Year and Semester:	2013 FALL
Course Number:	CS-336
Course Title:	Intro. to Information Assurance
Work Number:	LA-02
Work Name:	Format-String Vulnerability
Work Version:	Version 1
Long Date:	Sunday, 13 October 2013
Author Name:	Shea Newt, Andrew Schwartzmeyer

Abstract

The purpose of this report is to detail the process of exploiting a format-string input to gain read and write access to arbitrary locations in the program's memory. The vulnerability that allowed the exploit detailed in this report was due to a call to the C function 'printf' that accepts a user-alterable format-string, which can be abused to pop values off the stack since 'printf' cannot verify if it was supplied with enough arguments.

Problem and Background

The vulnerability being studied is that of a format-string in a printf call, which is where a program accidentally allows the popping of values off the stack that it was not supposed to. If the format-string printf accepts is supplied by an external user, a malicious person can craft the input in such a way as to read and write arbitrary memory values. This can lead to the user changing the behavior of or crashing the program. This type of vulnerability is introduced through unsafe programming, where the C function printf is given a format-string that is directly alterable by a user, instead of a format-string that asks through an argument for a user provided string to print. The vulnerability exists because printf accepts a variable-length argument list, which prevents the compiler from checking that it is supplied with the proper number of arguments(Du 5). Thus, printf can pop values off the stack past what it was given. It is important to study format-string vulnerabilities because introducing them into a program accidentally as a programmer is easy, but they are preventable if the programmer understands their causes.

Problem Detail

For this particular lab, we are studying the vulnerability of a provided C source code file, *vul_prog.c* (Du 2). It consists of two hard-coded (for simplicity) secrets, “SECRET1” and “SECRET2” holding the values 0x44 and 0x55 respectively, and a sole main function, which does the following things:

1. Declares an array “user_input” of 100 chars
2. Declares an int pointer “secret”
3. Declares an int “int_input”
4. Declares the ints “a”, “b”, “c”, and “d” which are unused
5. Allocates memory for two ints and returns the pointer into “secret”
6. Assigns “SECRET1” and “SECRET2” into the first and second indices of “secret” respectively, which are stored on the heap
7. Prints the address on the stack of the variable “secret”
8. Prints the value on the heap of the variable “secret”
9. Prints the addresses of the two secrets as stored in “secret”
10. Asks for a decimal integer and stores it in “int_input” using scanf
11. Asks for a string and stores it in “user_input” (note that scanf introduces a buffer overflow vulnerability here)
12. Calls printf with “user_input” as the format-string
13. Finally, prints the original and current values of the secrets

Because of the way that printf is called on the user’s input, this program has a format-string vulnerability of which we are going to take advantage in order to a) crash the program, b) print out the secret[1] value, c) modify the secret[1] value, and d) modify the secret[1] value to a pre-determined value. We are to do this without modifying *vul_prog.c*’s source code, but instead exploit it exclusively via the user input.

Tasks

Our first task is to properly setup the provided virtual machine environment in which we will complete this lab. We are given an Ubuntu Desktop 11.04 VM, and the first order of business is to lock it down. To do this we set its virtual network adaptor to “Internal Network” so that it cannot communicate with any other machine, then we change the credentials of both Ubuntu users and the MySQL user (although the latter is not used in this lab).

In order to exploit *vul_prog.c*, we need to craft our inputs in a way that will abuse printf to print out the value at a specific memory location. Fortunately, *vul_prog.c* intentionally provides us with the address of the secret on the stack, and accepts a decimal input before accepting the format string. We can translate the hexadecimal address on the heap of “secret[1]” (or whichever variable we want to target) into a decimal, and input this into “int_input”, which exists on the stack. Now we have, located on the stack, the heap address of the secret which we want to read.

Since the input string is also located on the stack, and printf reads values from the stack, we can fill the format-string with the necessary number of format calls to advance printf’s pointer to “int_input”. These format calls are parameters in the string which printf interprets as commands to pop values from the stack. They look like “%x”, “%d”, “%s”, “%n” etc., which correspond to a hexadecimal (unsigned int), decimal (int), string ((const) (unsigned) char *), and number of bytes written so far (* int). The first two are passed by value and the last two by reference.

