CS-E4760: Platform Security (2023)
Exercise 3: Run-time security

Part 1: Theory

Answer the following questions:

1. Undefined behaviour.

Secure programs generally avoid triggering undefined behaviour, as this means that the language standard allows anything to occur. Each time an action triggers undefined behaviour, it could be ignored, or it could hand control of the user's machine to an attacker: both are equally legal.

a. If an application in its source code representation triggers undefined behaviour, can we conclude that it is insecure when run? Why [not]? [2 points]

From lecture slides:
An undefined behavior is a behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements.
"No requirements" means all the followings are allowed:
• Detect the error and crash
• Assume that it never happens
• Hand control to an attacker

Insecurity means that the attacker can exploit the the program code when the code enters the weird states outside the intended finite states. By the definition of undefined behavior, if an application in its source code representation triggers UB, we cannot conclude if it is necessarily insecure during runtime. For example, consider this code

```
#include <iostream>
int main(){
    int n = 0;
    if (0){
      n=n++; // Undefined behaviour if executed, but never executed
    }
    printf("Hello world.\n");
}
```

In this example, since if (0) is always false, the UB is never executed. So an UB may not imply any insecurities. On the other hand, if an UB is being guarded by the compiler by being supppressed from throwing an exception or crashing, it also cannot be exploitable by an

attacker. In some cases, array buffer overflows can be exploited by attackers, such as overwriting sensitive data. In conclusion, a representation of an UB in the source code does not decisively imply it is insecure during runtime. Having said that, trying to minimize the number of UBs helps reduce the security risks exploitable by attackers.

> **b. Undefined behaviour in a language standard is often contrasted with implementation-defined behaviour, defined in Section 3.4.1 of the C99 standard as "unspecified behavior where each implementation documents how the choice is made". Must developers avoid implementation-defined behaviour to develop a secure program? Why [not]? [2 points]**

From the C99 standard:
- Unspecified behavior is the use of an unspecified value, or other behavior where this International Standard provides two or more possibilities and imposes no further requirements on which is chosen in any instance. An example of unspecified behavior is the order in which the arguments to a function are evaluated (either left to right or right to left).
- Implementation-defined behavior is unspecified behavior where each implementation documents how the choice is made. An example of implementation-defined behavior is the propagation of the high-order bit when a signed integer is shifted right.

Based on these two definitions, it is concluded that implementation defined behaviour is not undefined behaviour. However, implementation-defined behavior results in behavior differences in the program code when it runs on different OS, settings or even compilers. As a consequence, testing and debugging a program becomes more challenging as it is subject to the running environment, while the attack surface is widened as the attackers have more choices of attack based on a particular vulnerable implementation. However, sometimes the implementation-defined behaviors cannot be avoided depending on the requirements of the program. So in conclusion, while developers should make efforts to understand and avoid implementation-defined behaviors, they should study and minimize the potential security risks caused by the program if the implementation-defined code cannot be avoided.

> **c. Formal methods can be used to mathematically prove that one representation of a program is a refinement of another (that is, that its behaviour is equivalent after transformation from a higher-level form to a lower-level form).**
> **Can we prove the correctness of a program by using formal methods to show that each step in the program transformation model described in the lecture slides is a refinement of the previous one? Why [not]? [2 points]**

In formal methods, program refinement is the verifiable transformation of an abstract (high-level) formal specification into a concrete (low-level) executable program. Stepwise refinement allows this process to be done in stages.

