Reading: [PF] 3.1, 3.2, 3.3

Lecture 7: Software Security (Part I)

7.1 Overview of software security

7.2 Computer architecture background

- 7.2.1 Code vs data, program counter
- 7.2.2 Stack (aka execution stack, call stack)
- 7.2.3 Control flow integrity

7.2 printf() and format string vulnerability

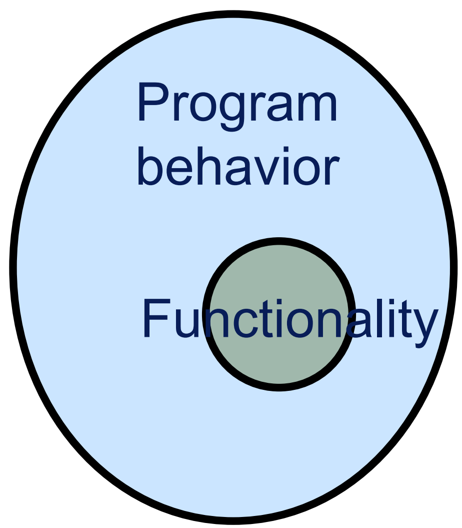
7.1 Overview of Software Security

Program: Requirements and Possible Behavior

Requirements of a program:

- A program has to be **correct**
- A program has to be **efficient**
- A program also has to be ***secure***

Targeted program functionality vs possible behavior:



A program *may* behave **beyond** its intended functionality!

Possible Security Problems

- **Insecure implementation**:
  Many programs are ***not** implemented properly*,
  allowing attacker (i.e. the person who invokes the process)
  to **deviate from the programmer's intents**

- **Unanticipated input**:
  The attacker may supply **input** in a form that is ***not** anticipated* by the programmer, which can unintendedly cause the process to:
  - Access sensitive resources;
  - Execute some "injected" codes; or
  - Deviate from the original intended execution path.

  Either way, the attacker manages to **elevate its privilege**.

- In the next **3 lectures**, we will look at several **classes of insecure programs** and also **the reasons** behind their insecurity!

Some Sample Cases

Buffer Overflow:

- **Morris worm** (1988): exploited a Unix *finger* service to propagate itself over the Internet
- **Code Red worm** (2001): exploited Microsoft's IIS 5.0
- **SQL Slammer worm** (2003): compromised machines running Microsoft SQL Server 2000
- Various attacks on **game consoles** so that unlicensed software can run without the need for hardware modifications: **Xbox**, **PlayStation 2** (PS2 Independence Exploit), **Wii** (Twilight hack)
- …

Some Sample Cases

SQL Injection:

- **Yahoo!** (2012): a hacker group was reported to have stolen **450,000** login credentials from Yahoo! by using a "union-based SQL injection technique“
- **British telco company TalkTalk** (2015): an attack exploiting a vulnerability in a legacy web portal was used to steal the personal details of **156,959** customers

Integer Overflow:

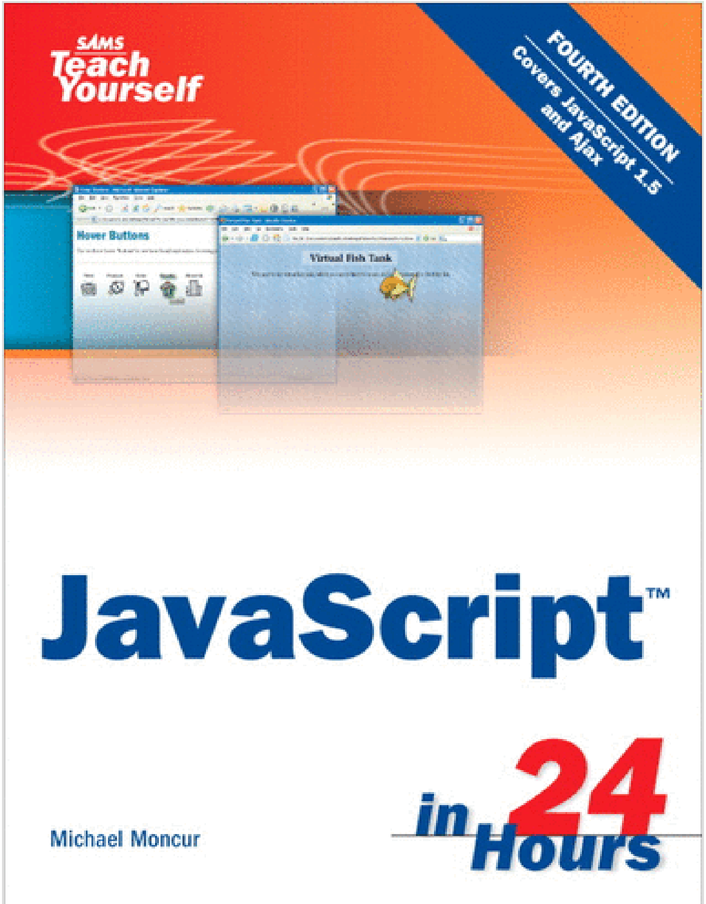
- **European Space Agency’s Ariane 5 rocket** (1996): an unhandled arithmetic overflow in the engine steering software caused its crash, costing $7 billion
- **Resorts World Casino** (2016): a casino machine printed a prize ticket of $42,949,672.76

Root Causes of Security Problems

Why is there a **big security** challenge?

- **Functionality**: still the **primary concern** during design and implementation
  - Security is a **secondary goal**
  - Features pay the bills (typically)
- Unavoidable **human mistakes**:
  - (Lack of) awareness of security problems
  - Careless programmers
- **Complex** modern computing systems:
  - Many of the “bugs” are very *simple* and seem easy to prevent. But programs for complex systems are **large**, e.g. Window XP has 45 millions SLOC (source line of codes) http://en.wikipedia.org/wiki/Source_lines_of_code.
  - Large **attack surface** as well