So, to advance printf’s pointer to “int_input” and print the data at the target address, we chain together a string of %x parameters followed by a %s, like this: “%x,%x,%x,%x,%x,%x,%x,%s” (Du 7). The number of %x’s required is determined by experimentation, and is system dependent. With the pointer sufficiently advanced, the %s parameter will cause printf to interpret the value it pops from the stack (in our case, it will be “int_input”) as an address to dereference and print the value it contains. In this way, we can read arbitrary memory locations through printf.

Next we want to overwrite the secret values. Again printf is easily

exploited to accomplish this. The previously mentioned “%n” parameter will count the number of printed characters in the string up to the %n and save this number into the provided variable reference. However, if no variable is provided, it will still pop a value from the stack, interpret it as an address, and save the number of characters to the location pointed by that address. By replacing the trailing %s in our previous format-string with %n, instead of reading the value of the target variable, we can now write to it. We can write any arbitrary number to the variable by padding our format-string with dummy characters so that the total number of characters before the %n parameter is equal to the arbitrary number we wish to write. (Du 7)

Next we try the task again, but without a) address randomization, and b) the first scanf statement which puts a decimal into “int_input”. We disable the Linux kernel’s (version 2.6.38-8-generic) address randomization by executing the command “sysctl -w kernel.randomize_va_space=0” as root (Du 3). Address randomization is a safety feature designed to make attacks such as ours more difficult, by ensuring that each execution of the program gets assigned a different address space, so that we cannot execute it, record the addresses, and then execute it again using the information to read/write the variables, since each subsequent execution will be using different addresses.

Now a problem arises with having to type into “user_input”, as scanf takes the typed characters and does an ASCII table translate on them, so what we type is not what actually gets stored (Du 3). This was not previously a problem as before we had a scanf statement that expected a decimal input, and thus did not change the values, but now we our only scanf input accepts a string, thus causing the ASCII lookup. If we want to explicitly input an exact memory address, we are going to have to use another program, *write_string.c*, to craft the input for us. The author of this lab provide us with the code to do this. (Du 4)

This *write_string.c* program will write a string to a file *mystring* which we then input to *vul_prog.c*’s stdin rather than our keyboard input. This can be accomplished on the command line, where *vul* is the compiled *vul_prog.c* program: “./vul < mystring”. In *write_string.c*, we can set the first four bytes of a buffer of chars to any arbitrary address we want, and follow it with input from the keyboard for the rest of the format-string; the

program then writes this *mystring*. This mitigates the problem of otherwise being unable to type the hexadecimal address from our keyboard without it being interpreted as ASCII text.

With address randomization turned off, we can execute *vul_prog.c*, record the address of “secret[1]” which we want to target, use *write_string.c* to put the address followed by the format-string into *mystring*, and now exploit *vul_prog.c* by running it again using *mystring* as its input. This would not work with address randomization as the address of “secret[1]” would change between executions. Since it is not on though, we are able to repeat our previous attacks and gain read and write access to the secret variables.

However, before successfully repeating our attacks, we were unlucky enough to have an address assigned to “secret[1]” which, as a hexadecimal number, contained scanf separator characters (in particular, the address we had contained “0x0C”). This is because there are special numbers such as 0x0A (newline), 0x0C (form feed), 0x0D (return), and 0x20 (space) that, when encountered by scanf, will cause it to stop reading. The author of the lab allowed us to essentially cheat and insert an additional malloc call before the allocation of “secret” took place, which let us change the address of “secret” until we got a lucky address that did not contain separator characters, and thus could be input via scanf. (Du 3)

Finally, crashing *vul_prog.c* was relatively easy. By using a format-string with enough %x’s to put a trailing %s after the variable frame, when printf tried to dereference a number that was not a proper memory address, the program underwent a segmentation fault and crashed. Although crashing is not particularly useful for gaining or altering information, it is an effective attack if the goal is destroying service reliability, or if the crashed program would otherwise have been a security measure we needed to get past.

