This background is required to appreciate buffer overflow and stack smash.

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

Topic 7
Call Stack

7.1 Background
7.2 Compromising control flow
7.3 Stack (aka Execution Stack, Call Stack)

We have seen confidentiality, authenticity of data and availability of services. Now, integrity of processes.

Summary and takeaways

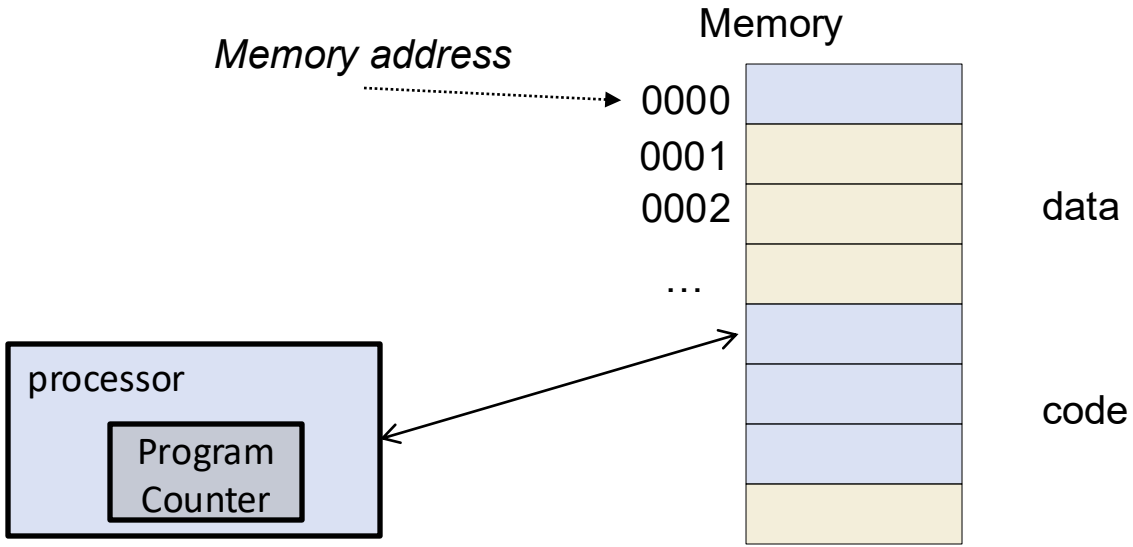
- Demonstrate how process integrity can be compromised by modifying values in memory.
- ***Call stack*** facilitates function calls by maintaining and keeping track of ***runtime environments***.
- An element in the stack is a "stack frame". It contains runtime info such as control flow info (return address), local variables, and parameters. Malicious modification of those info would have significant consequence (next lecture on stack smash + buffer overflow).
- ***Stack Smack***: Due to how OS/compiler maintain the runtime environments, there are opportunities for an attacker to compromised its victim's call stack (e.g. buffer overflow of local variables into return address).

7.1 background

Many concepts in software security requires good understanding of the “runtime environment”.

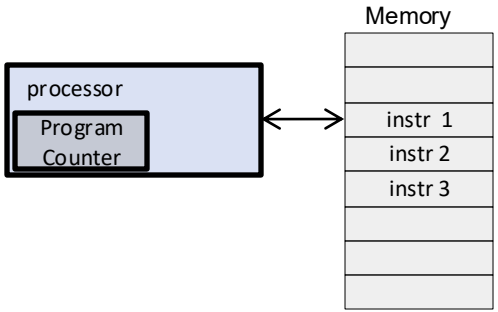
Code vs Data

- Modern computers are based on the *Von Neumann computer architecture.* The *code* are treated as *data* and both are stored in the memory. There are no clear distinction of code and data. The “program counter” indicate location of next instruction to be executed.



Control Flow

- The program counter (aka Instruction Pointer) is a register (i.e. small & fast memory within the processor) that stores the address of the next instruction.



