MONASH University

FIT5003: Other Exploits, Memory Defense and Secure Implementation

Dr Xiaoning Du
Department of Software Systems and Cybersecurity
Faculty of Information Technology

Learning Outcomes of This Lecture

- Discuss integer overflow and exploits
- Discuss format string vulnerability and exploits
- Understand the meaning of general memory safety
- Understand the meaning of type safety

Recap: Buffer Read Overflow

```c
int main()
{
    char buf[100], *p;
    int i, len;
    while (1)
    {
        p = fgets(buf,sizeof(buf),stdin);
        if (p == NULL) return 0;

        len = atoi(p);
        p = fgets(buf,sizeof(buf),stdin);
        if (p == NULL) return 0;

        for (i=0; i<len; i++)
        {
            if (!iscntrl(buf[i]))
            {
              putchar(buf[i]);
            }
            else
            {
              putchar('.');
            }
        }
        printf("\n");
    }
}
```

read integer

read message

echo message

Q: Is this program vulnerable?

Recap: Buffer Read Overflow

```c
int main()
{
    char buf[100], *p;
    int i, len;
    while (1)
    {
        p = fgets(buf,sizeof(buf),stdin);
        if (p == NULL) return 0;

        len = atoi(p);
        p = fgets(buf,sizeof(buf),stdin);
        if (p == NULL) return 0;

        for (i=0; i<len; i++)
        {
            if (!iscntrl(buf[i]))
            {
              putchar(buf[i]);
            }
            else
            {
              putchar('.');
            }
        }
        printf("\n");
    }
}
```

read integer

read message

echo message

```
% ./echo-server

24

every good guy does fine

ECHO: |every good guy does fine|

10

hello there

ECHO: |hello ther|  <-- OK: input len < buf size

25

hello

ECHO: |hello..here..y does fine.|
```

Recap: Buffer Read Overflow

```c
int main()
{
    char buf[100], *p;
    int i, len;
    while (1)
    {
        p = fgets(buf,sizeof(buf),stdin);
        if (p == NULL) return 0;

        len = atoi(p);
        p = fgets(buf,sizeof(buf),stdin);
        if (p == NULL) return 0;

        for (i=0; i<len; i++)
        {
            if (!iscntrl(buf[i]))
            {
              putchar(buf[i]);
            }
            else
            {
              putchar('.');
            }
        }
        printf("\n");
    }
}
```

read integer

read message

echo message

```
% ./echo-server
24
every good guy does fine
ECHO: |every good guy does fine|
10
hello there
ECHO: |hello ther|
25
hello
ECHO: |hello..here..y does fine.|
```

Leak data of previous message: len > size

Other Memory Vulnerability and Its Exploitation

- Integer overflow

Integer Overflow

- Integer variables commonly used for array indexing and arithmetic
  - unsigned int: 32 bit [0, 2^{32}-1]
  - int: 32 bit [-2^{31}, 2^{31}-1]
  - short: 16 bit [-2^{15}, 2^{15}-1]
  - unsigned short: 16 bit [0, 2^{16}-1]
- Integer variables can overflow if one writes a value larger or smaller than the maximum possible value for the integer type
- Maliciously chosen integer overflows can also change program behaviour, and can be exploitable by attackers

Example 1

```c
unsigned short num = 0xffff;
num = num + 2;
printf("num = 0x%x\n", num);

unsigned short num_mul = 0x4000;
num_mul = num_mul * 4;
printf("num_mul = 0x%x\n", num_mul);
```

Q: what is the output of this function?

Example 1

```c
unsigned short num = 0xffff;
num = num + 2;
printf("num = 0x%x\n", num);

unsigned short num_mul = 0x4000;
num_mul = num_mul * 4;
printf("num_mul = 0x%x\n", num_mul);
```

Ans:

num = 0x01

num_mul = 0x00

Example 2

```
int catvars(char *buf1, char *buf2, unsigned int len1, unsigned int len2)
{
    char mybuf[256];
    if((len1 + len2) > 256)
        {return -1; }
    memcpy(mybuf, buf1, len1);
    memcpy(mybuf + len1, buf2, len2);
    do_some_stuff(mybuf);
    return 0;
}
```

Q: which line is vulnerable?

Is possible that len1 or len2 are extremely large, but their sum are less than 256?

