Format String Vulnerabilities

February 26th, 2019

Today's Goals

- Introduce format string vulnerabilities

Announcements

- See https://sites.google.com/whitehatters.org/wcsc/announcements for CTFs we will be playing in this week

Reading Material

- Hacking: The Art of Exploitation
  - Available through USF Library online!
  - Live CD available on Starch Press website
  - Section 0x350 has a detailed overview of format string vulnerabilities
- (Most) original papers (I could find) on the topic
  - Format string vulnerability (Tue, 18 Jul 2000)
    - Kalou/Pascal Bouchareine
  - Exploiting Format String Vulnerabilities (September 1, 2001)
    - scut / team teso
- FormatGuard
  - Crispin Cowan was the lead author of this paper as well as StackGuard, which we haven't talked about yet but will soon (next week, actually)
- A Comparison of Techniques to Prevent Format String Attacks

Format Strings

What is a format string?

A format string uses a simple templating language to create complex strings based on provided arguments. The most notable instance of this is likely the printf function, which takes a format string and a variable number of arguments and prints the formatted string to stdout. Other programming languages also support format strings, including the handy format operator in Python (the % symbol). Let's take a look at this two.

```
#include<stdio.h>

int main()
{
	printf("I will print two formatted integers: %d %d\n", 100, 200);
}
```

```
print "I will also print two formatted integers: %d %d" % (100, 200)
```

In both of the above examples, the %d operators are replaced by the integers 100 and 200, respectively. Let's take a closer look at some other specifiers, then see how this works in C.

Format String specifiers

A great intro to the specifiers can be found here. However, this does not include a very useful field, the parameter field, that is available in POSIX's implementation of gcc. The wikipedia page for the printf format string is very helpful, and includes a good description of this field. Let's take a look at a few of this specifiers that we will want to know (parameter, width and type).

`%[parameter][flags][width][.precision][length]type`

Types

Specifier	Explanation	Example	Output
%d	Specifies an integer argument	printf("%d", 100)	100
%c	Specifies a char argument	printf("%c", 'A')	A
%x	Specifies an integer as hex	printf("%x", 100)	64
%s	Specifies a string	printf("%s", "Hello")	Hello
%p	Specifies a pointer	printf("%p", 0x4000)	0x4000
%n	Writes to an integer the number of characters output before the %n	printf("Hello%n", &i)	i = 5

Width

The width parameter sets the minimum length of the output. If a argument would be shorter than the width parameter, it is padded by spaces (or 0's with a flag).

```
printf("%10d", 1) //prints 9 spaces and a 1
printf("%010d", 1) //prints 0000000001
```

Parameter

The parameter option allows you to specify which argument to print instead of printing the next one. For example, you can print argument 2, 3, and 4 before you print argument 1.

```
printf("%1$d %2$d %3$d", 1, 2, 3) //prints 1 2 3
printf("%3$d %2$d %1$d", 1, 2, 3) //prints 3 2 1
```

An extremely useful tool when exploiting format string vulnerabilities. We will see how this makes things much easier later.

Variable Arguments in C

While discussing how to program with variable arguments is out of scope for this presentation, let's talk about what is actually happening in a small code segment adapted from here.

