Forum:Hack Forums

Thread: [Tutorial] ~ Format String Exploitation ~ Understanding The Stack

Goodies - *May 18, 2014, 2:27 pm*

Hello everyone!

It's been a while since I have been able to write tutorials. This tutorial will be about format string vulnerabilities. Though they are not quite as common as some buffer overflows or null pointer dereferences, there have been several reliable exploits using format string vulnerabilities.

What is a "format string vulnerability"?
As any beginner C programmer will be able to tell you, format strings allow you to print variables in the form of strings. Various functions allow you to perform specific actions to or with the strings. For example:
printf - Prints a string on screen using stdout file descriptor stream.

fprintf - Allows you to choose to which stream you want to send it. Either the defaults (stdin, stdout, stderr) or one you have established in the program.

sprintf - Copies a formatted string into a string variable

Examples:
PHP Code
<?php

```
#include <stdio.h> // for all format string functions
#include <stdlib.h> // for dynamic memory allocation
int main(int argc, char **argv){
    printf("This is a test using %s\n", "printf"); // printf uses stdout
    fprintf(stderr, "This is a test error message using %s\n", "fprintf"); // printing to stderr file stream
    char *str = calloc(30, sizeof(char)); // dynamically allocate 30 bytes
    sprintf(str, "I'm writing to 'str' using %s!\n", "sprintf"); // write a string to the 'str' string.
    printf("%s", str); // print the 'str' string
    return 0;
}
```

How do format strings work?
From here on, I'll be referring to the format string function printf() more so than any other function. Just realize that I am, in fact, talking about the other string functions as well. That being said, I'd like to explain how these format string functions work. After all, since these are all executed in runtime (IE: this is has nothing to do with the pre-processor), there must be something happening during the execution program.
When executing a printf() function, the parameters that will be printed out are PUSHed onto the stack. The format string will then be interpreted and POP specific parameters from the stack and read them into stdout file stream (which are interpreted as ASCII strings). For example:
%s - This takes a pointer to a string. Remember that a string is an array of characters terminated by a NULL character (\x00)

%c - takes only one byte as a char is only one byte in size

%x - Takes an integer and prints it out in hexadecimal representation

%p - rather than printing the value, it will print out the pointer. A memory address must be passed for this to work correctly.

etc... There are more, but they are not pertinent to this tutorial.

Taking a look:
Let's take a look at a file vulnerable to format string vulnerabilities:
Compiled with:
gcc fmtstr.c -mpreferred-stack-boundary=2 -m32 -ggdb -o fmtstr
PHP Code

```
<?php
#include <stdio.h>

int main(int argc, char **argv){
    if(argc!=2) return 1;
    char *str1 = "Test string #1\n";
    char *str2 = "Test string #2\n";
    char *str3 = "Test string #3\n";
    printf(*(argv+1));
    return 0;
}
```

This is all well and good, but what does this mean? This program seems completely innocent? First, it will declare 3 separate strings. Then, it will print whatever the user passes into the program. How is this harmful?

Well, since the first parameter in the printf()function is the format string, nothing is stopping the user from supplying his own format escape strings.

Say the user supplies something like (command and output):
./fmtstr "This is Goodies!"
This is Goodies!

This is exactly what we had expected, right? Well, what if the user supplied, say, a format string:

./fmtstr "Goodies says '%s'"
Goodies says 'Test string #3
'

This is definitely not the output we expected! What happened?

Well, in a format string, passing the %s takes a pointer off of the stack and will read it as an ASCII string until it hits a NULL byte. Since the local variables str1, str2, and str3 are declared on the stack (and in that order, too), they are popped off in reverse order (remember, the stack is First In Last Out) and interpreted as a string.