Programming can be Easy, but Good Programming Isn't So



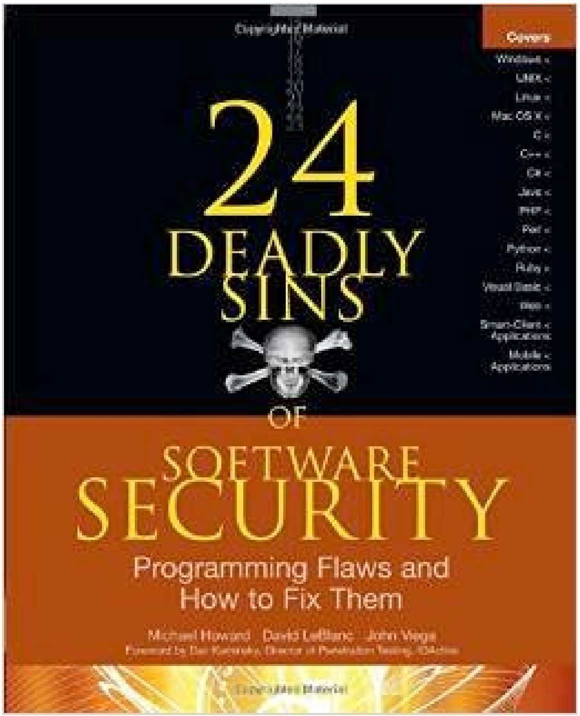
- Maybe enough for learning basic functionality
- **Never enough** for learning *subtle implications* of functionalities
- Result: programs can **do more** than you expect!

Recommended References for Secure Programming

Optional

Some well-known references:

- Michael Howard and David LeBlanc, *Writing Secure Code*, 2nd ed, Microsoft Press, 2002
- Michael Howard, David LeBlanc, and John Viega, *24 Dealy Sins of Software Security: Programming Flaws and How to Fix Them,* McGraw-Hill, 2010



7.2 Computer Architecture Background

7.2.1 Code vs data, program counter
7.2.2 Stack (a.k.a execution stack, call stack)
7.2.3 Control flow integrity

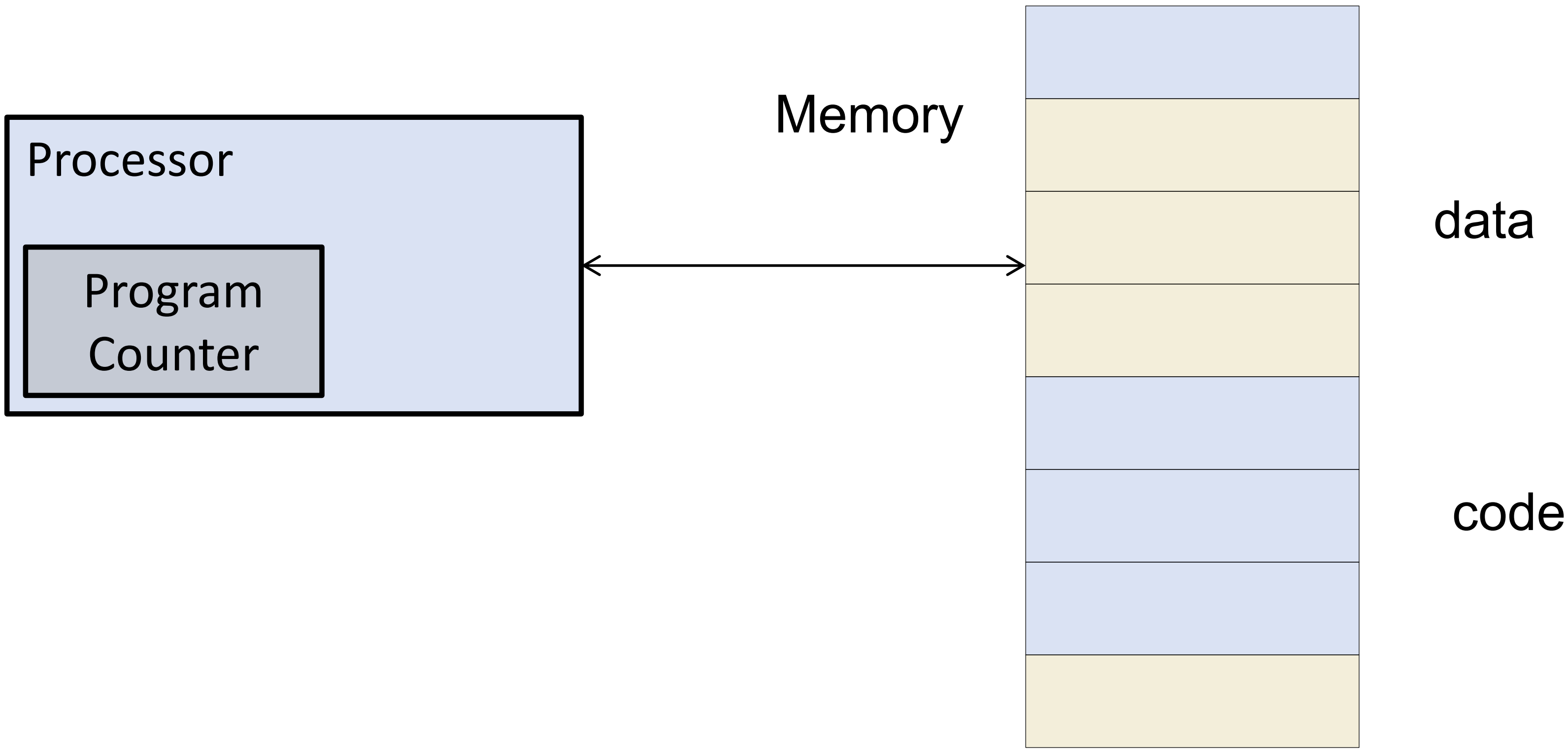
7.2.1 Code vs Data, Program Counter

Code vs Data in Modern Computers

Modern computers are based on the
Von Neumann computer architecture:

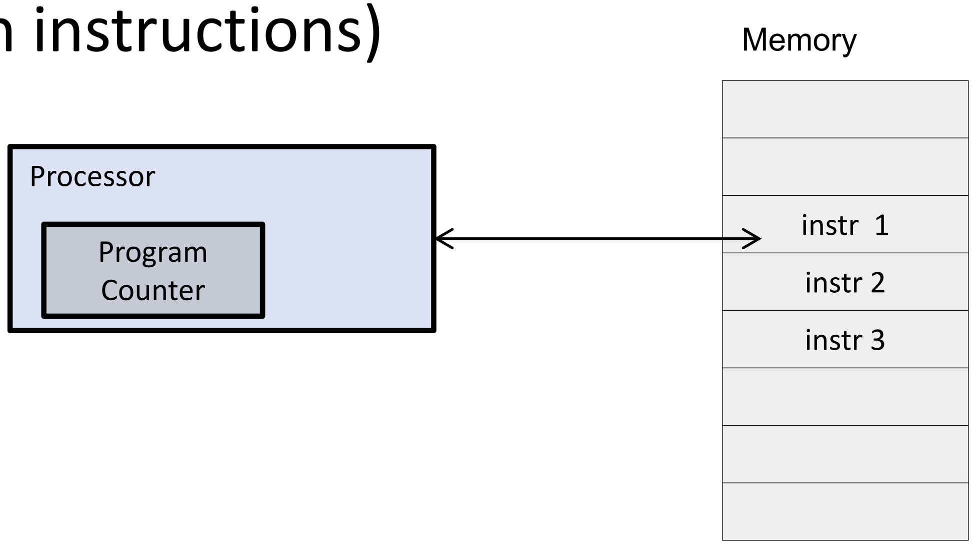
- The code and data are stored **together** in the memory
- There are **no clear distinction** of code and data.
- This is in contrast to the ***Harvard architecture,*** which has hardware components that separately store code and data
- **Serious implication**: programs may be tricked into **treating input data as code**: basis for all *code-injection attacks*!

Code vs Data in Modern Computers



Control Flow

- The ***program counter*** (aka ***Instruction Pointer***):
  a register (i.e. small & fast memory within the processor)
  that stores the address of the next instruction
- After an instruction is completed, the processor **fetches** the next instruction from the address stored in the program counter
- After the next instruction is fetched, the program counter automatically **increases** by 1 (assuming a system with fixed-length instructions)

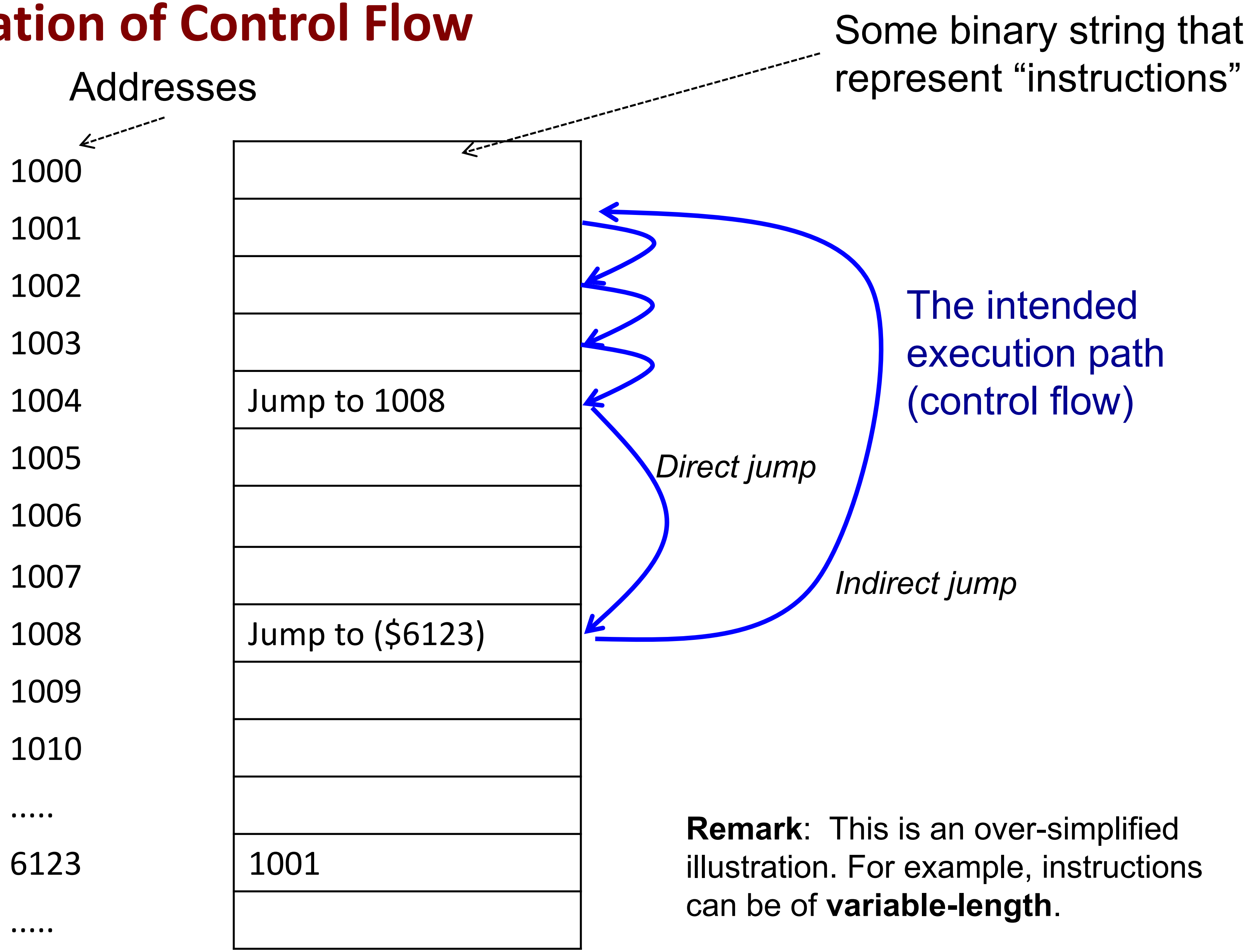


Control Flow

- During execution, besides getting **incremented**, the *program counter* can also be **changed**, for examples*, by:
  1. **Direct jump**:
     Replaced with a *constant value* specified in the instruction
  2. **Indirect jump**:
     Replaced with a *value fetched from the memory*
     (Note that there are many different forms of indirect jump)

*: For simplicity in this module, we omit conditional branch as well as call/return here

Illustration of Control Flow



Remark: This is an over-simplified illustration. For example, instructions can be of **variable-length**.

