**Intro To Exploitation - FormatStrings Solution**

**Introduction:** FormatStrings is an easy exploitation challenge designed to teach students about basic format string vulnerabilities. Format string vulnerabilities allow attackers to read or write values to specific addresses specified by the attacker. This vulnerability arises when a programmer doesn’t properly pass input into a format function such as printf(). There’s an excellent writeup on format string vulnerabilities located over at [CodeArcana](http://codearcana.com/posts/2013/05/02/introduction-to-format-string-exploits.html) which dives into more detail then I’m going to cover here. I will however explain the basics.

Printf() and by extension other format functions can accept a variable length of parameters, assume the following as an example. If I had a printf() statement containing printf(“This is a format string, and specifies what is to be output. Here are some numbers %d %d %d”, 1, 3, 5). When executed the previous statement would output: “This is a format string, and specifies what is to be output. Here are some numbers 1 3 5”. In the case where the programmer specifies exactly what is to be output by printf() there exists no format string vulnerability. But what happens if the programmer writes something like, printf(user\_input[100]); ? Since printf could have variable number of arguments, printf() must use the format string provided by the programmer or passed by the user to determine the number of arguments. In the case above, an attacker can pass the string "%x %x %x %x %x %x %x %x %x %x %x %x %x %x %x" and fool the printf into thinking it has 15 arguments. Printf() will then naively print the next 15 addresses on the stack, thinking they are its arguments.

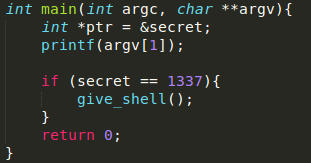
|  |  |  |
| --- | --- | --- |
| **Format String** | **Output** | **Usage** |
| %d | Decimal Numbers (Ints) | Outputs decimal numbers |
| %s | Character Strings | Reads strings from memory |
| %x | Hexadecimal Addresses | Output hexadecimal address |
| %n | Number of bytes written so far | Writes the number of bytes till the format string to memory |

There are 3 basic steps to exploiting any format string vulnerability.

1. Determine if the program is contains a format string vulnerability
2. Find the address you wish to read or write to on the stack
3. Exploit the vulnerability

Enough talk for now, let’s get into the challenge.

**Task:** The provided code and binary is vulnerable to format string exploits can you modify the ‘secret’ variable to pass the comparison and launch the a shell? I’ve provided the relevant code snippet below.



**Solving:** Let’s begin by verifying this program is in fact vulnerable to format string exploits. This is done by passing either of the following format string specifiers: ‘%x’, ‘%p’. The first format specifier %p tells printf() to print out a pointer. The second format specifier %x tells printf() to print out a hexadecimal number.

The way I prefer to test for format string vulnerabilities is as follows: ./programToRun ‘AAAA%x %x %x %x %x %x %x….’

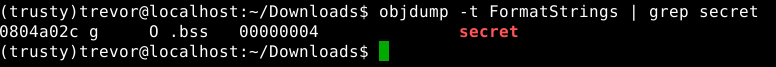
The AAAA portion of the string allows me to keep a reference of where we are on the stack once we start reading values. There are ‘....’s following the %x’s to indicate I can use as many, or as few %x arguments as a like. Let’s go ahead and give it a shot.

Screenshot - 080216 - 18:50:39.png

This program is infact vulnerable to format string exploits. When I passed the string ‘AAAA%x %x %x %x’ the program printed out ‘AAAA’ correctly and then proceeded to print out the addresses: 0xffad1434, 0xffad1440, 0xf755a4ad, and 0xf76d23c4 off of the stack.

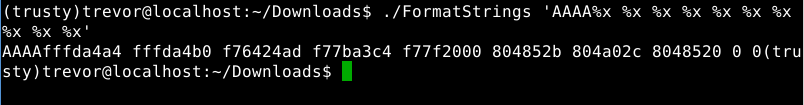
Now that we know this program is vulnerable it’s time to find the address of the ‘secret’ variable on the stack. To do this we can use the ‘objdump’ command as follows: ‘objdump -t PROGRAMTORUN | grep VARIABLE’.