The program transformation model desecribed in the lecture slides example:

Type confusion

```
struct point {
    uint64_t x;
    uint64_t y;
    uint64_t z;
};
```

```
void read_point(struct point* out) {
    scanf("%ld %ld %ld",
                &out->x,
                &out->y,
                &out->z);
}
```

```
read_point:
        mov     rsi, rdi
        lea     rdx, [rdi + 8]
        lea     rcx, [rdi + 16]
        lea     rdi, [rip + .L.str]
        xor     eax, eax
        jmp     __isoc99_scanf@PLT
```

&out->x = out

&out->y

&out->z

Weird machines

Recall: $\theta = (Q, i, F, \Sigma, \Delta, \delta, \sigma)$

Q = set of states, i = initial state

F = final state, Σ, Δ = input and output alphabets

state transition function $\delta: Q \times \Sigma \to Q$, output function $\sigma: Q \times \Sigma \to \Delta$

A weird machine is a computational device where IFSM transitions operate on weird states

$$\theta_{weird} = \left(Q_{cpu}^{weird}, q_{init}, Q_{cpu}^{IFSM} \cup Q_{cpu}^{trans}, \Sigma', \Delta', \delta', \sigma'\right)$$

Instruction stream depends on input

- weird machine programmed through carefully crafted input to p once q_{init} has been entered

Based on the theory of finite state machines in the CPU, the lower-level form of the high level-code can directly determine the states at which the code is current contained in (Q_{trans},Q_{IFSM} and Q_{weird}), while the commands such as mov determines the transition δ between the states. Therefore, formal methods can determine if the code enters the weird states by evaluating the safe state q_i and weird state q_{init} that the low-level code is located in.

Conclusion: we can prove the correctness of a program by using formal methods

Part 2: Vulnerabilities

Write the following code:

2. Triggering undefined behaviour.

For each of the following kinds of undefined behaviour, write a program in C that triggers it and demonstrates how it can cause data corruption (e.g. by crashing or modifying data). Build your programs in a Docker container for ease of reproduction.

a. Stack array overflow [4 points]

I have an array of 10 integers, and I try to access and write to the 11th element, which causes a stack array overflow error

```c
#include <stdio.h>

int main() {
    // Initialize an array of only 5 elements
    int stackoverflow[10];
    int i;
    for (i = 0; i <= 10; i++) {
        stackoverflow[i] = i;
    }
    // Accessing the eleventh element, causing stack array overflow
    printf("%d", stackoverflow[10]);
    return 0;
}
```

This is my Dockerfile

```
FROM ubuntu:latest

RUN apt-get update && apt-get install -y gcc

WORKDIR /app

COPY stackoverflow.c /app

RUN gcc -o stackoverflow stackoverflow.c

CMD "./stackoverflow"
```

The stack array overflow error occurs when the docker image is run

```
springnuance@springnuance-VirtualBox:~/Desktop/UndefinedBehavior/stackoverflow
$ docker run stackoverflow
*** stack smashing detected ***: terminated
Aborted (core dumped)
```

b. Type confusion [4 points]

This error is produced by accessing an integer as a pointer

```c
#include <stdio.h>

int main() {
    int x = 5;
    printf("The value of x is: %p\n", x);
}
```

This is my Dockerfile

```
FROM ubuntu:latest

RUN apt-get update && apt-get install -y gcc

WORKDIR /app

COPY typeconfusion.c /app

RUN gcc -o typeconfusion typeconfusion.c

CMD "./typeconfusion"
```

The type confusion error causes some warning during compilation and outputs undefined data when an integer is accessed as a pointer.

```
Step 5/6 : RUN gcc -o typeconfusion typeconfusion.c
 ---> Running in f34b59a9457d
typeconfusion.c: In function 'main':
typeconfusion.c:5:33: warning: format '%p' expects argument of type 'void *', bu
t argument 2 has type 'int' [-Wformat=]
    5 |     printf("The value of x is: %p\n", x);
      |                                ~^     ~
      |                                 |     |
      |                                 |     int
      |                                 void *
      |                                %d
Removing intermediate container f34b59a9457d
 ---> ff62e21d8421
Step 6/6 : CMD "./typeconfusion"
 ---> Running in dffdade913c4
Removing intermediate container dffdade913c4
 ---> 4dd638e62521
Successfully built 4dd638e62521
Successfully tagged typeconfusion:latest
springnuance@springnuance-VirtualBox:~/Desktop/UndefinedBehavior/typeconfusion$
docker run typeconfusion
The value of x is: 0x5
```

As we can see, the printed value is 0x5, which looks neither like an integer nor an address. I believe the data has been modified to look something like this.