7.2.2 Stack
(aka Execution Stack, Call Stack)

See: https://en.wikipedia.org/wiki/Call_stack

Functions and Their Executions

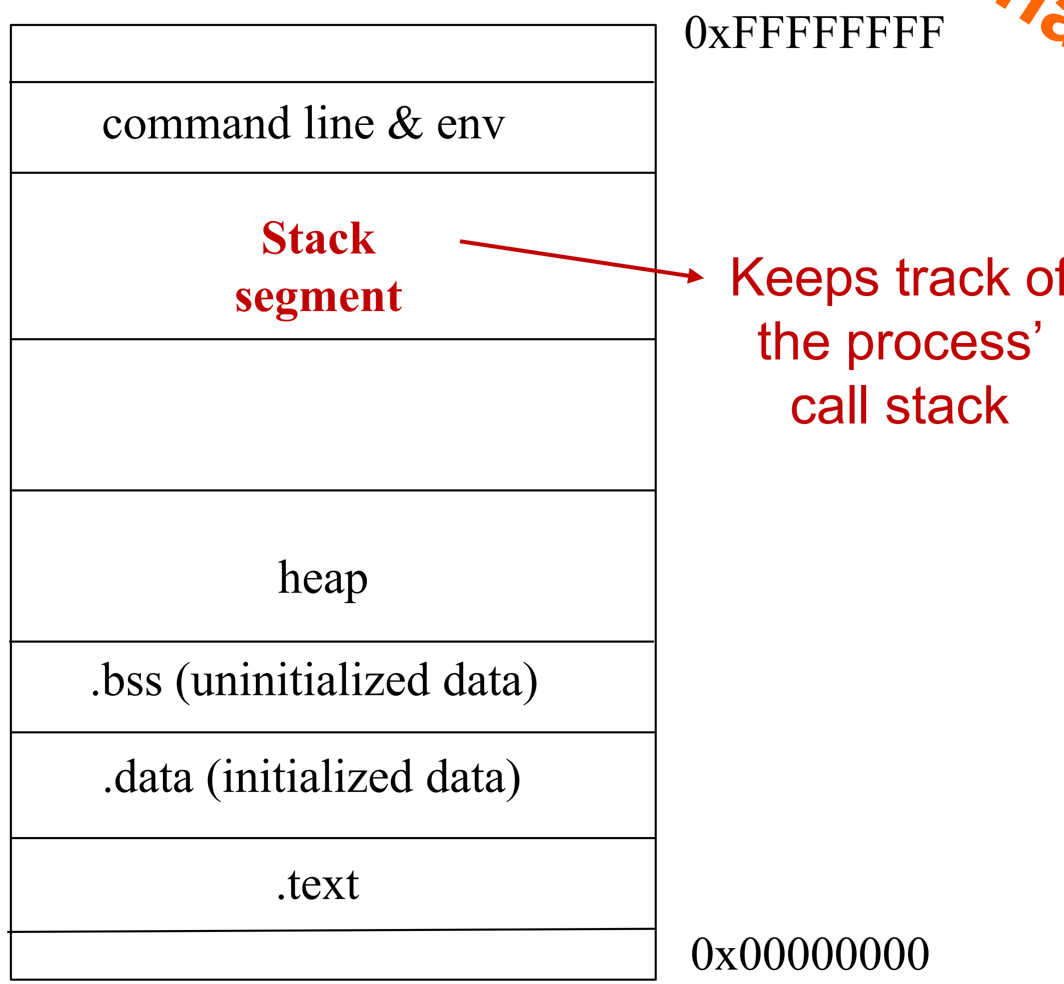
- **Functions** break code into smaller pieces:
  - Facilitate modular design and code reuse
- A function can be called in **many** program locations, e.g. 2, 10, 100, … times (e.g. recursive function)
- **Question 1**: how does the program know where it should continue after it finishes?
- **Question 2**: where are the function's *arguments* and *local variables* stored?

```
void sample_function(void)
   {
        char buffer[10];
        printf("Hello!\n");
        return;
   }
main()
   {
        sample_function();
        printf("Loc 1\n");
        sample_function();
        printf("Loc 2\n");
   }
```

Process Memory Layout

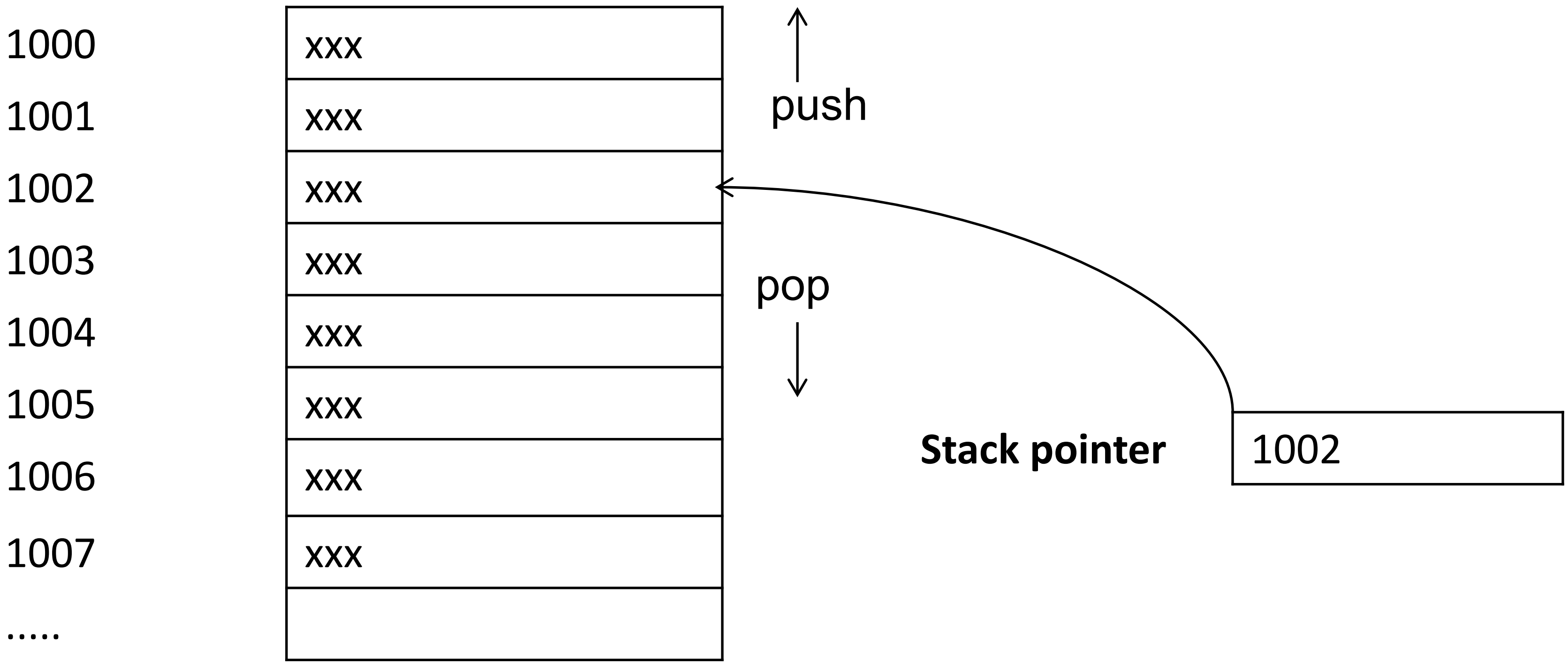
Optional

- Simplified **Linux process memory** showing various *segments*:
- (Optional: http://dirac.org/linux/gdb/02a-Memory_Layout_And_The_Stack.php)



Call Stack

- **Call stack** is a **data structure in the memory** (*not* in a separate hardware): stores important information of a running process
- **Last in, first out (LIFO)**: with push(), pop(), top() operations
- The location of the top element is referred to by the ***stack pointer***



In this example, the stack grows "**upward**", ***but*** from **high addresses** to **low addresses**.
It is possible to have stack that grows downward.