In our case the proper usage would be: ‘objdump -t FormatStrings | grep secret’



The output of objdump tells us that the variable ‘secret’ is located at the address 0x0804a02c. Using the format string vulnerability we now need to find where the address 0x0804a02c is located on the stack when the program executes. Repeat the above process to dump addresses by passing a variable length of either %x or %p until you find the address on the stack.

In the example below I pass my ‘AAAA’ string followed by 10 %x format specifiers.



When I executed my format string much like last time the program output the ‘AAAA’ segment followed by addresses as they were pulled from the stack, the addresses read are as follows: 0xfffda4a4, 0xfffda4b0, 0xf7643ad, 0xf77ba3c4, 0xf77f2000, 0x804852b, 0x804a02c, and 0x8048520. The last two ‘0’, ‘0’ numbers are non valid addresses.

So what does this tell us? If you remember above the variable ‘secret’ is stored at address 0x0804a02c. The seventh address printed by our format string corresponds to the address of the variable secret. This tells us that the variable secret is stored at the seventh address on the stack. Using this piece of information we now know we need to write the needed value to bypass the conditional statement to the seventh address on the stack.

We can accomplish this by using another handy format specifier in conjunction with our format string. The ‘%#$x’ or ‘%#$p’ format specifier where ‘#’ is the position of the variable you’re trying to read / write on the stack allows us to specify the exact position on the stack we would like to read or write to. An example is below.

Screenshot - 080216 - 19:20:29.png

I passed ‘AAAA %7$p’ to the program and printf() output my ‘AAAA’ as well as the seventh argument on the stack as specified.

Now all that’s left to do is modify the value stored in the address corresponding to the variable ‘secret’ so we can pass the conditional statement. Luckily printf() has a very odd format string specifier ‘%n’. This format specifier means:

“*The number of characters written so far is stored into the integer indicated by the int \* (or variant) pointer argument. No argument is converted.”*

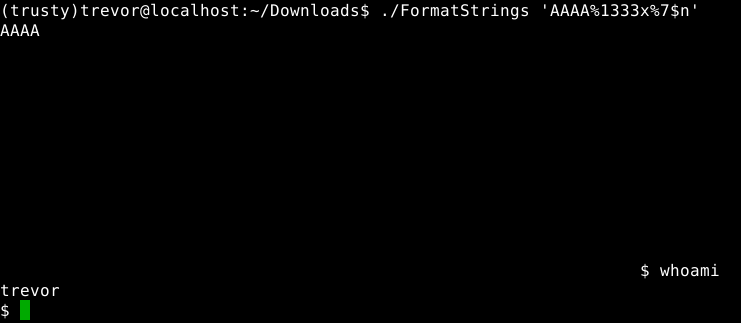
This means if we were to pass the string ‘AAAA%7$n’ we would write the decimal value 4 (each char is one byte) to the address containing our secret value. Using this trick we can write any value we wish to the address that contains our secret variable. However it seems like we would have to print an exorbitant amount of characters to write the needed value.

Fortunately we can bypass this pesky requirement by using a width specifier in our format string. if we pass ‘AAAA%100x’ to the program, 104 characters will be output. This is because %100x prints the argument padded to at least 100 characters. Therefore we could do the following ‘AAAA%<value-4>x%7$n’ to write an arbitrary value to the address 0x0804902c.

The only thing left to do now is execute our final format string. What’s the value we need to write again to pass the conditional statement?