Example 2

```
int catvars(char *buf1, char *buf2, unsigned int len1, unsigned int len2)
{
    char mybuf[256];
    if((len1 + len2) > 256)
        {return -1; }
    memcpy(mybuf, buf1, len1);
    memcpy(mybuf + len1, buf2, len2);
    do_some_stuff(mybuf);
    return 0;
}
```

- With input:
  - len1 = 0x104 (= 256 + 4)
  - len2 = 0xfffffffc (= 2^{32} – 4)
- Length sum overflows:
  - len1 + len 2 = 256
- Bypass the buffer overflow check
- How to fix it?

Example 3

```
void vulnerable()
{
    char **response;
    int nresp = packet_get_int();
    if (nresp > 0)
    {
        response = malloc (nresp*sizeof(char*));
        for (i = 0; i < nresp; i++)
        {
            response [i] = packet_get_string (NULL);
        }
    }
}
```

Q: which line is vulnerable?

- size_t sizeof (expr), and size_t is an unsigned integral data type

Example 3

```
void vulnerable()
{
    char **response;
    int nresp = packet_get_int();
    if (nresp > 0)
    {
        response = malloc (nresp*sizeof(char*));
        for (i = 0; i < nresp; i++)
        {
            response [i] = packet_get_string (NULL);
        }
    }
}
```

unsigned int

- if **nresp** is set as 2^30 (0x40000000)
- then **nresp*sizeof(char*)** overflows to become 0
- subsequent writes to allocated **response** overflow it
- How to fix it?

Example 4

```
int table[800];
int insert_in_table(int val, int pos)
{
    if(pos > sizeof(table) / sizeof(int))
        { return -1; }
    table[pos] = val;
    return 0;
}
```

This is another type of error that may be associated with integer overflow.

"Negative indexing" errors:

- Will input **pos = -1** get through the if test?

Example 4

```
int table[800];
int insert_in_table(int val, int pos)
{
    if(pos > sizeof(table) / sizeof(int))
        { return -1; }
    table[pos] = val;
    return 0;
}
```

“Negative indexing” errors:

- Will input **pos = -1** get through the if test?
- No. pos will be cast to unsigned int before the comparison. The program will directly return -1.
- What if the table size is larger than 2^31?

Example 4

```
int table[800];
int insert_in_table(int val, int pos)
{
    if(pos > sizeof(table) / sizeof(int))
        { return -1; }
    table[pos] = val;
    return 0;
}
```

"Negative indexing" errors:

- Will input **pos = -1** passes the if test?
- No. pos will be cast to unsigned int before the comparison. The program will directly return -1.
- What if the table size is larger than 2^31?
  - It can pass the if check with pos = -2^31.

Other Memory Vulnerability and Its Exploitation

- Format string vulnerability

Format Strings

- C's printf family supports formatted I/O: print data in specific formats
  - Syntax: `printf("format string", <vars>)`

```
int int1 = 12;
char str1[6];
strcpy(str1, "hello");
printf ("%d %x %s", int1, int1, str1);
```

  - Output: `12 c hello`

- Format string is optional
  - When passing only one variable, if it's not a format string, the variable will be printed in default format, which opens a door to format string injection attack!

What's the difference?

```
void safe()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf("%s",buf);
}
```

```
void vulnerable()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf(buf);
}
```