Call Stack

- During a program execution, a stack is used to keep tracks of:
  - **Control flow information**: i.e. return addresses
  - **Parameters** passed to functions
  - **Local variables** of functions
- Each call of a function pushes an ***activation record*** (***stack frame***) to the stack, which contains:
  passed parameters, return address, pointer to the previous stack frame, and function local variables
- **Note**: this stack is known as ***call stack***
  In the context of system security, very often it is simply called the "***stack***".
  Sample expression: "smashing the *stack* for fun and profit".

Illustration: A Function Call

When a function is called, the parameters, return address, and local variables are “**pushed**" into the stack

Consider the following C program segment:

```
int test(int a) {
    int b = 1;
      ...
}
int main() {
      test(5);
      ...
}
```

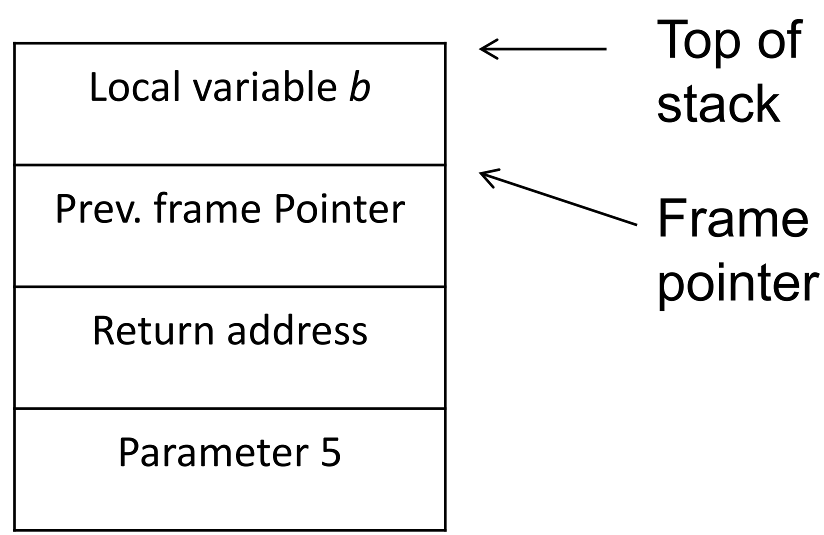


Illustration: A Function Call

When the function `test(5)` is invoked, the following are carried out:

(1) Some values are pushed into the stack:
the parameter (i.e. 5), the return address, the previous frame pointer, and the value of the local variable *b* (i.e. 1)

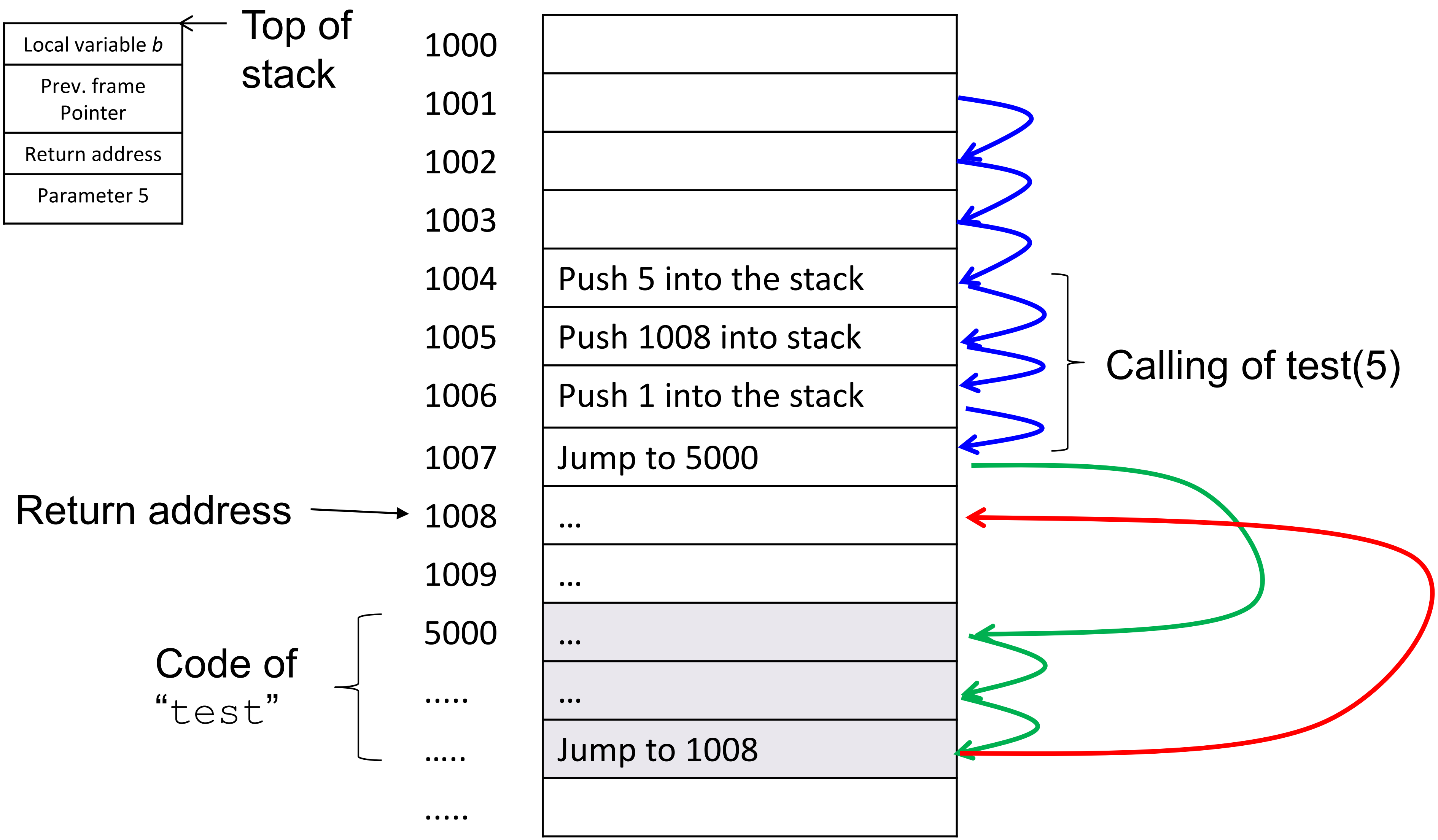
(2) The control flow jumps into the code of "`test`"