- The following 3 steps keep looping:

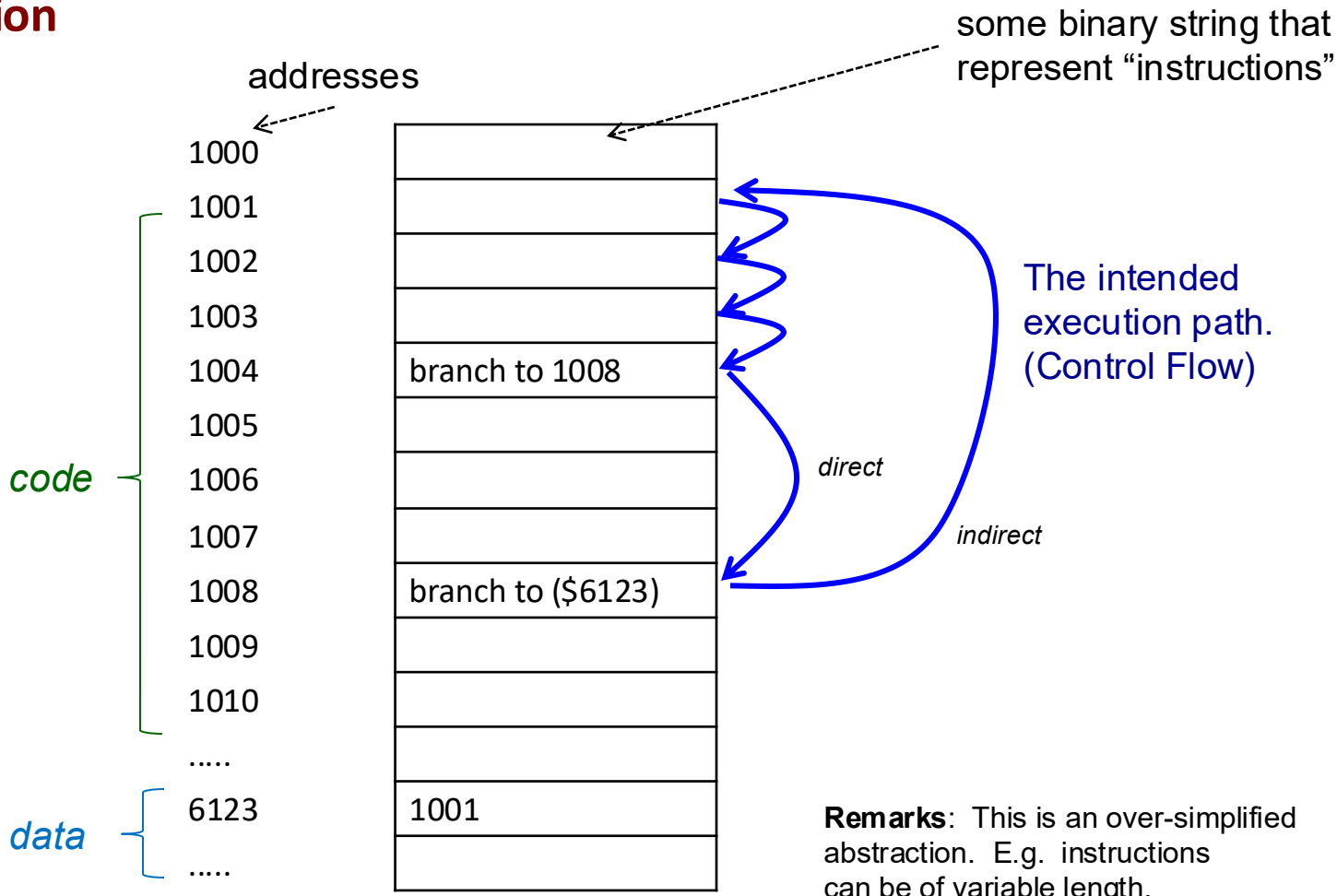
1. The processor fetches the data in location pointed by the PC.
2. Increase the PC by 1.
3. The processor treat the fetched data as an instruction and execute it.

Note that by looping the above, instructions would be executed sequentially in the memory.