Before we get ahead of ourselves, read this comment that I had originally written inside the file, but

in order to keep it small for HF, I decided to separate it :)
PHP Code
<?php
/*
* Let's talk about the difference between *str and str[]. In *str, the characters
* are declared elsewhere in memory and only a pointer to the first character is
* pushed onto the stack. In str[], the entire value of the string is saved on the
* stack and printing it out will not work. Not in string form anyway since %s can
* only take a pointer to a string.
*/

Debugging for a closer look:
Before we get ahead of ourselves, let's take a look at how the stack grows. I compiled it with -ggdb so I can more easily debug it, but it's definitely not required!

gdb -q ./fmtstr
(gdb)

Alright, fine. We are in Gnu Debugger :) If you are like me and prefer Intel ASM syntax over AT&T syntax, run:
(gdb) set disassembly-flavor intel

With that, go ahead and disassemble the main function:
(gdb) disassemble main
PHP Code
<?php

```
Dump of assembler code for function main:
   0x004016b0 <+0>:     push   ebp
   0x004016b1 <+1>:     mov    ebp,esp
   0x004016b3 <+3>:     sub    esp,0x10
   0x004016b6 <+6>:     call   0x401c80 <__main>
   0x004016bb <+11>:    cmp    DWORD PTR [ebp+0x8],0x2
   0x004016bf <+15>:    je     0x4016c8 <main+24>
   0x004016c1 <+17>:    mov    eax,0x1
   0x004016c6 <+22>:    jmp    0x4016f2 <main+66>
   0x004016c8 <+24>:    mov    DWORD PTR [ebp-0x4],0x405064
   0x004016cf <+31>:    mov    DWORD PTR [ebp-0x8],0x405074
   0x004016d6 <+38>:    mov    DWORD PTR [ebp-0xc],0x405084
   0x004016dd <+45>:    mov    eax,DWORD PTR [ebp+0xc]
   0x004016e0 <+48>:    add    eax,0x4
   0x004016e3 <+51>:    mov    eax,DWORD PTR [eax]
   0x004016e5 <+53>:    mov    DWORD PTR [esp],eax
   0x004016e8 <+56>:    call   0x403688 <printf>
   0x004016ed <+61>:    mov    eax,0x0
   0x004016f2 <+66>:    leave
   0x004016f3 <+67>:    ret
End of assembler dump.
```

From this alone, we can easily see the addresses of the strings. If you so chose, you can see these strings:

PHP Code
<?php

```
mov    DWORD PTR [ebp-0x4],0x405064 //str1
mov    DWORD PTR [ebp-0x8],0x405074 //str2
mov    DWORD PTR [ebp-0xc],0x405084 //str3
```

The memory addresses are in reverse order because the stack grows towards lower addresses, but local variables grow towards higher addresses.

Jest to test this out, let's extract 3 strings starting from the address 0x405064:

```
(gdb) x/3s 0x405064
```

PHP Code

```
<?php
0x405064 <__register_frame_info+4214884>:      "Test string #1\n"
0x405074 <__register_frame_info+4214900>:      "Test string #2\n"
0x405084 <__register_frame_info+4214916>:      "Test string #3\n"
```

Okay, great! Before we continue, let's set a breakpoint in the program so we can dump the registers and make alterations as needed. The best place for a breakpoint in this case, would be right at the printf() statement which is at 0x004016e8. After that, let's execute it and it will stop at the printf() function.

```
(gbd) break *0x004016e8
(gdb) run "Testing123"
```

Aaaaannnnddd we hit our break point. Let's look at our base pointer:

```
(gdb) x/8xw $ebp
```

Here is my output:

PHP Code

```
<?php
0x22ff28:      0x0022ffa0      0x00401413      0x0000002      0x003e24b8
0x22ff38:      0x003e2a48      0x0000002      0x000000      0x0000003
```

Well, we know EBP (That first address: 0x0022ffa0) is the base pointer and points the the beginning of the current stack frame. In our case, we are in the main() stack frame. That being said, we know the following address is the return address of our main function. Since I'm currently confined to a windows machine, my return address is in "__mingw_CRTStartup". In a Unix machine, it is different.