(3) Execute "`test`"

(4) After "`test`" is completed, the stack frame is popped from the stack

(5) The control flow jumps into the restored return address

(Simplified) Illustration*



*: This slide gives a simplified view. Actual implementation includes "function return value", and a "frame pointer". For more details, see: http://www.tenouk.com/Bufferoverflowc/Bufferoverflow2a.html or https://en.wikipedia.org/wiki/Stack_buffer_overflow.

7.2.3 Control Flow Integrity

Treating Code as Data: Security Implications

- You have seen how the call stack stores a ***return address*** *(i.e. location of a to-be-executed instruction*) as **data** in the memory
- In fact, the **instruction** itself is stored as data in the memory
- The flexibility of treating code as data is useful, but it leads to **many security issues**
- Attacker could compromise a **process'** ***execution integrity*** by either modifying the process' **code** or the **control flow**
- It is difficult for the system to distinguish those malicious pieces of code from benign data

Compromising Memory Integrity

- In general, it is ***not* so easy** for an attacker to compromise memory integrity
- For **example**, consider an attacker who can only remotely communicate with the targeted Web server via HTTP. How *can* he maliciously write to the web-server's memory?
- One way for the attacker to gain that capability is by: **exploiting some vulnerabilities** so as to "trick" the victim process to **write to** some of its memory locations, e.g. via a "**buffer overflow**" attack
- The above mechanisms typically have **some restrictions**: for example, the attacker can only write to a small number of memory, or can only write a sequence of consecutive bytes. Hence, the attack has to be extremely "**surgical**".

Possible Attack Mechanisms

Assuming that the attacker has the capability to **write** to some memory locations, the attacker could:

(**Attack 1**) Overwrite **existing execution code portion** with malicious code

(**Attack 2**) Overwrite a piece of **control-flow information**:

- (2a) Replace a **memory location** storing a code address that is used by a ***direct jump***
- (2b) Replace a **memory location** storing a code address that is used by an ***indirect jump***

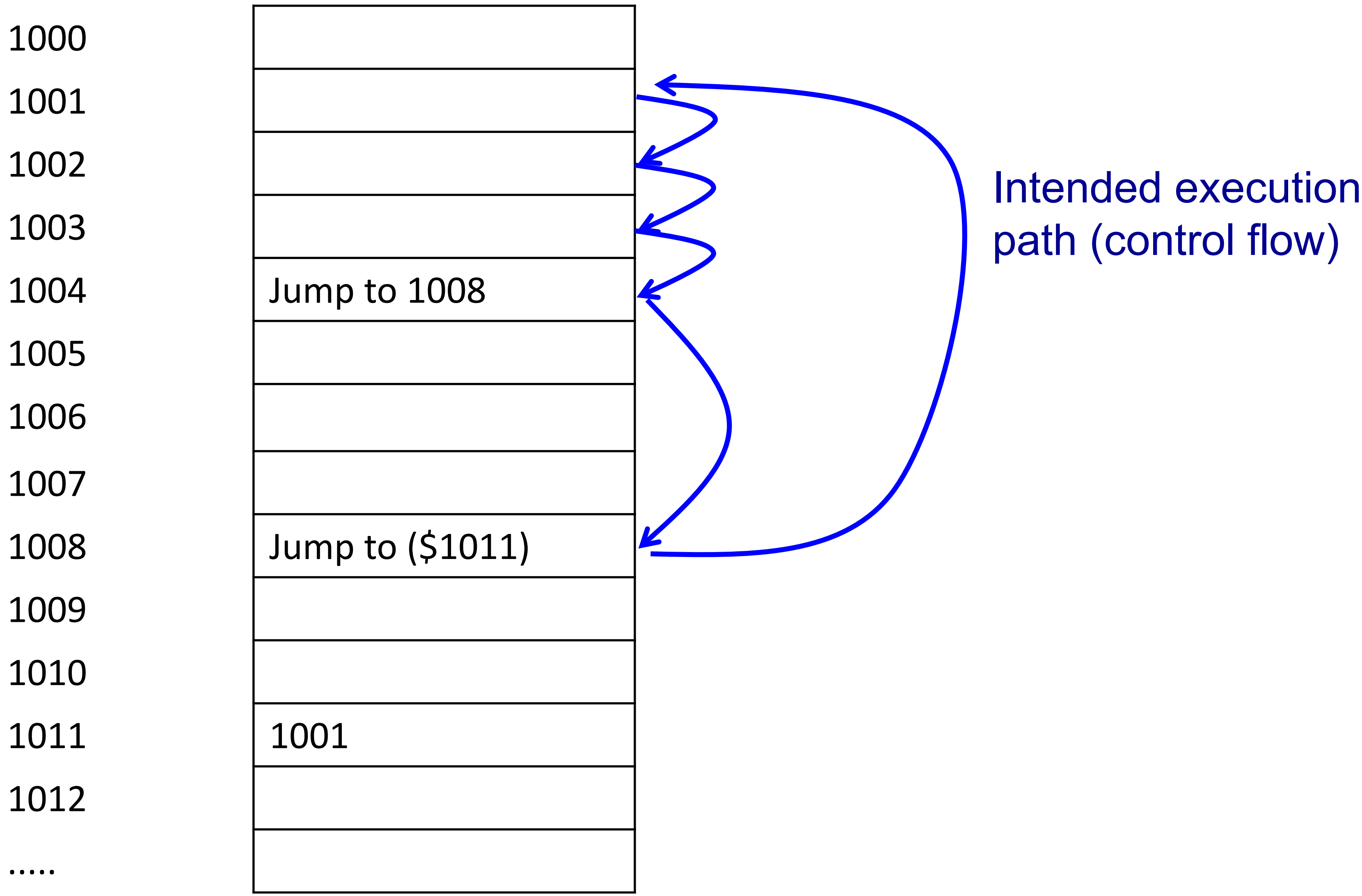
The three attacks above are illustrated in the next few slides