- During execution, program counter could also be changed by the fetched instruction, including*,
  1. (direct branch) replaced by a constant value specified in the instruction;
  2. (indirect branch) replaced by a value fetched from the memory. There are many different forms of indirect branch.

*: for simplicity, we omit conditional branch and call/return here.

illustration



Remarks: This is an over-simplified abstraction. E.g. instructions can be of variable length.

7.2 Control flow integrity

Compromising memory integrity → control flow integrity

- Suppose an attacker can modify some memory, the attacker could compromise the execution integrity by either:
  - directly modifies code in the memory; or
  - modifying the address in indirect branch.

- E.g., by exploiting buffer overflow, the attacker could write to certain memory that the attacker originally doesn't have access to.

- While possible, it is still not easy for an attacker to compromise memory integrity. In addition, it may come with some restrictions. For e.g. attackers can only write to some specific locations, or the attacker can only write a sequence of consecutive bytes, or the attacker can write but not read, etc.

Possible Attack Mechanisms

Let us assume that the attacker has the capability to *write* to some memory locations and wants to compromise the execution integrity. The attacker could:

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

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

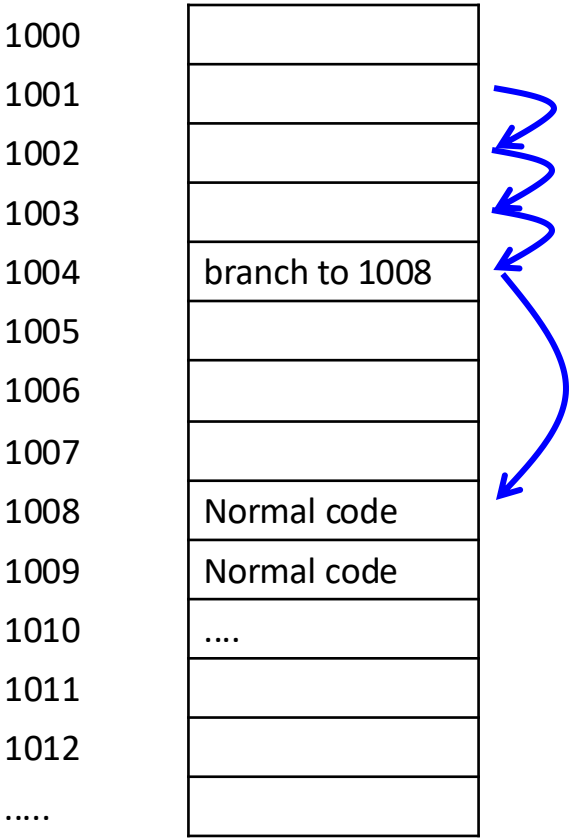
(2a) Replace a ***memory location*** storing a code address that is used by a *direct jump*

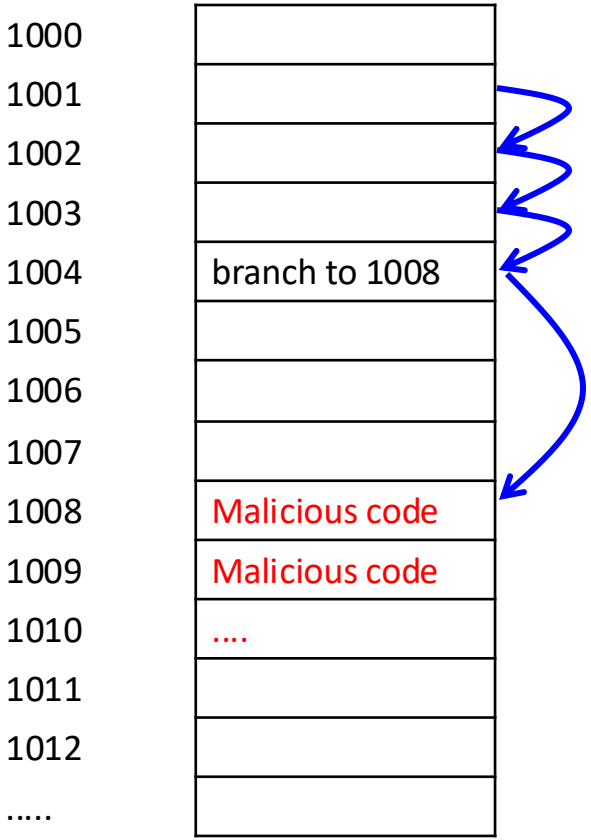
(*2b*) Replace a ***memory location*** storing a code address that is used by an *indirect jump*

The above three attacks are illustrated in the next few slides.