Now, let's dump the ESP which points to the top of the stack.

```
(gdb) x/8xw $esp
```

PHP Code

```
<?php
0x22ff18:      0x003e2490      0x00405084      0x00405074      0x00405064
0x22ff28:      0x0022ffa0      0x00401413      0x0000002      0x003e24b8
```

If you'd like, go ahead and get the string output of each of those (first 4. We know the first address in the second row is the base pointer).

```
(gdb) x/1s 0x003e2490
0x3e2490:      "Testing123"
(gdb) x/1s 0x00405084
0x405084 <__register_frame_info+4214916>:      "Test string #3\n"
```

(gdb) x/1s 0x00405074
0x405074 <__register_frame_info+4214900>: "Test string #2\n"
(gdb) x/1s 0x00405064
0x405064 <__register_frame_info+4214884>: "Test string #1\n"

"Testing123" is not actually the string at the address of the parameter that we provided. It is actually the format string itself. If you'd like to change the source code to something like:
PHP Code
<?php
printf("This is %s", "a test");
You will see that the first address pointed to by ESP is the format string address. The reason it is the same as the first parameter argument is because the format string, in our case, IS the command line argument.

The second through fourth strings are the test strings in, of course, reverse order as they are declared that way (local variables: lower memory addresses first, rise towards higher memory addresses in the stack frame).

(gdb) continue

Okay, so what EXACTLY is going on here? When the printf statement is called, it will find any arguments needed in the printf format string (%s, %x, %d, etc) and take the next memory address from the stack. In our case, it is interpreted as a string. This is all very neat, but what the hell can I use it for? Just to break a program or read a stored string??
Let's continue.

Viewing the Stack Contents
Viewing the stack contents is very easy in format string vulnerabilities. We're simply going to pass a %x which will pop the next address off of the stack and print it out. This is all well and good, but let's see it in action.
Tutorial written by Goodies from HackForums.net
First, let's dump the stack output for, oh, 12 words. This is equivalent to 12 %08x's.

If you cleared the breakpoint or exited the program, you will need to add it again.
Run the program with
(gdb) run "%08x %0x8 %08x %08x %08x %0x8 %08x %08x %08x %0x8 %08x %08x"

Now that we hit the breakpoint, before we continue, let's dump the ESP again using:
(gdb) x/12xw $esp
Dump GDB:
PHP Code
<?php
0x003e2490 0x00405084 0x00405074 0x00405064
0x0022ffa0 0x00401413 0x0000002 0x003e26f8
0x003e2a48 0x0000002 0x000000 0x0000002

Program output:Tutorial written by Goodies from HackForums.net
PHP Code
<?php
00405084 4050748 00405064 0022ffa0 00401413
00000028 003e26f8 003e2a48 0000002 0000008
0000002 000000
Okay, great... we can see that the stack identically matches the program's output. Wait, identically? I

see a spot where they don't match! Well, that, as we already established, is the frame pointer which is dynamic and subject to change as our program executes.

Upon reading the stack, we can use this information many ways. We can examine it, read it in the debugger, etc...

Writing contents to stack
This is going to be very difficult for me as I am inside windows and the gdb console with mingw does not support python scripting built in. Much of this section will be proof of concept and/or theoretical, however, I will do my best :)

Note: You may want to turn off ASLR when first doing this until you understand what's going on!

There is one format string that I have not gone over and that is the %n tool. %n is very interesting because it will WRITE to a memory address as opposed to READ from it.

%n - writes the number of bytes printed to the address provided.

Here is a practical example:
PHP Code
```
<?php
int i = 0;
int j = 0;
printf("ABCD%n1234%n", &i, &j);
/* At this point, 'i' is equal to 4 since 4 characters had been printed. 'j'
 * would be equal to 8 since 8 characters had been printed. Our goal is to
 * write the memory address of, say, shellcode or a ROP gadget (which
 * depends on our method of attack). This can be done using %n!
 */
```

Remember our return memory address? If you scroll back up, you'll see that it is 0x00401413. This can be shown as the integer 4,199,443. This is quite large!

Let's just say, for arguments sake, that our shellcode is somewhere in our program at the address 0x00401555. This is a totally made up address and may or may not even be valid inside the scope of our program. Anyways, we want to overwrite our return address with 0x00401555 which, in decimal, equates to: 4,199,765. We would need to print out exactly that many characters and pop off that many places in stack. How hard could it be?