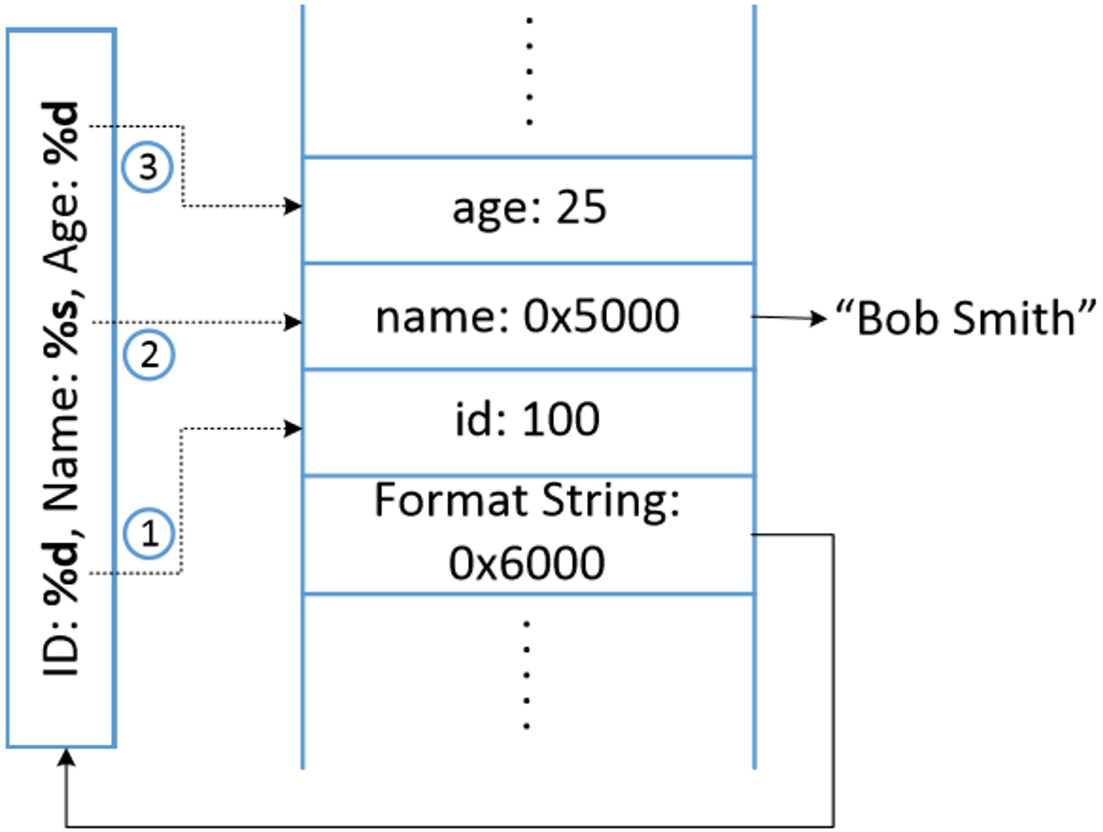
Attacker controls the format string

How `printf()` Access Optional Arguments

```
#include <stdio.h>

int main()
{
   int id=100, age=25; char *name = "Bob Smith";
   printf("ID: %d, Name: %s, Age: %d\n", id, name, age);
}
```

- Here, `printf()` has three optional arguments. Elements starting with "%" are called format specifiers.
- A variable pointer `va_list` is maintained to fetch the variables
- `printf()` scans the format string and prints out each character until "%" is encountered.
- `printf()` fetches the optional argument pointed by `va_list` and advances it to the next argument.