Part 3: Defences

Attempt the following steps, and answer the associated questions.

3. Address space layout randomization.

a. Modify one of your programs from Question 3 to corrupt a function pointer so that it will contain a user-supplied value from the terminal; then, call the function pointer. Add a function attack to the program that produces some output, so that you can see the results of your attack. [5 points]

Unfortunately, I run out of time to work on this assignment. Can you give me some hints?

b. Compile the program with the -fno-pie option to prevent the use of address space layout randomization (ASLR). Write a script that will print an input for the program that causes your new function to be called in place of the function pointer's original target [5 points]

Hints:

- **You may find the nm or objdump programs to be useful.**
- **You may need to pay attention to the endianness of your platform.**

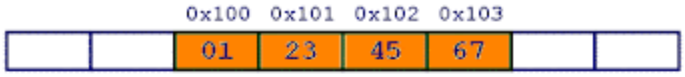
First of all, I need to check the endianness of my Linux system. Source code: https://cs-fundamentals.com/tech-interview/c/c-program-to-check-little-and-big-endian-architecture

```c
/*
   Write a C program to find out if the underlying
   architecture is little endian or big endian.
 */

#include <stdio.h>
int main () {
    unsigned int x = 0x76543210;
    char *c = (char*) &x;
    printf ("*c is: 0x%x\n", *c);
    if (*c == 0x10){
        printf ("Underlying architecture is little endian. \n");
    } else {
        printf ("Underlying architecture is big endian. \n");
    }
    return 0;
}
```

```
springnuance@springnuance-VirtualBox:~/Desktop/ASLR/Part.A$
gcc -o endianness endianness.c
springnuance@springnuance-VirtualBox:~/Desktop/ASLR/Part.A$
./endianness
*c is: 0x10
Underlying architecture is little endian.
```

It appears that my Linux is little endian. Little and big endian are two ways of storing multibyte data-types (int, float, etc). In little endian machines, last byte of binary representation of the multibyte data-type is stored first. On the other hand, in big endian machines, first byte of binary representation of the multibyte data-type is stored first. Suppose integer is stored as 4 bytes (For those who are using DOS-based compilers such as C++ 3.0, integer is 2 bytes) then a variable x with value 0x01234567 will be stored as following.



Big Endian

Little Endian

This is the main source code that demonstrates this part (b)

```c
#include <stdio.h>

void attack() {
    printf("I have been attacked! That's impressive!\n");
}

void benign() {
    printf("I am still not hacked! Please try again\n");
}

int main(int argc, char* argv[]) {
    printf("Hidden function is at %p\n",&attack);
    printf("Enter a string: ");
    // gets is a very dangerous function, which does not check
    // the limit of the input size. Attackers can use this loophole
```

```
    // to inject attack function address into this buffer
    char buffer[4];
    gets(buffer);
    printf("Your input data is %s\n",buffer);
    // Assigning the function pointer to a safe function
    void (*fp)() = benign;
    // Calling the benign function. Let's see how
    // array buffer overflow attack can change this
    // to the malicious function
    fp();
    return 0;
}
```

Now I compile the program. A warning about the gets function appears and it is indeed true that gets is a dangerous reading function in C

```
springnuance@springnuance-VirtualBox:~/Desktop/ASLR/Part.B$ gcc -o functionpointer functionpointer.c
functionpointer.c: In function ‘main’:
functionpointer.c:14:5: warning: implicit declaration of function ‘gets’; did you mean ‘fgets’? [-Wi
mplicit-function-declaration]
   14 |     gets(buffer);
      |     ^~~~
      |     fgets
/usr/bin/ld: /tmp/ccW3lrnI.o: in function `main':
functionpointer.c:(.text+0x86): warning: the `gets' function is dangerous and should not be used.
```