First, let's find how many until the return address. We know what the return address is, so let's use %x until we find it.

```
(gdb) run "%x %x %x %x %x %x %x"
405084 405074 405064 22ffa0 401413 2 3e24c0
```

Our return address is the 5th one which means popping 4 off of the stack would be perfect. Let's see exactly how many characters it would be:

```
(gdb) run "%x%x%x%x"
```
405084405074405064 22ffa0 = 24 characters. Since our return address is 4,199,765 in decimal form, we have a long way to go :)

If I were in GDB in linux and could use python scripting, I would do something along the lines of:

(gdb) run $(python -c 'print "%x%x%x%x" + "A"*(4199765-24)+"%n"')

Which would, in theory, cause us to return to 0x00401555!

Conclusion
Even if you never use a format string vulnerability in an actual system security test, they have been known to be very severe. For example screen, a program in Linux that multiplexes one terminal for several simultaneous processes, had a format sting vulnerability that lead to a local root privilege escalation! Novell had a format string vulnerability for several versions and decided that it was benign enough to ignore. After someone showed just how detrimental it was, they had it fixed and a patch was released within 3 days.

The most important information is this thread is the understanding of how the stack grows, how local variables are stored, and how memory works. Format string vulnerabilities are but one of many ways to exploit programs, but the success of all of them are contingent upon your understanding of the stack and memory process!

M3ga. - *May 18, 2014, 3:46 pm*
Good tutorial but you're missing lots of info..

Goodies - *May 18, 2014, 4:07 pm*
(May 18, 2014 03:46 PM)M3ga. Wrote: [CITE] Good tutorial but you're missing lots of info..
As much as I appreciate the post, why don't you actually contribute instead of simply trying to get 25 posts so you can PM? I'm happy yo help, but unless you actually contribute, I'm going to completely disregard you and your post.

Thanks.

RopMyCock - *May 18, 2014, 4:07 pm*
Really, I don't see this being needed, as Levelup and Divine already did it, but, I guess it's diverse from the usual shit here.

Actually, after reading it over a second time, I think you were more detailed and focused on the more important things, which is great. Good tutorial

mudasir007 - *May 18, 2014, 5:44 pm*
Nice tutorial,Keep it up, Good work

Goodies - *May 18, 2014, 7:22 pm*
(May 18, 2014 04:07 PM)RopMyCock Wrote: [CITE] Really, I don't see this being needed, as Levelup and Divine already did it, but, I guess it's diverse from the usual shit here.

Actually, after reading it over a second time, I think you were more detailed and focused on the more important things, which is great. Good tutorial
Wow. I honestly have never seem his tutorial before. However, like his other tutorials, I was able to find identical source codes online in tutorials he copied. Hmm.

SHADOW REALM - *May 19, 2014, 5:36 am*
(May 18, 2014 07:22 PM)Goodies Wrote: [CITE] Wow. I honestly have never seem his tutorial before. However, like his other tutorials, I was able to find identical source codes online in tutorials he copied. Hmm.
divine or levelup?

That reminds me I haven't talked to him in ages

RopMyCock - *May 19, 2014, 5:37 am*
(May 19, 2014 05:36 AM)SHADOW REALM Wrote: [CITE] divine or levelup?

That reminds me I haven't talked to him in ages
He's talking about Divine m8. Apparently some of his tutorials were copied but I can't say for certain

Goodies - *May 19, 2014, 5:13 pm*
Why did people rate this 1 star? Do you have any idea how long these tutorials take me?

RopMyCock - *May 19, 2014, 5:50 pm*
(May 19, 2014 05:13 PM)Goodies Wrote: [CITE] Why did people rate this 1 star? Do you have any idea how long these tutorials take me?
Tried 5 starring but then remembered reg members can't, which is bullshit.

DrDinosaur - *May 19, 2014, 6:32 pm*
(May 19, 2014 05:13 PM)Goodies Wrote: [CITE] Why did people rate this 1 star? Do you have any idea how long these tutorials take me?
Faggo.ts will always remain faggo.ts. faggot lol

A32 - *May 20, 2014, 12:14 am*
Thanks bro. Now, is it wrong to say that BOF with this method is highly unpractical? Writing that million bytes to, say, a remote program will take a long time. Unless you can fish some juicy info (say, passwords) from the memory with %x%x%x%x, it ain't worth it.
Correct me if I'm wrong :D

LevelUp - *May 20, 2014, 1:03 pm*
Really confused as to why this doesn't include something as basic as direct parameter access?

