Holden Stefan
09/14/2021

Demonstrating Risky Resource Management

Summary:

Resource management is the term used to describe the creation, transfer, use, and destruction of resources such as memory. These vulnerabilities are particularly dangerous as they often pertain directly to the operating system or even the hardware that a piece of software is running on, putting very high level privileges at risk. Demonstrated here are 2 examples of insecure resource management: 'CWE-120: Classic Buffer Overflow' and 'CWE-190: Integer Overflow or Wraparound'.

Buffer overflows relate to the Stack (CPU memory) where data and functionality that are in use by a program are stored. The major danger with this vulnerability is that if it is improperly mitigated, it can allow malicious code to be run directly on the CPU, making it very easy to get root access. In order to mitigate my example, a combination of strategies were used including the default OS protections (ASLR, DEP, and Stack Canaries) as well as substituting the 'strcpy()' function for the more suitable 'strncpy()' function, allowing the input to be truncated at a specified length. This effectively prevented the buffer overflow from overwriting the instruction pointer register.

The second demonstration is for integer overflows, in which a very basic and unrealistic banking application was created that allowed for unfiltered numerical input to be entered into dangerous functions. This vulnerability is related to memory and how data is stored. Computers must use binary data, and therefore have limits to the maximum value that can be stored. For example, a 32 bit system the maximum value for an integer is 2^{32} (that is 32 bits --32 x either a 1 or 0). The default integer value is signed, which means that 1 bit is designated for the sign, which leaves 31 bits to store the numerical value (1 repeated 31 times = 2,147,483,647). There just simply aren't enough bits to store a value larger, and if an attempt is made to force a larger number into a fixed register, an integer overflow occurs (and the specifics of how a computer handles that is implementation-defined). There are many ways in which this can happen --oftentimes operations adding large values, conversions from signed to unsigned or in value-check scenarios. If these occur at a critical variable, such as when defining a buffer size, then it can lead to a buffer overflow as the buffer is given a much smaller range than the data being inputted. In this demonstration the main vulnerability was in the conversion from an unsigned integer to a signed integer, where you have twice as much data in the unsigned vs signed. In order to effectively mitigate this, 2 strategies were used: First all inputs were changed to floating points, which can store significantly more values in the same 32 bits ($3.4*10^{38}$ on 32 bit systems), and second, creating a value check that prevented too large of a number from being inputted (since this is a banking application, it made sense to make the maximum inputs less than $100,000).

Example 1 - [CWE-120: Buffer Copy Without Checking Size of Input]

Overview:

For the demonstration of buffer overflow, I created a very basic program written in C, in which the program is run with an argument, and that argument is copied to the buffer. If it runs properly, it simply outputs “Finished…”, but if there is an error, it will output “Segmentation fault”.

```
vocstartsoft:~/environment/buffer_overflow $ ./bufferoverflow
Input: <buffer input>
vocstartsoft:~/environment/buffer_overflow $ ./bufferoverflow aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Segmentation fault
vocstartsoft:~/environment/buffer_overflow $ ./bufferoverflow aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Finished...
vocstartsoft:~/environment/buffer_overflow $ 
```

Analysis:

First, in order for this vulnerability to be exploited, there are a few default safety mechanisms that need to be turned off. In the OS, ASLR must be turned off, which on AWS CLI is the command ‘echo 0 | sudo tee /proc/sys/kernel/randomize_va_space’. In the compiler, DEP and Stack Canaries must be turned off in order to allow for execution of code in the stack. In GCC, this can be done with the ‘-fno-stack-protector’ and ‘-zexecstack’ flags. Secondly, in order to analyze the vulnerability, it is very helpful to use a debugger with a breakpoint just after the input is copied to the buffer, which prevents the program from crashing. Furthermore, you can pull the buffer memory, as well as an info frame, which allows you to easily see the copied input, as well as where the instruction pointer is, and what is stored inside of it.

```
(gdb) run aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/ec2-user/environment/buffer_overflow/bufferoverflow aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

Breakpoint 1, overflowFunction (input=0x7fffffffe050 'a' <repeats 75 times>) at bufferoverflow.c:8
8               strcpy(buffer, input);
(gdb) continue
Continuing.

Breakpoint 2, overflowFunction (input=0x7fffffffe050 'a' <repeats 75 times>) at bufferoverflow.c:9
9       }
(gdb) info frame
Stack level 0, frame at 0x7fffffffdbf0:
 rip = 0x400576 in overflowFunction (bufferoverflow.c:9); saved rip = 0x4005b5
 called by frame at 0x7fffffffdc10
 source language c.
 Arglist at 0x7fffffffdbe0, args: input=0x7fffffffe050 'a' <repeats 75 times>
 Locals at 0x7fffffffdbe0, Previous frame's sp is 0x7fffffffdbf0
 Saved registers:
  rbp at 0x7fffffffdbe0, rip at 0x7fffffffdbe8
(gdb) x /128bx buffer
0x7fffffffdb90: 0x61    0x61    0x61    0x61    0x61    0x61    0x61    0x61
0x7fffffffdb98: 0x61    0x61    0x61    0x61    0x61    0x61    0x61    0x61
0x7fffffffdba0: 0x61    0x61    0x61    0x61    0x61    0x61    0x61    0x61
0x7fffffffdba8: 0x61    0x61    0x61    0x61    0x61    0x61    0x61    0x61
0x7fffffffdbb0: 0x61    0x61    0x61    0x61    0x61    0x61    0x61    0x61
0x7fffffffdbb8: 0x61    0x61    0x61    0x61    0x61    0x61    0x61    0x61
0x7fffffffdbc0: 0x61    0x61    0x61    0x61    0x61    0x61    0x61    0x61
0x7fffffffdbc8: 0x61    0x61    0x61    0x61    0x61    0x61    0x61    0x61
0x7fffffffdbd0: 0x61    0x61    0x61    0x61    0x61    0x61    0x61    0x61
0x7fffffffdbd8: 0x61    0x61    0x61    0x00    0x00    0x00    0x00    0x00
0x7fffffffdbe0: 0x00    0xdc    0xff    0xff    0xff    0x7f    0x00    0x00
0x7fffffffdbe8: 0xb5    0x05    0x40    0x00    0x00    0x00    0x00    0x00
0x7fffffffdbf0: 0xe8    0xdc    0xff    0xff    0xff    0x7f    0x00    0x00
0x7fffffffdbf8: 0x00    0x00    0x00    0x00    0x02    0x00    0x00    0x00
0x7fffffffdc00: 0xd0    0x05    0x40    0x00    0x00    0x00    0x00    0x00
0x7fffffffdc08: 0xba    0xf0    0xa4    0xf7    0xff    0x7f    0x00    0x00
(gdb) 
```

In the image above, you can see the info frame as well as 128 bytes beginning with the buffer. The vulnerable function:

```
void overflowFunction(char* input) {
        char buffer[80];
        strcpy(buffer, input);}
```