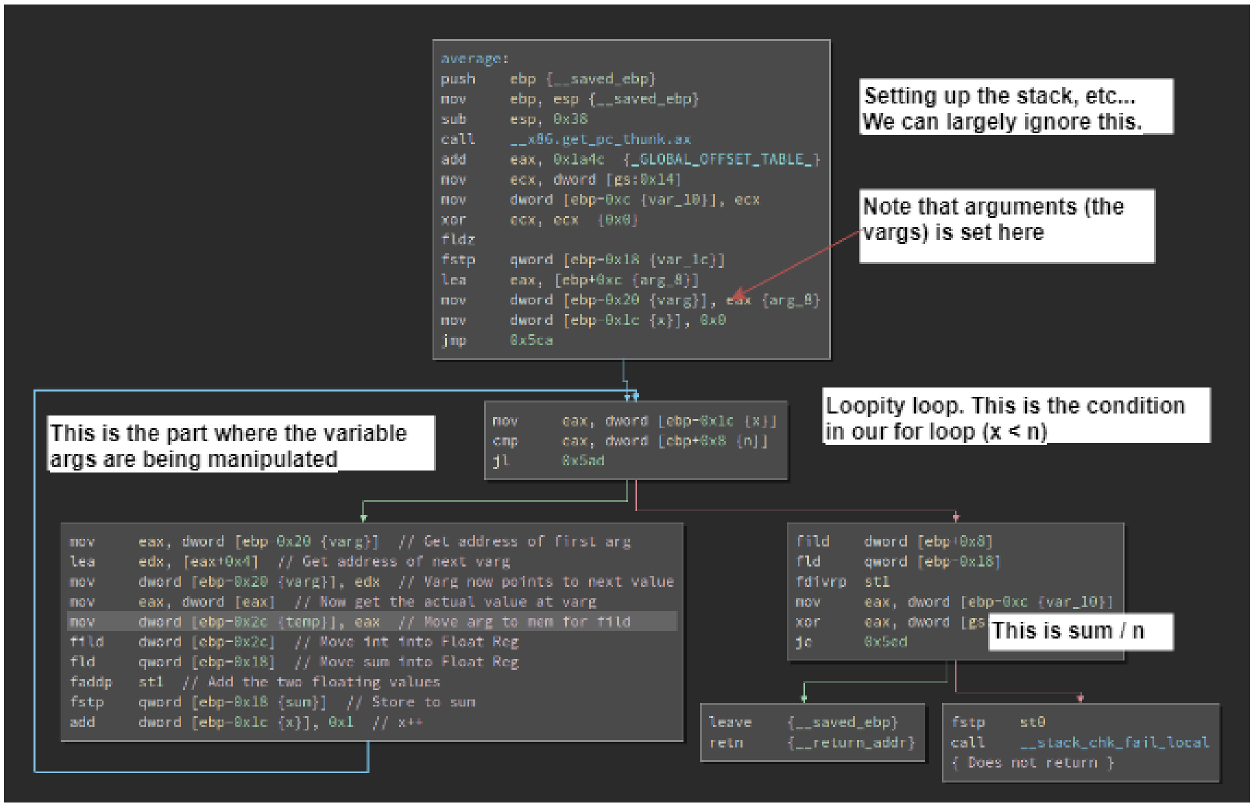
```
#include <stdio.h>
#include <stdarg.h>

// this function will take the number of values to average
// followed by all of the numbers to average
// Taken from https://www.cprogramming.com/tutorial/lesson17.html
double average ( int num, ... )
{
  va_list arguments;                    // A place to store the list of
arguments
  double sum = 0;

  va_start ( arguments, num );          // Initializing arguments to store
all values after num
  for ( int x = 0; x < num; x++ )       // Loop until all numbers are
added
```

```
    sum += va_arg ( arguments, int ); // Adds the next value in argument
list to sum.
  va_end ( arguments );                    // Cleans up the list

  return sum / num;                        // Returns the average
}

int main(int argc, char const *argv[])
{
      double avg1 = average(3, 100, 250, 275);
      double avg2 = average(5, 1, 2, 3, 4, 5);
      printf("Average of [%d, %d, %d] is %f\n", 100, 250, 275, avg1);
      printf("Average of [%d, %d, %d, %d, %d] is %f\n", 1, 2, 3, 4, 5,
avg2);
      return 0;
}
```

Alright, let's step through what's happening. Things to note:

- We had to tell vargs where to the arguments started (after num, the first argument)
- We have to specify what type each argument is (in this case, an int)
    - This is why printf can take so many different argument types in any order

What does this look like running?



```
average:
push    ebp {__saved_ebp}
mov     ebp, esp {__saved_ebp}
sub     esp, 0x38
call    __x86.get_pc_thunk.ax
add     eax, 0x1a4c  {_GLOBAL_OFFSET_TABLE_}
mov     ecx, dword [gs:0x14]
mov     dword [ebp-0xc {var_10}], ecx
xor     ecx, ecx  {0x0}
fldz
fstp    qword [ebp-0x18 {var_1c}]
lea     eax, [ebp+0xc {arg_8}]
mov     dword [ebp-0x20 {varg}], eax {arg_8}
mov     dword [ebp-0x1c {x}], 0x0
jmp     0x5ca
```

Setting up the stack, etc... We can largely ignore this.