Screenshot - 080216 - 19:43:39.png

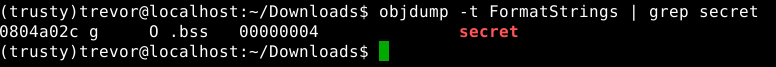
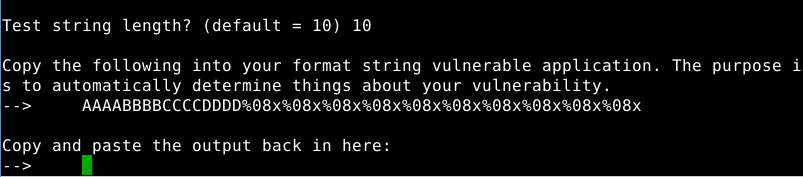
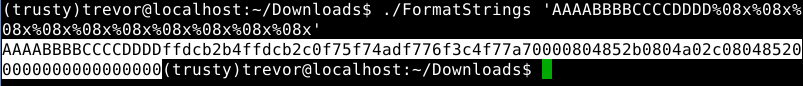
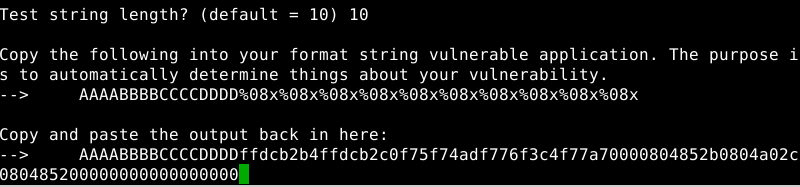
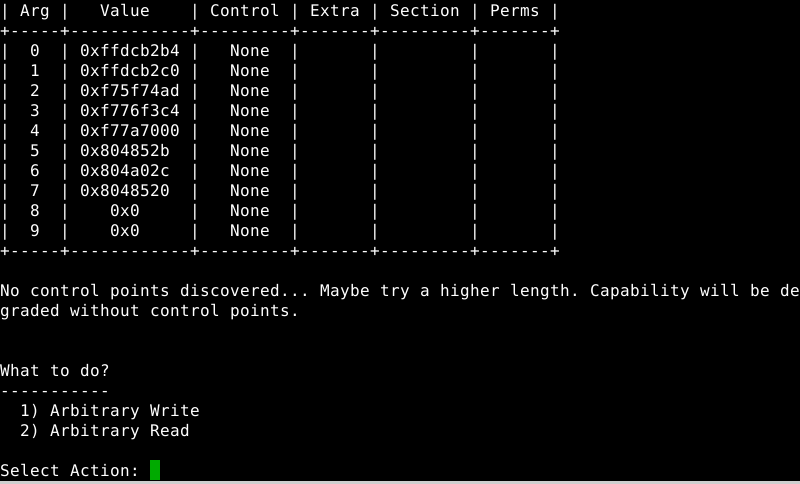
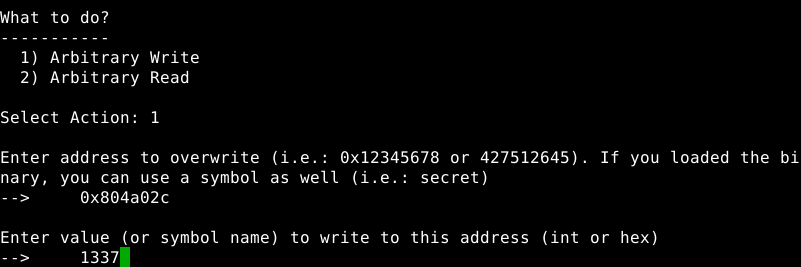
Oh right, 1337. We need to write 1337 into the address 0x0804902c. Let’s give it a shot.



Bingo! We’ve successfully written the desired value of 1337 into the variable containing ‘secret’ using a format string exploit and launched a new shell environment.

What if all of that was a little too much work though? Thankfully there’s a very nice tool called FormatStringExploiter we can use that will exploit format string vulnerabilities for us, and I’m going to show you how to use it! I’ve included the tool with the challenge package.

You too can be a 31337 hacker in 7 easy steps.

1. Using Objdump find the address of the variable you wish to read or write to.   
   
2. Launch the tool, specify the number of variables you would like to read off of the stack.   
   The tool will provide you with a generated format string to pass to the target program.   
   
3. Pass the generated format string to the target program, copy the output from the target program and paste the output from the target program back into the tool as requested.   
     
   
4. The tool will then output its findings based on the output of your target program. Then you will be prompted if you wish to perform an arbitrary read, or arbitrary write. Even though the tool reports no control points found we already know the address of the variable we want to write to.   
   
5. Select the option your wish, pass the address, and the value you want to write  
   
6. Paste the generated payload into your target program  
   Screenshot - 080216 - 20:14:47.png  
   Screenshot - 080216 - 20:15:49.png
7. Profit.