Moving to the next step, the address space layout randomization (ASLR) is a memory protection process for OSes that guards against buffer-overflow attacks by randomizing the location where system executables are loaded into memory. Turning off ASLR is required in this part (b).
In compiling the code, the -fno-pie is a flag that stands for "no position independent executable" and is used to disable the generation of position-independent code. In other words, this flag helps turn off the ASLR feature, which prevents buffer overflows vulnerabilities that can be exploited by attackers. Before turning off ASLR, each time the program is run, the address of the attack function always change.

```
springnuance@springnuance-VirtualBox:~/Desktop/ASLR/Part.B$ ./functionpointer
Attack function address is at 0x5626be8761a9
```

```
springnuance@springnuance-VirtualBox:~/Desktop/ASLR/Part.B$ ./functionpointer
Attack function address is at 0x559dbefac1a9
```

Because I cannot turn off ASLR using the tag -fno-pie, I use this command instead
$ sudo sysctl kernel.randomize_va_space=0
Now ASLR is disabled

```
springnuance@springnuance-VirtualBox:~/Desktop/ASLR/Part.B$
 sudo sysctl kernel.randomize_va_space=0
kernel.randomize_va_space = 0
```

To check if ASLR is correctly disabled, I need to recompile the program to check if the address of the attack function stays the same

```
springnuance@springnuance-VirtualBox:~/Desktop/ASLR/Part.B$ ./functionpointer
Attack function address is at 0x5555555551a9
```

```
springnuance@springnuance-VirtualBox:~/Desktop/ASLR/Part.B$ ./functionpointer
Attack function address is at 0x5555555551a9
```

The attack function stays the same, so ASLR is properly disabled.

Because the buffer only allows 4 bytes, if I input more than 4 bytes, the buffer overflow error will occur like this

```
springnuance@springnuance-VirtualBox:~/Desktop/ASLR/Part.B$ ./functionpointer
Attack function address is at 0x5555555551a9
Enter a string: 12345
Your input data is 12345
I am still not hacked! Please try again
*** stack smashing detected ***: terminated
Aborted (core dumped)
```

In order to write a script that will print an input for the program that causes the attack function to be called in place of the function pointer's original target benign function, two information needs to be known: the offset amount from the end of buffer until the benign function, and the address of the attack function, which is already known above. The buffer space grows towards the Base Pointer (BP) and Instruction Pointer (IP) from lower memory to higher memory, and below BP, there will be IP/Return Address. When we overwrite some important registers like IP andBP, it points to an address which can be utilized for exploitation. If the address is not meaningful, it gives “ERRORS”(Segmentation Fault/Stack Smashing Detected/Core Dumped)

gcc -g functionpointer.c -o functionpointer
The little endian representation of the attack 0x5555555551a9 address is 0xA9 0x51 0x55 0x55 0x55 0x55

Unfortunately, I run out of time to work on this assignment. Can you give me some hints?

c. Compile your program with -fpie -pie instead of -fno-pie. Verify that your original attack no longer works.

i. Why does the attack no longer work? [1 point]

The attack no longer works because address space layout randomization (ASLR) is enabled, which randomizes the memory addresses of functions pointer in the program. This makes it more difficult for an attacker to predict the memory address of the attack function

ii. What other information does the attacker need in order to carry out the attack? Modify your vulnerable program to leak this information, and demonstrate a new attack. [5 points]

> **NB: Leaking the address of *attack* directly (e.g. by printing it with printf("%p\n", attack)) is too trivial and will not receive any points. The address of anything else, or any other information, is fair game.**

Unfortunately, I run out of time to work on this assignment. Can you give me some hints?

Submission

Submit to MyCourses your answers to the questions above as a **text or PDF file (not ZIP)**. Attach a copy of your programs and scripts, along with the Dockerfile used to build them.