Note that arguments (the vargs) is set here

```
mov     eax, dword [ebp-0x1c {x}]
cmp     eax, dword [ebp+0x8 {n}]
jl      0x5ad
```

Loopity loop. This is the condition in our for loop (x < n)

This is the part where the variable args are being manipulated

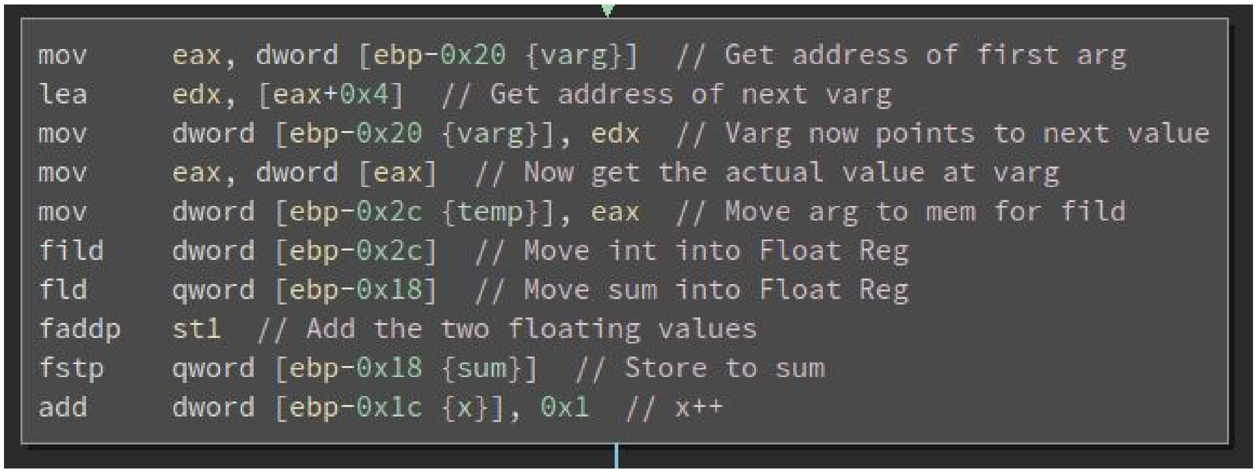
```
mov     eax, dword [ebp-0x20 {varg}]  // Get address of first arg
lea     edx, [eax+0x4]  // Get address of next varg
mov     dword [ebp-0x20 {varg}], edx  // Varg now points to next value
mov     eax, dword [eax]  // Now get the actual value at varg
mov     dword [ebp-0x2c {temp}], eax  // Move arg to mem for fild
fild    dword [ebp-0x2c]  // Move int into Float Reg
fld     qword [ebp-0x18]  // Move sum into Float Reg
faddp   st1  // Add the two floating values
fstp    qword [ebp-0x18 {sum}]  // Store to sum
add     dword [ebp-0x1c {x}], 0x1  // x++
```

```
fild    dword [ebp+0x8]
fld     qword [ebp-0x18]
fdivrp  st1
mov     eax, dword [ebp-0xc {var_10}]
xor     eax, dword [gs
jc      0x5ed
```

This is sum / n

```
leave   {__saved_ebp}
retn    {__return_addr}
```

```
fstp    st0
call    __stack_chk_fail_local
{ Does not return }
```



```
mov     eax, dword [ebp-0x20 {varg}]  // Get address of first arg
lea     edx, [eax+0x4]  // Get address of next varg
mov     dword [ebp-0x20 {varg}], edx  // Varg now points to next value
mov     eax, dword [eax]  // Now get the actual value at varg
mov     dword [ebp-0x2c {temp}], eax  // Move arg to mem for fild
fild    dword [ebp-0x2c]  // Move int into Float Reg
fld     qword [ebp-0x18]  // Move sum into Float Reg
faddp   st1  // Add the two floating values
fstp    qword [ebp-0x18 {sum}]  // Store to sum
add     dword [ebp-0x1c {x}], 0x1  // x++
```

Let’s walk through this together on the board, drawing the stack and variables for {100, 200, 300}.

Some notes:

- fild—Push int onto the FPU register stack
- fld—Push float onto the FPU register stack
- faddp—Add ST(0) to ST(1), store result in ST(1), and pop the register stack.
- fstp—Copy ST(0) to m32fp and pop register stack.

- Comments in Binary Ninja! A very nice feature. You can even save and return to your work in the paid version (I don't think that's a demo feature)

Moral of the story: vargs are just fancy pointers (isn't all of C just fancy pointers, though?)

Format String Vulnerabilities

Alright, now that we have a pretty good idea how printf works, what does a format string vulnerability look like?

