基于 Docker 容器的快速部署工具

胡尧, <huyao@hit.edu.cn>

1 简介

1.1 工具简介

本工具用于大规模的 Docker 容器部署，其功能在于大规模快速地启动已有的 Docker 镜像。

1.2 环境要求

- Linux 3.13.x 以上
- Python 2.7.x 以上
- Docker 1.6.2 以上

1.3 文件结构

工具包含三个文件：ser.py、cli.py 和 fd.py。其中前两个分别是控制节点和计算节点的守护进程，第三个是运行在控制节点的控制程序，前两个程序需要在 root 权限下执行。

2 部署方案

2.1 部署要求

系统包含一个控制节点和多个计算节点，所有的节点都有独立的 IP。

2.2 部署流程

1. 在控制节点上执行“`python ser.py`”。
2. 在所有计算上执行“`python cli.py` {控制节点的 IP}”。
3. 在控制节点上使用 fd.py 来控制整个系统。

3 控制程序使用说明

执行“`python fd.py`”打印说明：

```
USAGE:
        ./proc list {node | task}
        ./proc start {IMAGE_NAME} {"NODE_INDEX_ARRAY"} {"COUNT_ARRAY"} [ARGV...]
        ./proc stop {TASK_ID}
        ./proc clean
```

图 1: 使用说明

3.1 列举信息

```
python fd.py list {node | task}
```

list 命令用于查看状态信息，包含一个参数，是一个选项，包含 node 和 task，分别对应节点信息和任务信息。

3.1.1 节点信息

当计算节点成功执行“python cli.py 控制节点的 IP”命令后，会在控制节点中注册。使用控制程序的列举节点信息功能可以看到当前已在控制节点注册的计算节点的编号和 IP：

```
hy@hy-MS-7715:~/FD$ python fd.py list node
0. 173.26.100.44:65523
1. 173.26.100.211:65523
```

图 2: 节点列表

3.1.2 任务信息

本系统的部署单位是任务，一个任务包含的信息有：部署使用的计算节点、每个节点上部署的容器数量、部署用的镜像。每次部署都会启动一个任务，每个任务对应一个镜像：

```
hy@hy-MS-7715:~/FD$ python fd.py list task
Task 1 with image "dktest":
        Node: 0 Count: 5
        Node: 1 Count: 10
Task 2 with image "dktest":
        Node: 0 Count: 3
        Node: 1 Count: 7
```

图 3: 任务列表

3.2 启动任务

```
python fd.py start {IMAGE_NAME} {"NODE_INDEX_ARRAY"} {"COUNT_ARRAY"} [ARGV...]
```

start 命令就是用来启动一个部署任务，包含四个参数：第一个是镜像名，第二个是节点数组，第三个是每个对应节点需要启动的容器的数量数组，第四个参数是容器启动参数（可选）。

其中镜像名对应的镜像只需要在控制节点中存在就行，节点数组对应节点信息的编号，用空格隔开，容器数量数组与节点数组对应，代表着对应节点上启动的容器的数量，第四个是 Docker 容器启动参数，这个跟镜像有关，可选。

例如：`python fd.py start dktest "0 1" "5 10"` 这条命令在第 0 号和第 1 号节点上面分别启动了 5 个和 10 个容器，镜像名称为：“dktest”：

```
hy@hy-MS-7715:~/FD$ python fd.py start dktest "0 1" "3 7" test
Task 2 has been started
```

图 4: 启动任务

任务启动后，会返回任务的编号。同时，计算节点和控制节点中的守护进程都会打印出状态信息：

```
connect from  ('173.26.102.245', 45944)
Task_id: 2
Img_name: dktest
Count: 7
Param: test
New task [2] to start 7 dktest......
create container: 8e4ac4167df46e3cd90dca95c58f89fd0137e72ddd322313e68282aa28144fc6
create container: d396c9349d0f32b5ebfd654ecc387e0e68b2139bd6470ebda1b61e2e409adb72
create container: 43b8f1baf4fa993f6f20a30e1938733d5c46532eae57f81d33e3b48bd6b49ba3
create container: 94da9e5efe542627481f65f02f7f77b7ec095ddfe95fe018ffb0de96aeafee57
create container: 8224d2f810cd10e57dd2e5779aac415802fb9d15886de6ddaf4b4d3bc3d98abc
create container: 04366b2ed4723fed105fb78316eb88d2f4fe4f0711ba47e59af8ea9a403fab24
create container: 6bb424a1de1420cdc33def63cb3409c9b0fa1ab81526df48467bd159d2826888
Create successfully!
```