How `printf()` Access Optional Arguments

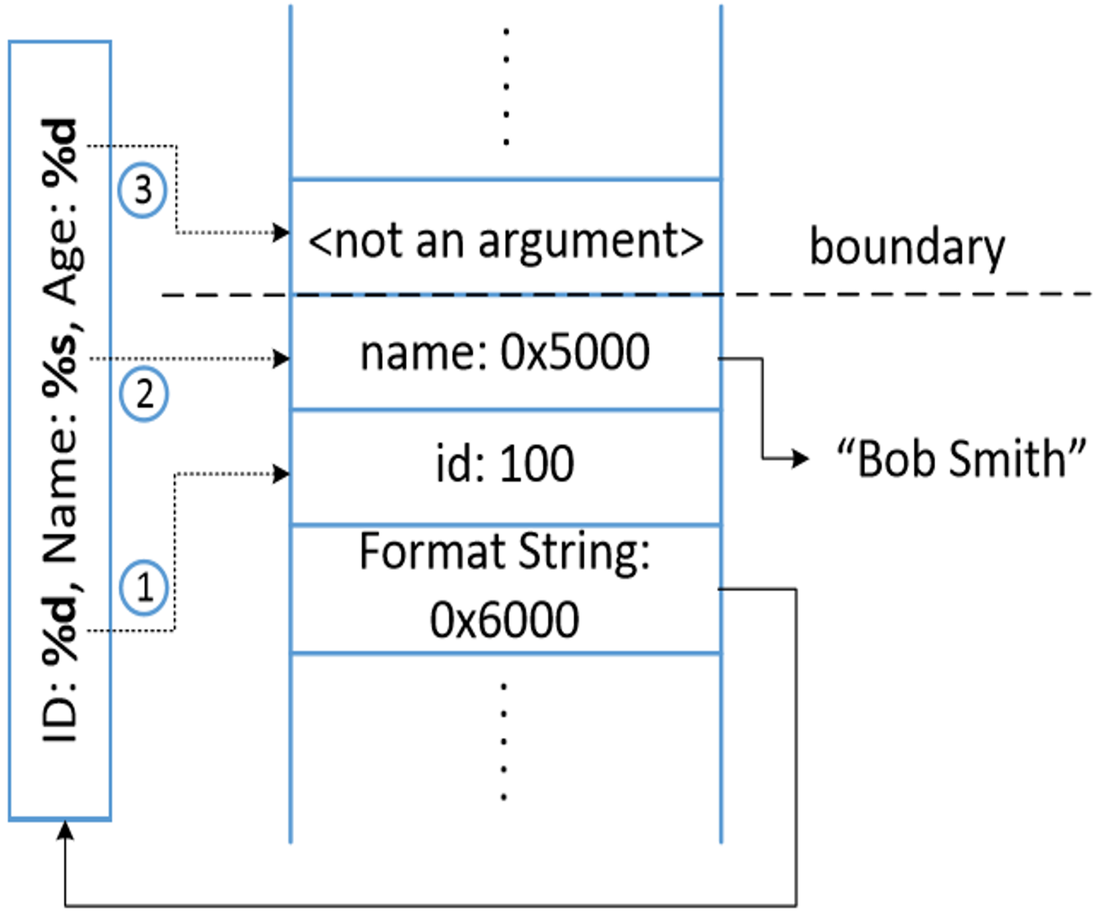


- When printf() is invoked, the arguments are pushed onto the stack in reverse order.
- When it scans and prints the format string, printf() replaces %d with the value from the first optional argument and prints out the value.
- va_list is then moved to the position 2.

Missing Optional Arguments

```
#include <stdio.h>

int main()
{
    int id=100, age=25; char *name = "Bob Smith";

    printf("ID: %d, Name: %s, Age: %d\n", id, name);
}
```

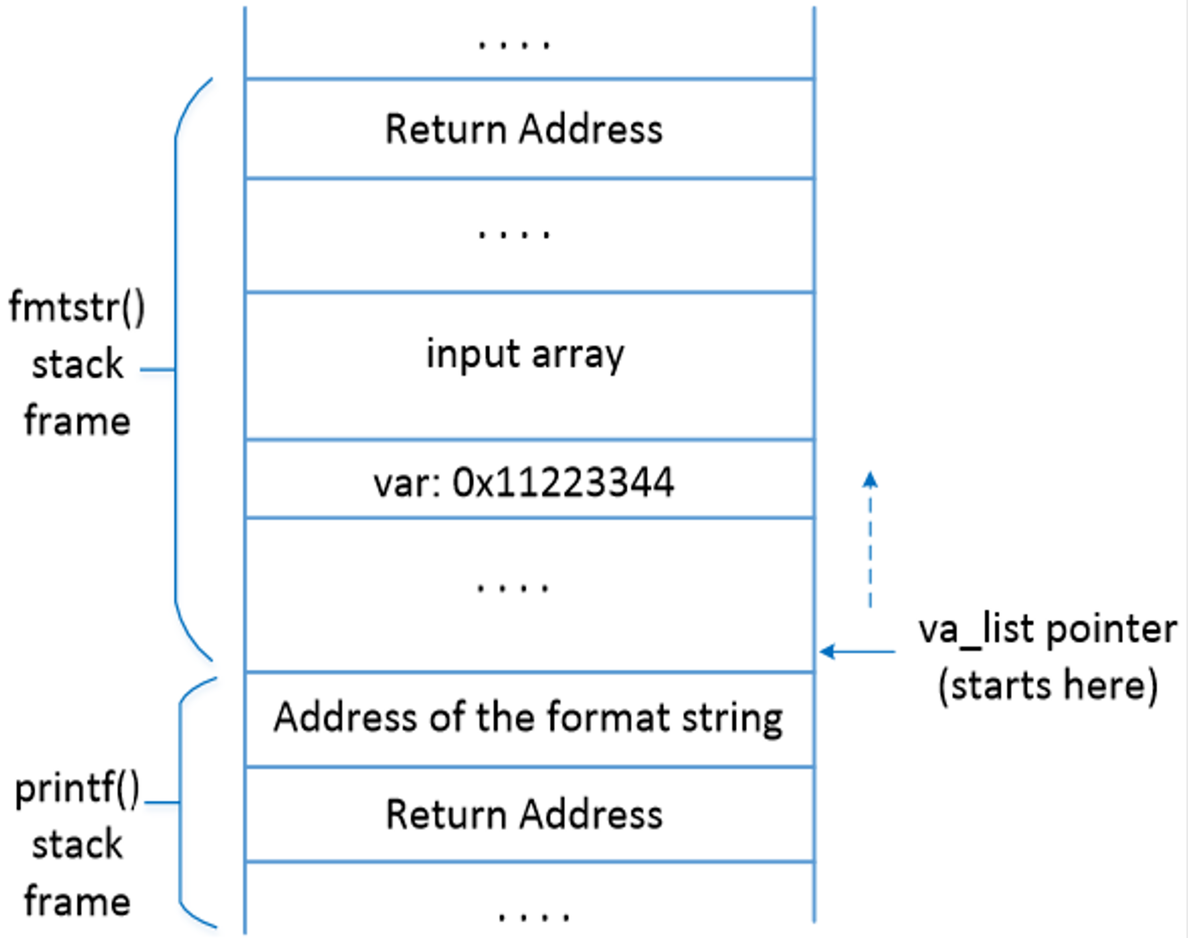
- No idea about the end of the optional argument list.
- It continues fetching data from the stack and advancing va_list pointer.