Answers

We only found two unanswered questions in this lab:

“When you run the program once again, will you get the same address?”

With address randomization off, yes, we are likely to get the same addresses. We can use this information to more easily exploit the program. With address randomization on, then no, we will not get the same addresses.

“What trouble can be caused by printf() when it starts to fetch data that is meant for it?”

Assuming that the author meant “data that is *not* for it”, there is much trouble that can be caused. The %x value allows us to move printf’s pointer wherever we would like in the stack, so that we can use the %s parameter to read arbitrary data (which can also crash the program, if we have it attempt to dereference an invalid memory address), or use the %n parameter to overwrite a variable with an arbitrary number of choosing. This arbitrary number could even, in theory, be crafted in such a way as to be the hexadecimal representation of assembly code, which could let us execute arbitrary instructions and even open up a shell.

Code

Here we present the fully commented code from our lab work. For readability purposes, the font-size of the code has been shrunk from 14-point to 10-point.

First, the vulnerable *vul_prog.c* with the vulnerability underlined:

```
/* vul_prog.c */

/* This code was provided by the lab's author Wenliang Du, of Syracuse
   University under the GNU Free Documentation License*/

#define SECRET1 0x44
#define SECRET2 0x55

int main(int argc, char *argv[])
{
  char user_input[100];
  int *secret;
  int int_input;
  int a, b, c, d; /* other variables, not used here.*/

/* Second malloc statement needed to remedy initial address containing
 * x0C the form feed character. 16 was an arbitrary number, I guessed
 * until the address was free of terminating hex values.
 */
  a = (int *) malloc(16*sizeof(int));

  /* The secret value is stored on the heap */
  secret = (int *) malloc(2*sizeof(int));

  /* getting the secret */
  secret[0] = SECRET1; secret[1] = SECRET2;

  printf("The variable secret's address is 0x%8x (on stack)\n", &secret);
  printf("The variable secret's value is 0x%8x (on heap)\n", secret);
  printf("secret[0]'s address is 0x%8x (on heap)\n", &secret[0]);
  printf("secret[1]'s address is 0x%8x (on heap)\n", &secret[1]);

  printf("Please enter a decimal integer\n");
```

```
  scanf("%d", &int_input);  /* getting an input from user */
  printf("Please enter a string\n");
  scanf("%s", user_input); /* getting a string from user */

  /* Vulnerable place */
  printf(user_input);
  printf("\n");

  /* Verify whether your attack is successful */
  printf("The original secrets: 0x%x -- 0x%x\n", SECRET1, SECRET2);
  printf("The new secrets:      0x%x -- 0x%x\n", secret[0], secret[1]);
  return 0;
}
```

Now, *vul_prog.c* without the first scanf into a decimal integer, used without address randomization:

```
/* vul_prog.c */

/* This code was provided by the lab's author Wenliang Du, of Syracuse
   University under the GNU Free Documentation License*/

#define SECRET1 0x44
#define SECRET2 0x55

int main(int argc, char *argv[])
{
  char user_input[100];
  int *secret;
  /* Here the int int_input was removed */
  int a, b, c, d; /* other variables, not used here.*/

/* Second malloc statement needed to remedy initial address containing
 * x0C the form feed character. 16 was an arbitrary number, I guessed
 * until the address was free of terminating hex values. */
  a = (int *) malloc(16*sizeof(int));

  /* The secret value is stored on the heap */
  secret = (int *) malloc(2*sizeof(int));

  /* getting the secret */
  secret[0] = SECRET1; secret[1] = SECRET2;

  printf("The variable secret's address is 0x%8x (on stack)\n", &secret);
  printf("The variable secret's value is 0x%8x (on heap)\n", secret);
  printf("secret[0]'s address is 0x%8x (on heap)\n", &secret[0]);
  printf("secret[1]'s address is 0x%8x (on heap)\n", &secret[1]);

  /* Here the scanf into int_input was removed */
  printf("Please enter a string\n");
  scanf("%s", user_input); /* getting a string from user */
```

```
  /* Vulnerable place */
  printf(user_input);
  printf("\n");

  /* Verify whether your attack is successful */
  printf("The original secrets: 0x%x -- 0x%x\n", SECRET1, SECRET2);
  printf("The new secrets:      0x%x -- 0x%x\n", secret[0], secret[1]);
  return 0;
}
```