Attack 1 (Replace Existing Code)

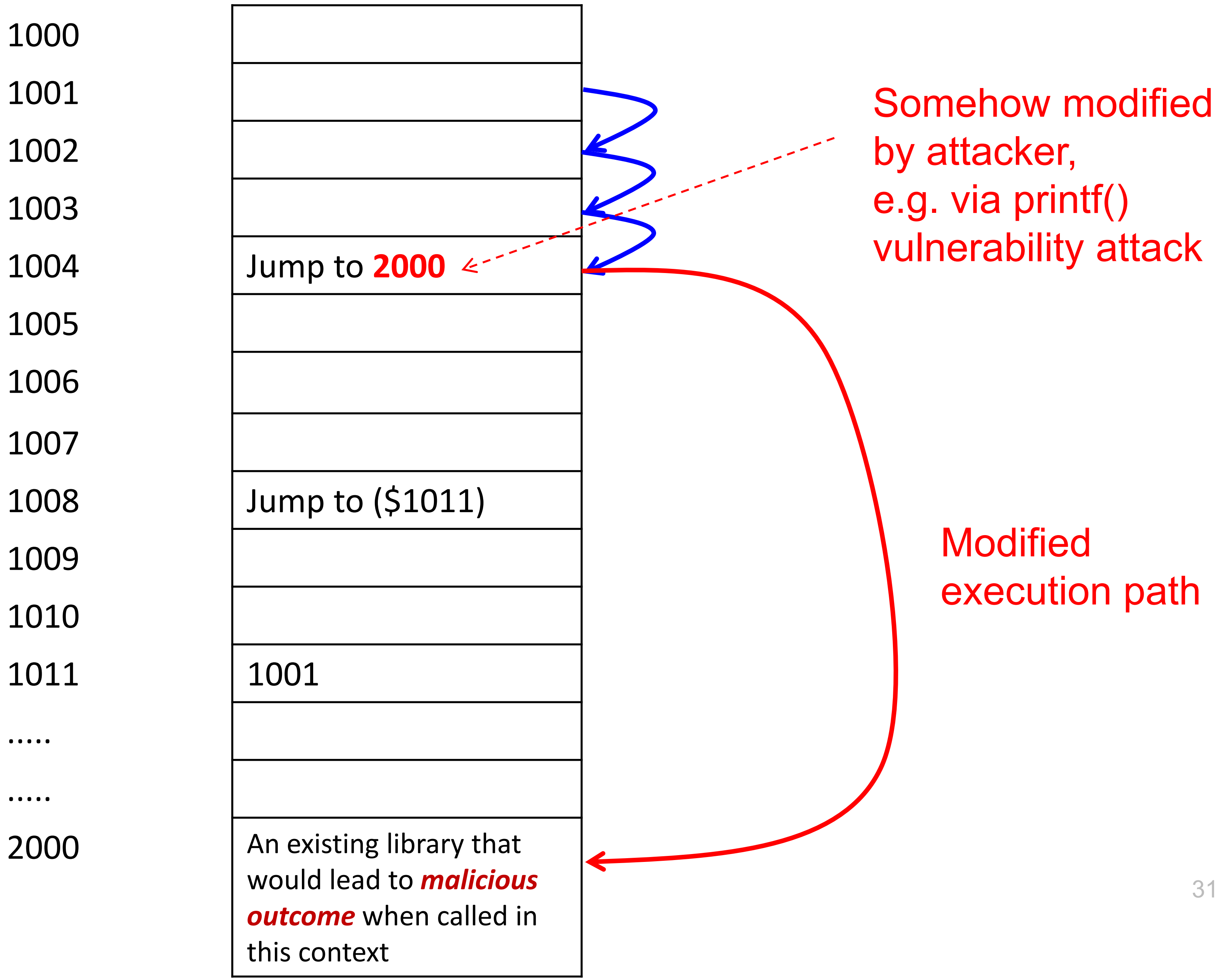
1000	
1001	
1002	
1003	
1004	Jump to 1008
1005	
1006	
1007	
1008	Normal code
1009	Normal code
1010	....
1011	
1012	
.....	

1000	
1001	
1002	
1003	
1004	Jump to 1008
1005	
1006	
1007	
1008	Malicious code
1009	Malicious code
1010	....
1011	
1012	
.....	