Vulnerable Program's Stack

Inside `printf()`, the starting point of the optional arguments (va_list pointer) is the position right above the format string argument.

```
#include <stdio.h>

void fmtstr()
{
    char input[100];
    int var = 0x11223344;

    /* print out information for experiment purpose */
    printf("Target address: %x\n", (unsigned) &var);
    printf("Data at target address: 0x%x\n", var);

    printf("Please enter a string: ");
    fgets(input, sizeof(input)-1, stdin);

    printf(input); // The vulnerable place    ①

    printf("Data at target address: 0x%x\n",var);
}

void main() { fmtstr(); }
```



Format String Vulnerability

```
printf(user_input);
```

```
sprintf(format, "%s %s", user_input, ": %d");
printf(format, program_data);
```

```
sprintf(format, "%s %s", getenv("PWD"), ": %d");
printf(format, program_data);
```

In these three examples, user's input (user_input) becomes part of a format string.

What will happen if **user_input** contains format specifiers?

What can be achieved using Format String Vulnerability?

- Crash program
- Print out data on the stack (memory dump)
- Change the program's data in the memory
- Change the program's data to specific value
- Inject Malicious Code

Some Useful Format specifiers:

- %x - Read data from the stack
- %c - Read character from memory
- %s – Read a string from memory
- %n - Write an integer to locations in memory

Crash Program

```
$ ./vul
......
Please enter a string: %s%s%s%s%s%s%s%s
Segmentation fault (core dumped)
```

- Use input: `%s%s%s%s%s%s%s%s`
- `printf()` parses the format string.
- For each `%s`, it fetches a value where va_list points to and advances va_list to the next position.

- As we give %s, `printf()` treats the value as address and fetches data from that address. If the value is not a valid address, the program crashes.

Print Out Data on the Stack

```
$ ./vul
......
Please enter a string: %x.%x.%x.%x.%x.%x.%x.%x
63.b7fc5ac0.b7eb8309.bffff33f.11223344.252e7825.78252e78.2e78252e
```

- Suppose a variable on the stack contains a secret (constant) and we need to print it out.
- Use user input: `%x%x%x%x%x%x%x%x`

- `printf()` prints out the integer value pointed by va_list pointer and advances it by 4 bytes.
- Number of `%x` is decided by the distance between the starting point of the va_list pointer and the variable. It can be achieved by trial and error.

Change Program's Data in the Memory

Example's Goal: change the value of `var` variable from 0x11223344 to some other value.

- **`%n`**: Writes the number of characters printed out so far into memory.
- `printf("hello%n",&i)` ⇒ When printf() gets to `%n`, it has already printed 5 characters, so it stores 5 to the provided memory address.
- `%n` treats the value pointed by the va_list pointer as a memory address and writes into that location.
- Hence, if we want to write a value to a memory location, we need to have it's address on the stack.

```
#include <stdio.h>

void fmtstr()
{
    char input[100];
    int var = 0x11223344;

    /* print out information for experiment purpose */
    printf("Target address: %x\n", (unsigned) &var);
    printf("Data at target address: 0x%x\n", var);

    printf("Please enter a string: ");
    fgets(input, sizeof(input)-1, stdin);

    printf(input); // The vulnerable place    ①

    printf("Data at target address: 0x%x\n",var);
}

void main() { fmtstr(); }
```

Change Program's Data in the Memory

Assuming the address of `var` is `0xbffff304` (can be obtained using gdb)

```
$ echo $(printf "\x04\xf3\xff\xbf").%x.%x.%x.%x.%x.%n > input
```