Has a buffer size of 80 bytes, and the above image was given an input of 75 bytes. The addresses circled in red (rip) are where the next instruction is stored, and what an attacker would try to overwrite. Since this input is within the proper range, it was not overwritten. We can however see that the rip begins 88 bytes after the beginning of the buffer (0x7fffffffdbe8-0x7fffffffdb90 = 88 bytes, and we therefore know that in order to overwrite the rip, we must use an input of 96 bytes (80 byte buffer + 8 byte padding + 8 byte rip).

```
(gdb) run aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/ec2-user/environment/buffer_overflow/bufferoverflow aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

Breakpoint 1, overflowFunction (input=0x7fffffffe03b 'a' <repeats 96 times>) at bufferoverflow.c:8
8               strcpy(buffer, input);
(gdb) continue
Continuing.

Breakpoint 2, overflowFunction (input=0x7fffffffe03b 'a' <repeats 96 times>) at bufferoverflow.c:9
9       }
(gdb) info frame
Stack level 0, frame at 0x7fffffffdbe0:
 rip = 0x400576 in overflowFunction (bufferoverflow.c:9); saved rip = 0x6161616161616161
 called by frame at 0x7fffffffdbe8
 source language c.
 Arglist at 0x7fffffffdbd0, args: input=0x7fffffffe03b 'a' <repeats 96 times>
 Locals at 0x7fffffffdbd0, Previous frame's sp is 0x7fffffffdbe0
 Saved registers:
  rbp at 0x7fffffffdbd0, rip at 0x7fffffffdbd8
(gdb) x /128bx buffer
0x7fffffffdb80: 0x61    0x61    0x61    0x61    0x61    0x61    0x61    0x61
0x7fffffffdb88: 0x61    0x61    0x61    0x61    0x61    0x61    0x61    0x61
0x7fffffffdb90: 0x61    0x61    0x61    0x61    0x61    0x61    0x61    0x61
0x7fffffffdb98: 0x61    0x61    0x61    0x61    0x61    0x61    0x61    0x61
0x7fffffffdba0: 0x61    0x61    0x61    0x61    0x61    0x61    0x61    0x61
0x7fffffffdba8: 0x61    0x61    0x61    0x61    0x61    0x61    0x61    0x61
0x7fffffffdbb0: 0x61    0x61    0x61    0x61    0x61    0x61    0x61    0x61
0x7fffffffdbb8: 0x61    0x61    0x61    0x61    0x61    0x61    0x61    0x61
0x7fffffffdbc0: 0x61    0x61    0x61    0x61    0x61    0x61    0x61    0x61
0x7fffffffdbc8: 0x61    0x61    0x61    0x61    0x61    0x61    0x61    0x61
0x7fffffffdbd0: 0x61    0x61    0x61    0x61    0x61    0x61    0x61    0x61
0x7fffffffdbd8: 0x61    0x61    0x61    0x61    0x61    0x61    0x61    0x61
0x7fffffffdbe0: 0x00    0xdc    0xff    0xff    0xff    0x7f    0x00    0x00
0x7fffffffdbe8: 0x00    0x00    0x00    0x00    0x02    0x00    0x00    0x00
0x7fffffffdbf0: 0xd0    0x05    0x40    0x00    0x00    0x00    0x00    0x00
0x7fffffffdbf8: 0xba    0xf0    0xa4    0xf7    0xff    0x7f    0x00    0x00
(gdb)
```

When the program is run again with ‘a’ repeated 96 times, we can see that the ‘saved rip’ contains 0x6161616161616161, which is ‘a’ re[eated 8 times -- we can successfully overwrite the instruction pointer. In order to fully exploit this, we need to change the last 8 bytes of the input to the 8 byte address at the beginning of the buffer. We can then insert shellcode into the beginning of the buffer, and effectively get root access to the machine. I was unfortunately unable to do this as AWS CLI kept interpreting my hexadecimal input as ASCII, converting it to 24 bytes.

Mitigation:

In order to mitigate this vulnerability, the first step was to enable ASLR, DEP, and Stack Canaries, which are usually on by default. However, for reasons that I was not able to understand, ASLR did not effectively change the address of the buffer or rip, even in between reboots or recompilations. Secondly, and most effective, the function ‘strcpy()’ was replaced with ‘strncpy()’ in which a maximum size is defined before copying:

```
void overflowFunction(char* input) {
        char buffer[80];
        strncpy(buffer, input, sizeof(buffer)-1);]
```

In the snippet above, the input is copied to the buffer, but only up to the point of ‘sizeof(buffer)-1’, which allows for the string to be null terminated.