Now the provided *write_string.c* code, with the address we needed:

```
/* write_string.c */

/* This code was provided by the lab's author Wenliang Du, of Syracuse
   University under the GNU Free Documentation License*/
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main() {
  char buf[1000];
  int fp, size;
  unsigned int *address;

/* Putting any number you like at the beginning of the format string */
  address = (unsigned int *) buf;

/* This is the address of the second secret number we want we want
 * scanf to accept before reading the rest of our exploit string.
 * This is not necessary if there is a scanf reading an int before
 * reading the buffer because we can put this value in at that time.
 */
  *address = 0x804b054;

/* Getting the rest of the format string */
  scanf("%s", buf+4);
  size = strlen(buf+4) + 4;
  printf("The string length is %d\n", size);
/* Writing buf to "mystring" */
  fp = open("mystring", O_RDWR | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
  if (fp != -1) {
    write(fp, buf, size);
    close(fp);
  }
  else {
    printf("Open failed!\n");
  }
}
```

Finally, *vul_prog.c* with this specific vulnerability fixed:

```
/* vul_prog.c */

/* This code was provided by the lab's author Wenliang Du, of Syracuse
   University under the GNU Free Documentation License*/

#define SECRET1 0x44
#define SECRET2 0x55

int main(int argc, char *argv[])
{
  char user_input[100];
  int *secret;
  int int_input;
  int a, b, c, d; /* other variables, not used here.*/

/* Second malloc statement needed to remedy initial address containing
 * x0C the form feed character. 16 was an arbitrary number, I guessed
 * until the address was free of terminating hex values.
 */
  a = (int *) malloc(16*sizeof(int));

  /* The secret value is stored on the heap */
  secret = (int *) malloc(2*sizeof(int));

  /* getting the secret */
  secret[0] = SECRET1; secret[1] = SECRET2;

  printf("The variable secret's address is 0x%8x (on stack)\n", &secret);
  printf("The variable secret's value is 0x%8x (on heap)\n", secret);
  printf("secret[0]'s address is 0x%8x (on heap)\n", &secret[0]);
  printf("secret[1]'s address is 0x%8x (on heap)\n", &secret[1]);

  /* Note that these scanf's also introduce a buffer overflow
     vulnerability. If a user inputs a string longer than 100 chars,
     user_input will overflow. We will 'fix' that by setting a maximum
     width one less than the size of user_input (to account for the
     trailing null character). Fix is in quotes because a better
     solution would be to replace these functions with fgets; however,
     we feel that is outside the scope of this particular lab. */
  printf("Please enter a decimal integer\n");
  scanf("%d", &int_input);  /* getting an input from user */
  printf("Please enter a string\n");
  scanf("%99s", user_input); /* getting a string from user */

  /* Previous vulnerable place. This is a simple fix: instead of
     allowing a user-alterable format-string, we supply a hard-coded
     format-string consisting of a single %s parameter, and pass our
     user_input string as an argument to printf with which it fills
     the %s */
  printf("%s", user_input);
  printf("\n");
```

```
  /* Verify whether your attack is successful */
  printf("The original secrets: 0x%x -- 0x%x\n", SECRET1, SECRET2);
  printf("The new secrets:      0x%x -- 0x%x\n", secret[0], secret[1]);
  return 0;
}
```

References

[1] Du, Wenliang. 2006-2013. “Format String Vulnerability Lab”. http://www.cis.syr.edu/~wedu/seed/Labs/Vulnerability/Format_String/