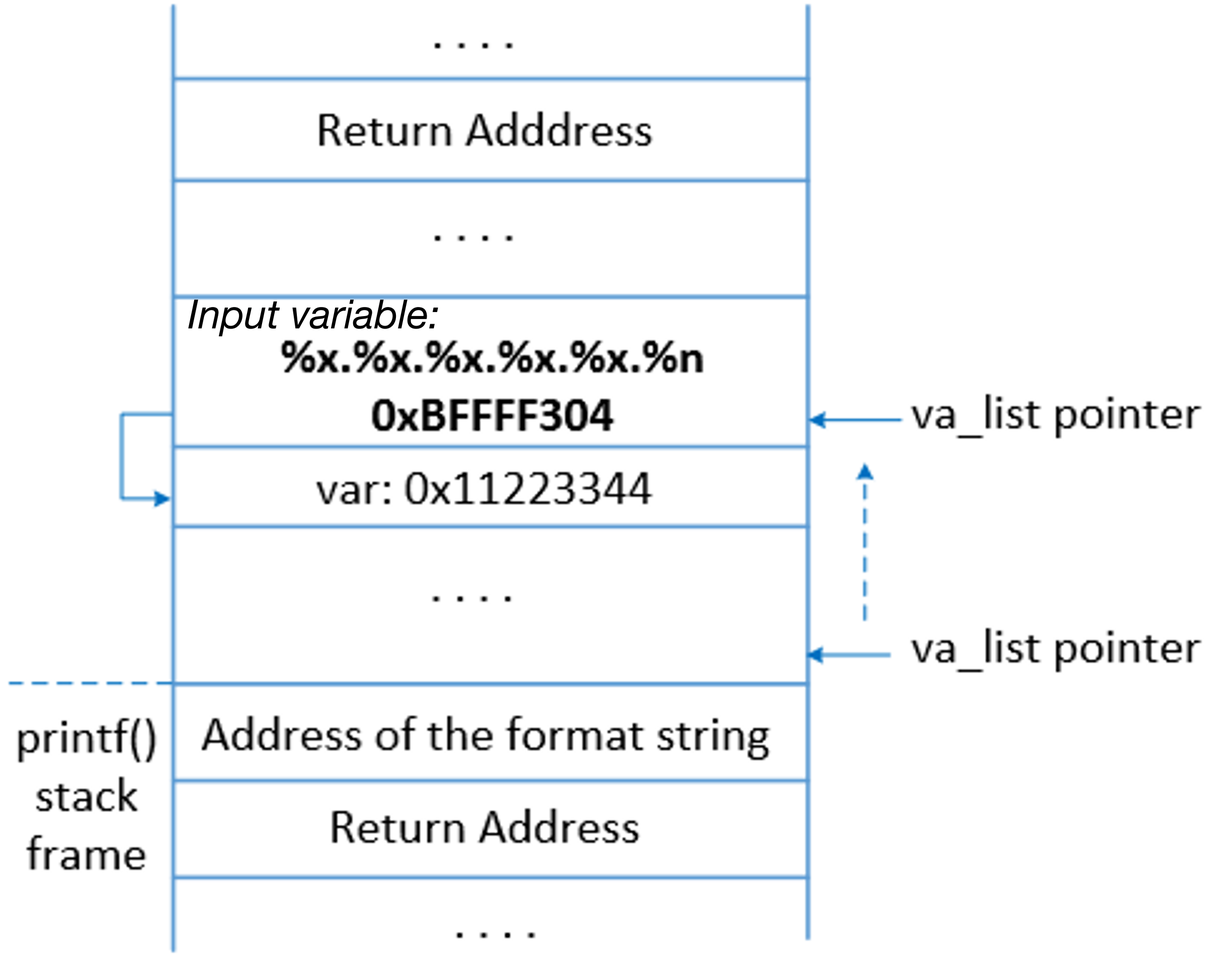
```
$ ./vul <input
```

- The address of `var` is given in the beginning of the input so that it is stored on the stack.
- $(command): Command substitution. Allows the output of the command to replace the command itself.
- "`\x04`" : Indicates that "`04`" is an actual number and not as two ascii characters.

Change Program's Data in the Memory

- `var`'s address (`0xbffff304`) is on the stack.
- **Goal :** To move the va_list pointer to this location and then use `%n` to store some value.
- `%x` is used to advance the va_list pointer.

Q: *What value will be placed in var?*



Change Program's Data in the Memory

```
$ echo $(printf "\x04\xf3\xff\xbf").%x.%x.%x.%x.%x.%n > input
$ vul < input
Target address: bffff304
Data at target address: 0x11223344
Please enter a string: ****.63.b7fc5ac0.b7eb8309.bffff33f.11223344.
Data at target address: 0x2c      ← The value is modified!
```

- Using trial and error, we check how many `%x` are needed to print out `0xbffff304`.
- Here we need 6 `%x` format specifiers, indicating 5 `%x` and 1 `%n`.
- After the attack, data in the target address is modified to 0x2c (44 in decimal).
- Because 44 characters have been printed out before `%n`.

