IOActive™
COMPREHENSIVE COMPUTER SECURITY SERVICES

printf() tricks

DC4420 slides, Feb 2012

Who am I?

- Shaun Colley
- Senior Security Consultant at IOActive
- Exploit development, reverse engineering, development, pen-testing

Printf() tricks - Agenda

- Shifting the stack pointer & arbitrary mem writes…
  - … in order to exploit format string bugs without %n

- When is a NULL pointer not just a NULL pointer?
  - …don’t expect printf() & family to crash on NULL pointers

- These are just a few things I played with a while back

- There was a good article in Phrack #67 called "A Eulogy for format strings" (phrack.org/issues.html?issue=67&id=9) by Captain Planet
- Main point of the article was disabling the anti-format string bug exploitation measures implemented by the FORTIFY_SOURCE patch (gcc prog.c –o prog -D_FORTIFY_SOURCE=2)

- The patch's anti-exploit measures are:
  - Detect 'holes' in direct parameter access, i.e. %16$x and not %16$x %15$x %14$x … %1$x
  - Detect %n in format strings that are in writable segments (stack, heap, BSS, …)
  - Both of these result in an abort()

- How did the author, Captain Planet disable FORTIFY_SOURCE?
- Need to look into the GLIBC vfprintf.c source code...
- Warning – it’s not pretty. In fact understanding the code is more of a reverse engineering job than just reading C code ☺

See code on next slide...

```
    args_type = alloca (nargs * sizeof (int)); // !!! UNBOUNDED ALLOCA = STACK
    SHIFTING !!!
    memset (args_type, s->_flags2 & _IO_FLAGS2_FORTIFY ? '\xff' : '\0',
            nargs * sizeof (int));

args_value = alloca (nargs * sizeof (union printf_arg));  // !! UNBOUNDED STACK
SHIFTING !!!

    /* XXX Could do sanity check here: If any element in ARGS_TYPE is
       still zero after this loop, format is invalid.  For now we
       simply use 0 as the value.  */

    /* Fill in the types of all the arguments.  */
    for (cnt = 0; cnt < nspecs; ++cnt)
      {
        /* If the width is determined by an argument this is an int.  */
        if (specs[cnt].width_arg != -1)

args_type[specs[cnt].width_arg] = PA_INT;  // UNBOUNDED NULL DWORD WRITE
```

- Nargs = maximum possible number of format args, i.e. %10$x %12345$x would give nargs = 12345

- And specs[cnt].width_arg = width of currently parsing format specifier

- So args_type[specs[cnt].width_arg] = PA_INT; can ultimately lead to an (almost-)arbitrary addr NULL DWORD write

- This allowed the author to toggle off the _IO_FLAGS2_FORTIFY flag in the file stream being used.
- Very important point to note is that nargs was set to something that would wrap to 0 in the memset, i.e. %1073741824$
- And then another format specifier was used to exploit

args_type[specs[cnt].width_arg] = PA_INT

- If width_arg is chosen very carefully the FORTIFY_SOURCE flag in the file stream is NULLed.
- At this point you can use direct parameter access + %n’s to carry out a fairly standard format string attack

- Cool, patch bypassed…

- But are there any other ways to exploit this arbitrary stack pointer shift and/or arbitrary NULL dword write?
  - For example, without later having to use %n like in normal format string exploits?

- Yes, but they're fairly application-specific. Let's consider each of the attack vectors - 1) stack shifting and 2) arbitrary address write (not arbitrary value)

- Stack pointer shifting with alloca()…

- Few different possibilities. Firstly you could use a large DPA to shift the stack pointer into the heap:

```
args_type = alloca (nargs * sizeof (int)); // !!! UNBOUNDED ALLOCA = STACK SHIFTING !!!
    memset (args_type, s->_flags2 & _IO_FLAGS2_FORTIFY ? '\xff' : '\0',
            nargs * sizeof (int));
```

- However you'll generally get a SIGSEGV because of the memset()

- Sometimes this doesn’t matter
- The memset has still corrupted memory up to the point a guard page is hit…
- We just need some of this memory to be used in a SEGV signal handler
- i.e. SEGV signal handler tries to drop privileges to do something priv-sensitive but the saved UID has been overwritten with 0’s…
- Could be pretty bad news.
- Demo (on a VM!!)

- What if there is no signal handler and a seg fault in memset() will just crash the app?
- Sometimes we may be able to work it so that nargs * sizeof(int) at [1] is small enough that no page fault happens at memset()…

```
args_type = alloca (nargs * sizeof (int)); // !!! [1] UNBOUNDED
ALLOCA = STACK SHIFTING !!!
    memset (args_type, s->_flags2 & _IO_FLAGS2_FORTIFY ?
'\xff' : '\0',nargs * sizeof (int));

args_value = alloca (nargs * sizeof (union printf_arg)); // [2]
```

- Yet at the same time we make nargs * sizeof(union printf_arg) is large enough to shift the stack pointer past the guard page and into the heap

- So we use a %<number>$x with number small enough that <number> * sizeof(int) still leaves ESP in the stack therefore the memset() doesn’t page fault…
- ..Then the next alloca() with no annoying memset() shifts the stack pointer past the guard page and into an area of memory we (in/)directly control i.e. heap
- Any further function calls after this point will push stack frames into this memory area
- What if another (p)thread then clobbers this area with data we control?
- You’ve potentially got an exploitable vector…and you didn’t even use a %n specifier
- You just need to find somewhere you can shift to that you have some control over

- Can be a little messy
- Often need to play around with rlimits and get a lot of heap malloc()’ed

- Demo…

- What about using the arbitrary NULL overwrite for something?
- Again, application-specific just like the first demo

- Could be used to zero out some context-specific int like Captain Planet used to zero out the FORTIFY_SOURCE flag
- There are these assignment ops as well:

```
args_type[specs[cnt].data_arg] = specs[cnt].data_arg_type;
  break;
default:
  /* We have more than one argument for this format spec.
     We must call the arginfo function again to determine

              __printf_arginfo_table[specs[cnt].info.spec])
       specs[cnt].info,
     specs[cnt].ndata_args, &args_type[specs[cnt].data_arg]);
```

- Be imaginative and do some digging – there may be something you can overwrite that will be enough to affect execution flow in your favour
  - Application-specific privilege flags
  - Loop counters
    - i.e. overwrite a decrementing loop counter with zero, then…
      - counter--; → 0xffffffff
      - Could lead to memory corruption
  - …

- Lastly, be aware that printf(“abcd %s\n”, NULL) does not necessarily crash at a NULL pointer dereference
- According to C99, the behavior is actually undefined
- But glibc’s *printf() and other implementations will replace such an occurance with “(null)” (not always, sometimes it will seg fault – it depends what else is in the format string)
- i.e.

```
root@bt:~# ./null
abcd (null)
```

- Potential to be abused?
- Again, application-specific but could lead to an overflow in sprintf() if ptr was supposed to point to a string shorter than strlen(“(null)”) = 6 bytes.
- i.e. char *ptr = NULL;
-

```
      switch(user_controlled_int) {
                case 0 :  ptr = “AB1”;
                          break;


                case 1 : ptr = “AB2”;
                          break;
                case 2 : ptr = “AB3”;
                  break;
          }
             sprintf(buf, “abcd %s”, ptr);  // could be an overflow
```

- Just some *printf() internals/tricks I thought might be interesting.
- Thanks for listening.

Questions?

IOActive, Ltd

www.ioactive.co.uk

scolley@ioactive.co.uk