图 5: 计算节点信息

```
connect from  ('127.0.0.1', 37686)
Add Image: dktest
Add Param: test
ADD C_INDEX: 0 C_COUNT: 3
ADD C_INDEX: 1 C_COUNT: 7
Task 2 has been added
Task 2 has been deployed
```

图 6: 控制节点信息

3.3 停止任务

`python fd.py stop {TASK_ID}`

stop 命令停止指定编号的任务，该命令有一个参数，即任务编号，编号可以通过任务信息查询到：

```
hy@hy-MS-7715:~/FD$ python fd.py list task
Task 1 with image "dktest":
        Node: 0 Count: 5
        Node: 1 Count: 10
Task 2 with image "dktest":
        Node: 0 Count: 3
        Node: 1 Count: 7
hy@hy-MS-7715:~/FD$ python fd.py stop 1
Task 1 has been killed
hy@hy-MS-7715:~/FD$ python fd.py list task
Task 2 with image "dktest":
        Node: 0 Count: 3
        Node: 1 Count: 7
```

图 7: 停止任务

3.4 清理任务

`python fd.py clean`

clean 命令将停止所有任务，并终止所有计算节点上由本系统部署的容器。

3.5 数据结构

3.5.1 Arena

Arena 是顶层内存管理实体，包括两种类型：Main arena 和 Non-main arena。

Main arena 就是内存映射中传统意义上的堆区，即 start_brk 到 brk 的区域，通常情况下一个进程只有一个 Main arena。Non-main arena 管理着通过 mmap() 系统调用分配的内存，根据线程的数目，可以有多个。本教程暂不讨论 Non-main arena。

3.5.2 chunk

chunk 是底层内存管理实体，我们使用 malloc 函数返回的每一段内存都对应着一个 chunk，堆中的数据就是由一个一个 chunk 构成的，其数据结构定义如下：

```
1  struct malloc_chunk {
2
3    INTERNAL_SIZE_T      prev_size;  /* Size of previous chunk (if free).  */
4    INTERNAL_SIZE_T      size;       /* Size in bytes, including overhead. */
5
6    struct malloc_chunk* fd;         /* double links -- used only if free. */
7    struct malloc_chunk* bk;
8
9    /* Only used for large blocks: pointer to next larger size.  */
10   struct malloc_chunk* fd_nextsize;
11   struct malloc_chunk* bk_nextsize;
12 };
```

3.5.3 题目描述

例题源于 SCTF 2016 PWN 300，大家看讲解之前先要对程序有个大致的分析，教程中就不详细讲分析过程了。用 IDA 打开，定位到主函数：

4 总结

本章教程中，我们介绍了堆内存的管理机制，并通过例题讲解了使用 double free 方式对 unlink 漏洞的利用过程，并给出了完整的利用脚本。

堆溢出的漏洞还有很多，例如 fastbin[1]的相关漏洞等，这些漏洞随便拿一个就能作为一章进行讲解。限于篇幅，这里只是列举了一个进行讲解。本章教程的难度和篇幅都比前几章的要大，希望大家多花点时间研究一下。学习过程中可能需要参考大量资料，并进行试验验证。

到此为止，二进制渗透入门教程的内容在基于笔者的知识覆盖范围内的前提下已经讲解完毕，标准教程 4 章就此结束。接下来可能会对之前的教程 Debug，还可能出一些技巧性、总结性的补充（也有可能没有了 23333）。

[1]浅析 Linux 堆溢出之 fastbin