Change Program's Data to a Specific Value

Goal: To change the value of `var` from `0x11223344` to `0x9896a9`

```
$ echo $(printf
    "\x04\xf3\xff\xbf")_%.8x_%.8x_%.8x_%.8x_%.10000000x%n > input
$ uvl < input
Target address: bffff304
Data at target address: 0x11223344
Please enter a string:
    ****_00000063_b7fc5ac0_b7eb8309_bffff33f_000000
```

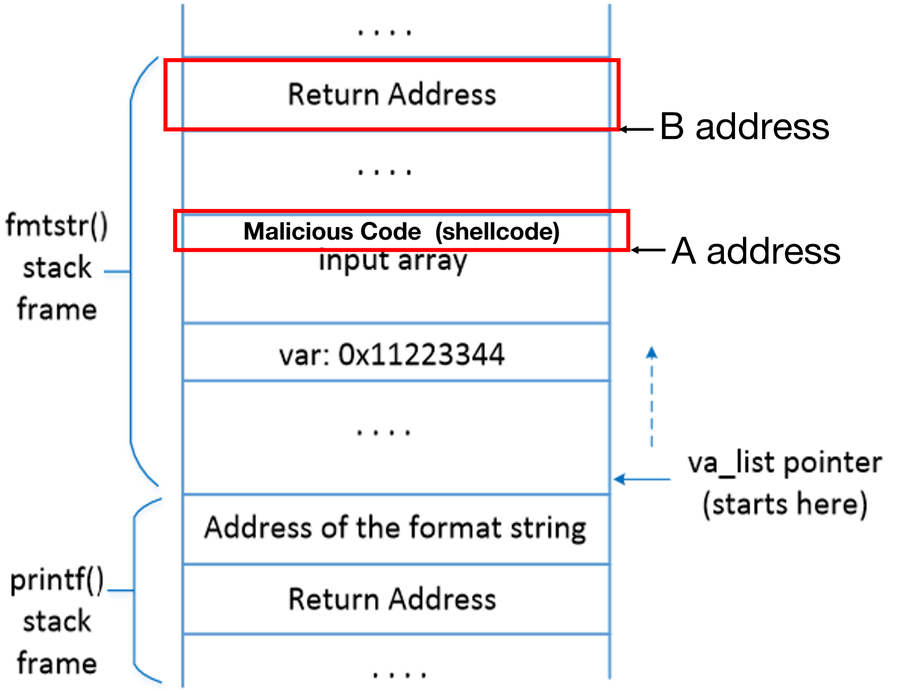
`printf()` has already printed out 41 characters before `%.10000000x`, so, `10000000+41 = 10000041 (0x9896a9)` will be stored in `0xbffff304`.

Inject Malicious Code

How?: Modify the return address of the vulnerable code and let it point it to the malicious code

Challenges :

- Inject Malicious code in the stack
- Find starting address (A) of the injected code
- Find return address (B) of the vulnerable code
- Write value A to B



Inject Malicious Code

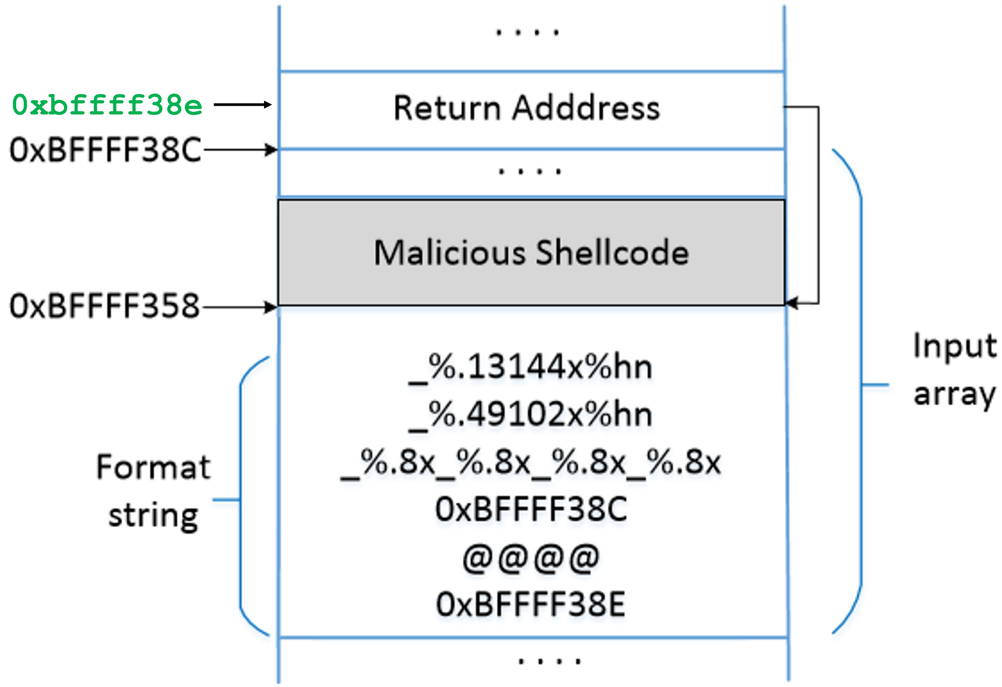
- Using gdb to get the return address and start address of the malicious code.
- Assume that the return address is at **`0xbffff38c`**
- Assume that the start address of the malicious code is **`0xbfff358`**