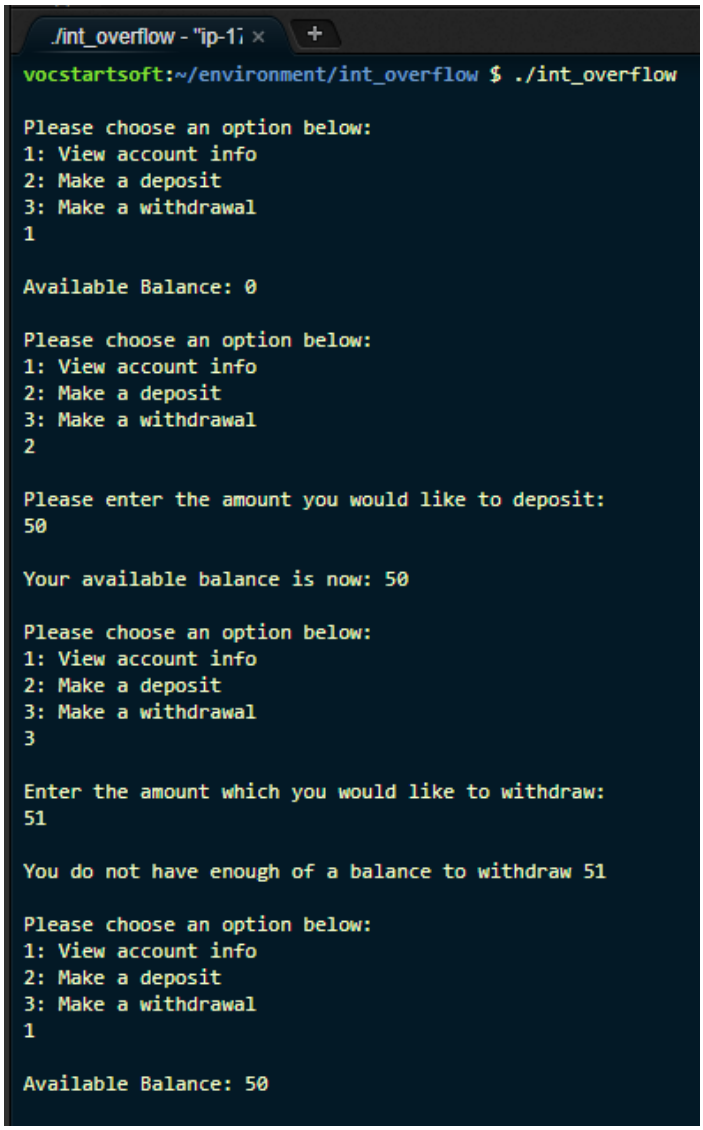
```
(gdb) run aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/ec2-user/environment/buffer_overflow/fixed_bufferoverflow aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaa

Breakpoint 1, overflowFunction (input=0x7fffffffdef8 'a' <repeats 120 times>) at fixed_bufferoverflow.c:9
9       }
(gdb) info frame
Stack level 0, frame at 0x7fffffffda90:
 rip = 0x40057b in overflowFunction (fixed_bufferoverflow.c:9); saved rip = 0x4005ba
 called by frame at 0x7fffffffdab0
 source language c.
 Arglist at 0x7fffffffda80, args: input=0x7fffffffdef8 'a' <repeats 120 times>
 Locals at 0x7fffffffda80, Previous frame's sp is 0x7fffffffda90
 Saved registers:
  rbp at 0x7fffffffda80, rip at 0x7fffffffda88
(gdb) x /128bx buffer
0x7fffffffda30: 0x61    0x61    0x61    0x61    0x61    0x61    0x61    0x61
0x7fffffffda38: 0x61    0x61    0x61    0x61    0x61    0x61    0x61    0x61
0x7fffffffda40: 0x61    0x61    0x61    0x61    0x61    0x61    0x61    0x61
0x7fffffffda48: 0x61    0x61    0x61    0x61    0x61    0x61    0x61    0x61
0x7fffffffda50: 0x61    0x61    0x61    0x61    0x61    0x61    0x61    0x61
0x7fffffffda58: 0x61    0x61    0x61    0x61    0x61    0x61    0x61    0x61
0x7fffffffda60: 0x61    0x61    0x61    0x61    0x61    0x61    0x61    0x61
0x7fffffffda68: 0x61    0x61    0x61    0x61    0x61    0x61    0x61    0x61
0x7fffffffda70: 0x61    0x61    0x61    0x61    0x61    0x61    0x61    0x61
0x7fffffffda78: 0x61    0x61    0x61    0x61    0x61    0x61    0x61    0x00
0x7fffffffda80: 0xa0    0xda    0xff    0xff    0xff    0x7f    0x00    0x00
0x7fffffffda88: 0xba    0x05    0x40    0x00    0x00    0x00    0x00    0x00
0x7fffffffda90: 0x88    0xdb    0xff    0xff    0xff    0x7f    0x00    0x00
0x7fffffffda98: 0x00    0x00    0x00    0x00    0x02    0x00    0x00    0x00
0x7fffffffdaa0: 0xd0    0x05    0x40    0x00    0x00    0x00    0x00    0x00
0x7fffffffdaa8: 0xba    0xf0    0xa4    0xf7    0xff    0x7f    0x00    0x00
(gdb)
```

In the above image, the corrected program was run with a buffer size of 80, and an input of ‘a’ repeated 120 times. As you can see, the ‘saved rip’ was not overwritten, and the buffer only contains ‘0x61’ repeated 79 times followed by a null terminator, which is exactly as the function intended.