(May 20, 2014 12:14 AM)A32 Wrote: [CITE] Thanks bro. Now, is it wrong to say that BOF with this method is highly unpractical? Writing that million bytes to, say, a remote program will take a long time. Unless you can fish some juicy info (say, passwords) from the memory with %x%x%x%x, it ain't worth it.
Correct me if I'm wrong :D
It's not useless, eg: Pwn the .dtors section's end address -> -> /bin/sh?

RopMyCock - *May 20, 2014, 2:07 pm*
(May 20, 2014 01:03 PM)LevelUp Wrote: [CITE] Really confused as to why this doesn't include something as basic as direct parameter access?

It's not useless, eg: Pwn the .dtors section's end address -> -> /bin/sh?
What's direct parameter access? ?

LevelUp - *May 20, 2014, 2:11 pm*
(May 20, 2014 02:07 PM)RopMyCock Wrote: [CITE] What's direct parameter access? ?
Instead of popping x 8 times with
%x%x%x%x%x%x%x%x
Read the 8th parameter with
%8$x

RopMyCock - *May 20, 2014, 2:21 pm*
(May 20, 2014 02:11 PM)LevelUp Wrote: [CITE] Instead of popping x 8 times with

%x%x%x%x%x%x%x%x
Read the 8th parameter with
%8$x
That does look a lot cleaner. I assume it's just to fasten up the process

LevelUp - *May 20, 2014, 2:24 pm*
(May 20, 2014 02:21 PM)RopMyCock Wrote: [CITE] That does look a lot cleaner. I assume it's just to fasten up the process
You don't have to pop args from the stack, you just read directly, thus the name.

Goodies - *May 20, 2014, 3:24 pm*
(May 20, 2014 02:11 PM)LevelUp Wrote: [CITE] Instead of popping x 8 times with
%x%x%x%x%x%x%x%x
Read the 8th parameter with
%8$x
Yes, I'm aware of that :) However, to my knowledge, it doesn't work in MinGW because I did try it and didn't get the response I was expecting. It should work on other unix and posix boxes.

LevelUp - *May 20, 2014, 3:34 pm*
(May 20, 2014 03:24 PM)Goodies Wrote: [CITE] Yes, I'm aware of that :) However, to my knowledge, it doesn't work in MinGW because I did try it and didn't get the response I was expecting. It should work on other unix and posix boxes.
Have you tried gnu_printf or some other posix compliant implementation?

Goodies - *May 20, 2014, 3:51 pm*
Well, wouldn't overwriting .dtors only be effective if there is a destructor? (Which there is obviously not one specified in my file)? Or are they always mapped?

I know for a fact .ctors and .dtors are writable (by default), but does there existence dependant on there being a constructor/destructor?

(May 20, 2014 03:34 PM)LevelUp Wrote: [CITE] Have you tried gnu_printf or some other posix compliant implementation?
I have not (nor do I plan on it tbh). The only posix machine I own is an iPod touch and that's plenty.

LevelUp - *May 20, 2014, 4:09 pm*
(May 20, 2014 03:51 PM)Goodies Wrote: [CITE] Well, wouldn't overwriting .dtors only be effective if there is a destructor? (Which there is obviously not one specified in my file)? Or are they always mapped?

I know for a fact .ctors and .dtors are writable (by default), but does there existence dependant on there being a constructor/destructor?

I have not (nor do I plan on it tbh). The only posix machine I own is an iPod touch and that's plenty.
I was talking about programs (c++ specifically) in general. The quickest way to lash one into your code here would be to define an attribute.

mr. smith - *May 20, 2014, 4:11 pm*
this is a very nice tutorial!! thanks for your effort!

iDuL - *May 21, 2014, 4:13 pm*
Your threads deserve to be stickied, Goodies.

Great!