Goal : Write the value **`0xbffff358`** to address **`0xbffff38c`**
Steps :

- **`0xbffff358`** is 3221222284, which is too large
- Break **`0xbffff38c`** into two contiguous 2-byte memory locations : `0xbffff38c` and `0xbffff38e`.
- Store **`0xbfff(49151)`** into **`0xbffff38e`** and **`0xf358(62296)`** into **`0xbffff38c`**

Inject Malicious Code

- **Number of characters printed before first `%hn` =**
  12 + (4x8) + 5 + 49102 = 49151 (`0xbfff`).
- **After first `%hn`, before second `%hn`, 13144 + 1 =13145 are printed**
- **49151 + 13145 = 62296 (`0xf358`) is printed on `0xbffff38c`**



%hn means to write a short value (2 bytes) at the given address

Countermeasures: Developer

- Avoid using untrusted user inputs for format strings in functions like `printf, sprintf, fprintf, vprintf, scanf, vfscanf.`

```
// Vulnerable version (user inputs become part of the format string):
    sprintf(format, "%s %s", user_input, ": %d");
    printf(format, program_data);


// Safe version (user inputs are not part of the format string):
    strcpy(format, "%s: %d");
    printf(format, user_input, program_data);
```

Countermeasures: Compiler

Compilers can detect potential format string vulnerabilities

```
#include <stdio.h>

int main()
{
   char *format = "Hello  %x%x%x\n";

   printf("Hello %x%x%x\n", 5, 4);     ①
   printf(format, 5, 4);               ②

   return 0;
}
```

- **Use two compilers to compile the program: `gcc` and `clang`.**
- **There is a mismatch in the format string.**

- With default settings, both compilers gave warning only for the first `printf()`.
- On giving an option `-wformat=2`, both compilers give warnings for both `printf` statements stating that the format string is not a string literal.
- These warnings just act as reminders to the developers that there is a potential problem but nevertheless compile the programs.

Will BOF Countermeasures work?

- **Address randomization**: Makes it difficult for the attackers to guess the address of the address of the target memory (return address, address of the malicious code)

- **Non-executable Stack/Heap**: This will not work. Attackers can use the return-to-libc technique to defeat the countermeasure.

- **StackGuard**: This will not work. Unlike buffer overflow, using format string vulnerabilities, we can ensure that only the target memory is modified; no other memory is affected.

Attacks in Common

- The attacker is able to control some data that is used by the program
- The use of that data permits unintentional access to some memory area in the program
  - to a buffer
  - to arbitrary positions on the stack

Memory Safety for C

- C/C++: not memory safe
  - You can write memory safe programs … (how to guarantee that?)
- Compilers could add **code** to check for violations
  - An out-of-bounds access would result in an immediate failure
- Performance has been the limiting factor
  - Work by Jones and Kelly in 1997 adds **12x** overhead, and valgrind memcheck adds **17x** overhead

Type Safety

- Each data object is ascribed a **type**
- Intuitively, the type determines how the data is interpreted.
- A value domain is associated with each type.
- Operations on the object are always compatible with the object's type
- Type safety is **stronger** than memory safety

C/C++ is not type safe

- C/C++ is designed for high performance
  - Manual memory management: interaction with low-level memory
- Typical enforcement of type safety is expensive
  - Garbage collection
  - Bounds and null-pointer checks
  - Strict casting rules

Statically Typed Languages

- Java
- Declaration of variables should explicitly indicate the type
- Casting basically happens between super classes and subclasses

Dynamically Typed Languages

- Dynamically typed languages, e.g., Ruby and Python
  - do not require declarations that identify types, can be viewed as type safe
- Each object has one type: Dynamic
  - Each operation on a Dynamic object is permitted, but may be unimplemented (checked at the runtime)

Q&A