Example 2 -[CWE-190: Integer Overflow or Wraparound]

Overview:

In this example, while not really something anyone would expect to see in the real world, for me provided a more concrete example of integer overflow. It is a basic banking app, written in C, which allows you to view your balance, make a deposit, or make a withdrawal. The only main logic is a value check in the withdrawal function, which prevents a withdrawal if you don't have enough money.



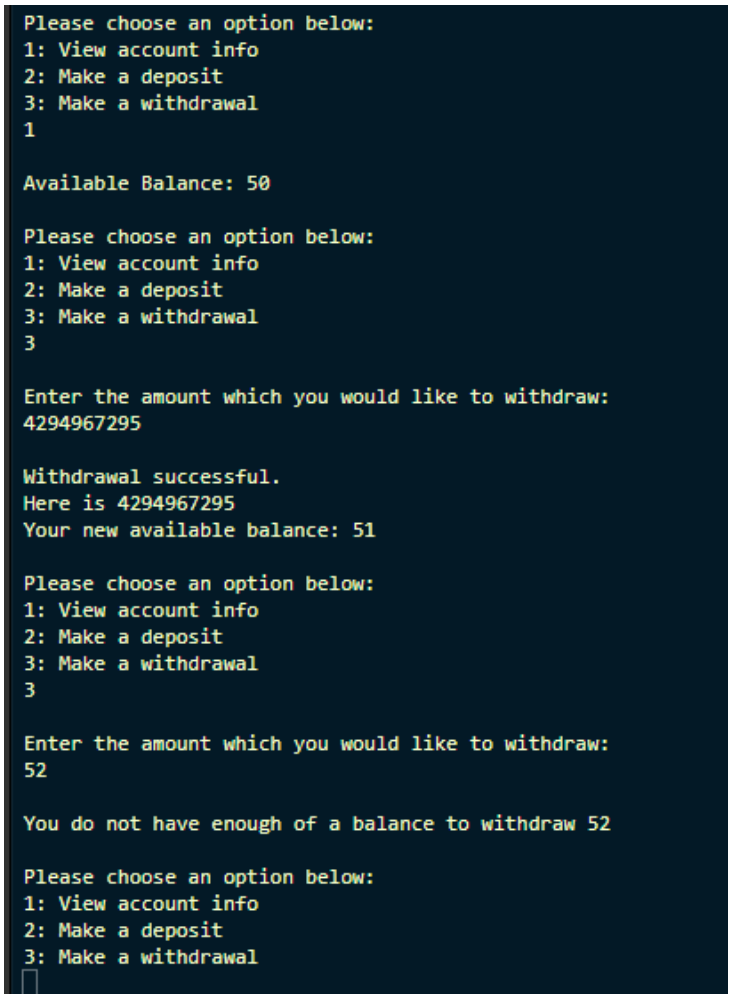
Analysis:

The vulnerability is mainly caused by a conversion between an unsigned integer to a signed integer in the value check of the withdrawal function. It allows for a very large number to pass the value check regardless of how much money is in your account (because the value can become negative, and the balance is always at least 0.

The vulnerable function:

```
void Withdraw() {
        unsigned int withdraw_num;
        printf("\nEnter the amount which you would like to withdraw:\n");
        scanf(" %u", &withdraw_num);

        if (available_balance >= (int) withdraw_num) {
                available_balance -= withdraw_num;
                printf("\nWithdrawal successful.");
                printf("\nHere is %u", withdraw_num);
                printf("\nYour new available balance: %i\n", available_balance);
        }
        else {
                printf("\nYou do not have enough of a balance to withdraw %u\n",
withdraw_num);
```