What would happen if, instead of passing a format string as the first argument to printf, we passed user input? For example:

```c
void vuln()
{
      char in[SIZE];

      fgets(in, SIZE - 1, stdin);

      printf(in);
}
```

A malicious user can input their own format strings! Why is this a big deal?

What does the %n type do?

By using the %n type, and a bit of stack magic, we can write to arbitrary values in memory! Let's test this out.

1. Run ./format_vuln
2. Pass in a normal string
3. Pass in a %x
4. Pass in a %n… Whoops
5. Now try a %4$x
    a. What's happening here?
    b. The fourth value on the stack is being printed!
6. Now try %1$n through %7$n
    a. This is tedious to write out, so I wrote a quick python script to do it
7. Huh… Argument 7 looks suspicious
    a. This is the address of i on the stack!
8. Now, let's try our %7$n

Arbitrary Writes

Let's try to write an arbitrary value, then see how to write to an aribitrary address!

Writing the right value to the return address

How do we write a specific value? Well, what %n output? The number of printed characters! So, if we want to write x to the integer, we need to write x bytes! We can use the width modifier to do this efficiently!

1. We want to overwrite the address to be the ox1337l
2. Cool, so we need to print… 0x1337 = 4919 bytes
3. We can do that with %4919x%7$n

Writing to an arbitrary address

Unfortunately, not every address we want to write to is on the stack. However, there is something important on the stack that we control. What is it?

Our string is actually sitting on the stack! Let's experiment:

1. Enter AAAA followed by several %p (up to 11)
    a. I'll use my script again
    b. Check out the tenth %p
2. Argument 11 is actually our AAAA (or 0x41414141) on the stack
3. So, we can place an address in front of the format string, then give it arg 10 to write to our address
4. See the python script to see how we do this

Advanced topics

Below shortly describes some advanced topics. Read Hacking: The Art of Exploitation for more info!

Large values

Well, all is good. We can write bytes to memory! But what if we want to write a really large value? Printing 0xdeadbeef characters is going to take a really long time, if it even works! To get around this, instead of writing all at once, we will write 0xbeef, then write 0xdead. Let's take a look at the python script. Note we use %hn to write a half-word, or 2 bytes, at a time.

So, we wrote 0xbeef characters, then wrote 0xdead-0xbeef characters. Pretty neat!

Large values, but higher address < lower address

Well, this is problematic. Let's say we want to write 0xbeefdead. We write 0xdead, and then we would need to write 0xbeef - 0xdead characters! This means we need to print negative characters! Oh no!

We can fix this using wrap around. We first write 0xdead, then write 0xfbeef - 0xdead characters. Then, when we write our half word, it will drop off the 0xf0000 to become 0xbeef! Nice.

Arbitrary Code Execution

It's hard to overwrite the return address like in a buffer overflow. Instead, we overwrite a entry in the Global Offset Table, and point that at our code. When the function is called, it will execute what we pointed it to. Check it out in python_sol.py

Note that this has been thwarted by RELRO, or RELocation Read-Only.

Format Guard

Why is it not widely deployed?

I could not find the original implementation, nor find any systems that actively implement FormatGuard. I have not seen any reports that discuss why, but here is my thoughts on why this is the case:

- It can't be the overhead…
    - The overhead is very low according to the StackGuard paper (37% overhead on an individual printf, and 1.3% overhead for a program that uses printf extensively)
    - Stackguard has been employed on almost all systems, and has an (upper bound of) 125% overhead on void function calls!
        - The overhead is less as you introduce more arguments (since StackGuard is a static 7 instructions)

Let's take a look at Section 1.1 in Exploiting Format String Vulnerabilities. Most notably, the visibility: format strings are rated as "easy to find" while buffer overflows are "sometimes very difficult to spot". This is my guess why StackGuard isn't widely deployed: with the introduction of the '-Wformat' warning in GCC, printf vulnerabilities/bugs are incredibly easy to solve. In addition, FormatGuard requires a modified glibc and requires the source program to be recompiled with the modified glibc; running a FormatGuard program on a system without the modified glibc would not work. Contrast this with StackGuard, which simply needs to recompile the source code. Check out the compatibility testing section of the FormatGuard paper for even concerns.

Breaking FormatGuard?

The paper describes three security limitations, of which the first is the most interesting. In this scenario, an attacker uses **less or the same** amount of arguments as StackGuard expects. I have an idea for a CTF challenge based off this idea, so watch out for that (I will probably have to reimplement it myself… It seems FormatGuard is quite dead, and I can't find anything for it anywhere)!