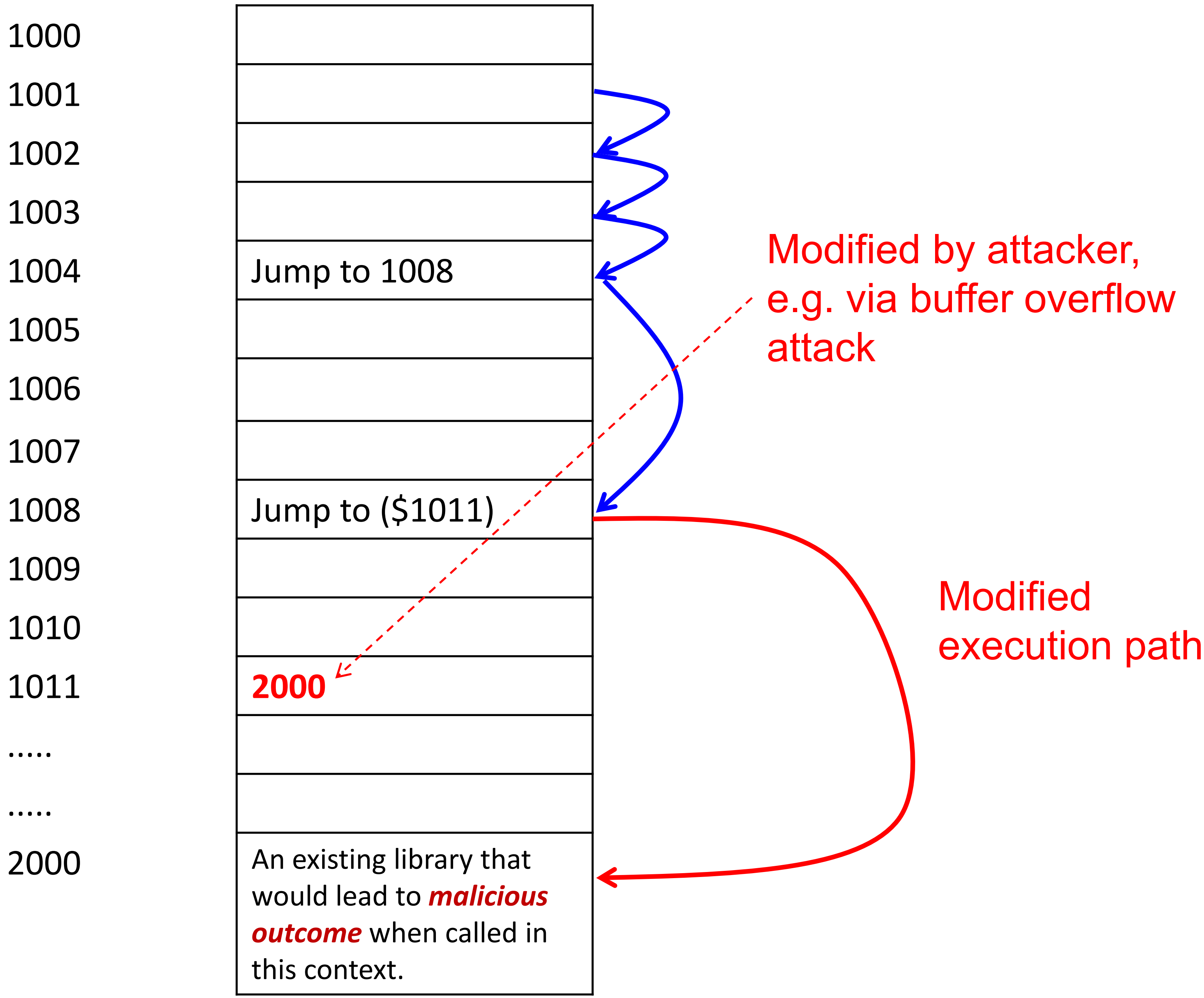
Attack 2a & 2b: Normal Control Flow before Being Attacked



Attack 2a (Replace Memory Location that Stores a Code Address)



Attack 2b (Replace Memory Location that Stores a <u>Code Address</u>)



7.3 printf() and Format String Vulnerability

Read Wiki http://en.wikipedia.org/wiki/Uncontrolled_format_string

Read https://www.owasp.org/index.php/Format_string_attack

For more details, see:
http://www.cis.syr.edu/~wedu/Teaching/cis643/LectureNotes_New/Format_String.pdf

printf() Function

- `printf()` is a C function for **formatting output**
- It is special in that it can take in ***any*** number of arguments: one, two, or more arguments
- General format as written in its function definition:
  `int printf(const char* format, ... );`
- Sample usage with two arguments:
  `printf (format, s);`
  where `format` specifies a format string, and
  `s` is the variable to be displayed
- Hence, `printf (”The value of tmp is %d.\n", tmp);`
  would display the following if `tmp` contains the value 100:
  `The value of tmp is 100.`

printf() Format Specifiers

- The special symbol “%d” indicates the **type** of the variable
- Example:

```
printf ( “1st string is %s, 2nd string is %s”,s1, s2);
```

Hence, `printf()` would:

1. Display “`1st string is `”
2. Look up for the 2nd parameter **in the call stack** and display its value
3. Display “`2nd string is`”
4. Look up for the 3rd parameter **in the call stack** and display its value

printf() Format Specifiers

- Some common format specifiers:
  - %d: decimal (`int`)
  - %u: unsigned decimal (`unsigned int`)
  - %x: hexadecimal (`unsigned int`)
  - %s: string (`(const) (unsigned) char *`)
  - %n: number of bytes written so far, (`* int`)
- Note that %s and %n are passed as a **reference**

The Case of Missing Argument(s)

- When only one parameter is supplied:

  `printf ("hello world");`

  Then, only `"hello world"` will be displayed.

- If there happens to have "%d" in the first parameter, e.g.:

  `printf ("hello world %d");`

  Then, `printf()` will **search for the 2nd parameter** in the stack to be displayed.

  Then what being displayed could be:

  `hello world 15`

- ***Any security implications??***

Example of a Vulnerable Program with Missing Argument(s)

- A format string variable **`t`** is supplied by the user (an attacker)
- The program calls `printf()` using only one parameter **`t`**
- The attacker **can get more information** by carefully designing the string **`t`**

```
#include <stdio.h>

int main()
{
   char t[100];

   scanf ("%s",t);
   printf (t);

   return 0;
}
```

Declare a string of 100 characters

Read in a string and store it in *t*

Display the string *t*

How Such Vulnerability Can be Exploited

- If a program is vulnerable, the attacker might able to:
  - Obtain **more information** of program’s call stack (e.g. using “%d.%d.%d” or “%08x.%08x.%08x”)
  - Cause the program to **crash** (e.g. using “%s%s%s%s%s”):
    - %s will fetch a number from the stack, treat this number as **an address**, and print out the memory contents pointed by this address as **a string** (until a NULL char)
    - Typically one fetched number is ***not* an address**, and thus the memory pointed by this number does not exist, therefore the program will crash
  - **Modify** program’s memory content (e.g. using %n): *not covered in this module*

How Such Vulnerability Can be Exploited

- How the vulnerability can be exploited?
- In a **multi-user setting**:
  if the program has an *elevated* privilege (i.e. set-UID),
  a user (attacker) might be able to obtain ***system-level information***
- In a **client-server setting**:
  if the server program is vulnerable, the client (attacker) might able to **submit a request** and ***obtain sensitive information*** (e.g. a secret key)

Simple Preventive Measures

Avoid taking a user input as format string:

- `printf (t)` Where `t` is supplied by the user, then it is potentially **insecure**
- `printf (f, t)` Where `f` is ***not*** supplied by the user, then it is generally **okay**

Many modern compilers can also **statically check** format strings and **produce warnings** for dangerous or suspect formats

E.g.: in GNU Compiler Collection, the relevant compiler flags are: `-Wall`, `-Wformat`, `-Wno-format-extra-args`, `-Wformat-security`, `-Wformat-nonliteral`, and `-Wformat=2`

Heartbleed Bug: Graphical Illustration of Over-Read Request

Source: http://xkcd.com/1354/