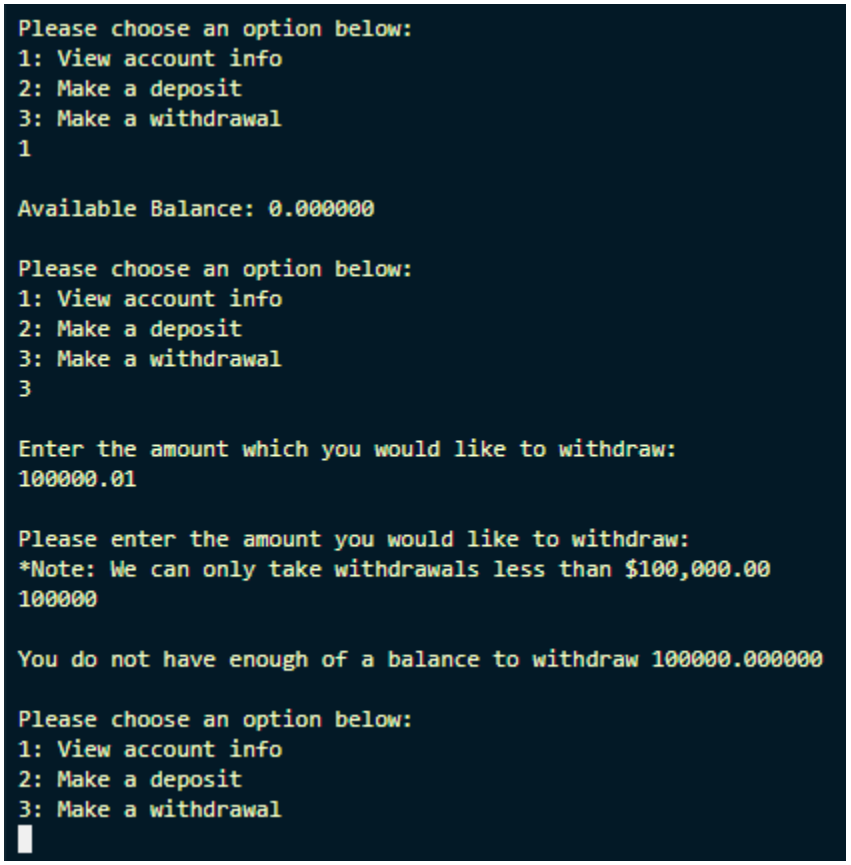
First makes the error of making the conversion without any checks, and also selectively using the signed int only within the value check. If the input is greater than 2^{31}, then the integer overflows, and errors occur.



In the example image, we can see that the application works as expected if the numbers are within a reasonable range, however, if the numbers are large enough, the program fails. In this case it prevented the withdrawal of $52 because there was only $50 in the account, yet it allowed the withdrawal of $4,294,967,295 when there was only $50 dollars in the account.

Mitigation:

In order to properly mitigate this weakness, first: all instances of integers, signed or unsigned were replaced with floating point values. This not only significantly raised the maximum value, but also just makes more sense when dealing with money. Secondly, a value check on the input was integrated. This could simply prevent inputting numbers greater than 2^31, however, as this is a banking app, and it seems incredibly unlikely for anyone to make individual 10-digit deposits or withdrawals, the limit was placed at 100,000. This effectively stopped the overflow from happening, as even if I wanted to I could not input a value large enough to overflow the integer register, let alone the floating point register.



The corrected function:

```
void Withdraw() {
	float withdraw_num;
	printf("\nEnter the amount which you would like to withdraw:\n");
	scanf(" %f", &withdraw_num);
	if (withdraw_num > 100000) {
		printf("\nPlease enter the amount you would like to withdraw:\n*Note: We can only take withdrawals less than $100,000.00\n");
		scanf(" %f", &withdraw_num);}
	if (available_balance >= withdraw_num) {
		available_balance -= withdraw_num;
		printf("\nWithdrawal successful.");
		printf("\nHere is %f", withdraw_num);
		printf("\nYour new available balance: %f\n", available_balance);}
	else {
		printf("\nYou do not have enough of a balance to withdraw %f\n", withdraw_num);
```

References

Gallagher, W. (2019, April 4). *Exploring buffer overflows in C, part two: The exploit*. Tallan. Retrieved September 14, 2021, from https://www.tallan.com/blog/2019/04/04/exploring-buffer-overflows-in-c-part-two-the-exploit/.

Microsoft. (2019, October 21). *C and c++ integer limits*. Microsoft Docs. Retrieved September 14, 2021, from https://docs.microsoft.com/en-us/cpp/c-language/cpp-integer-limits?view=msvc-160.

MITRE. (n.d.). *Common weakness enumeration*. CWE. Retrieved September 14, 2021, from https://cwe.mitre.org/data/definitions/190.html.

MITRE. (n.d.). *Common weakness enumeration*. CWE. Retrieved September 15, 2021, from https://cwe.mitre.org/data/definitions/120.html.

Poston, H. (2020, December 7). *How to exploit integer overflow and underflow*. Infosec Institute. Retrieved September 14, 2021, from https://resources.infosecinstitute.com/topic/how-to-exploit-integer-overflow-and-underflow/.