docker.com/storage/.