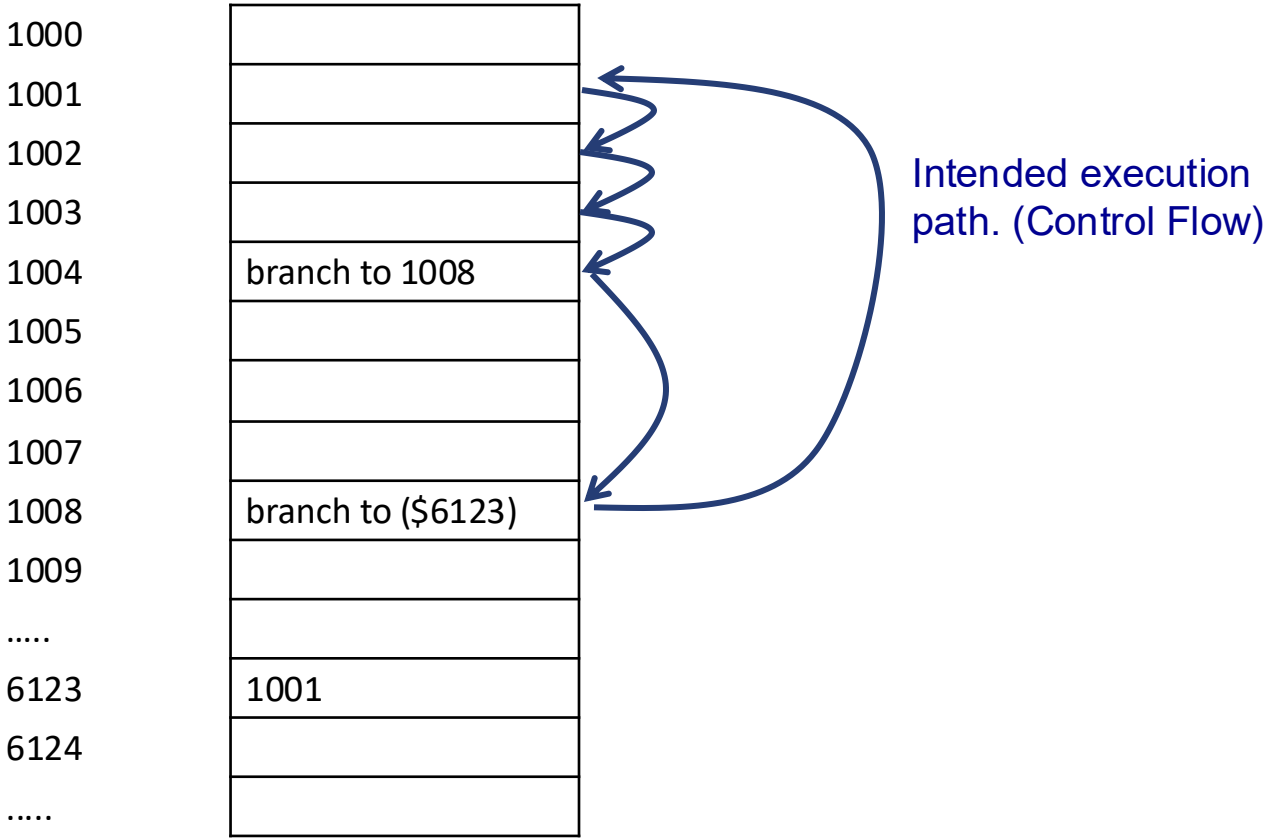
When Attack (2b) is carried out on stack, it is aka ***Stack smashing***.

Attack 1 (replace the code)

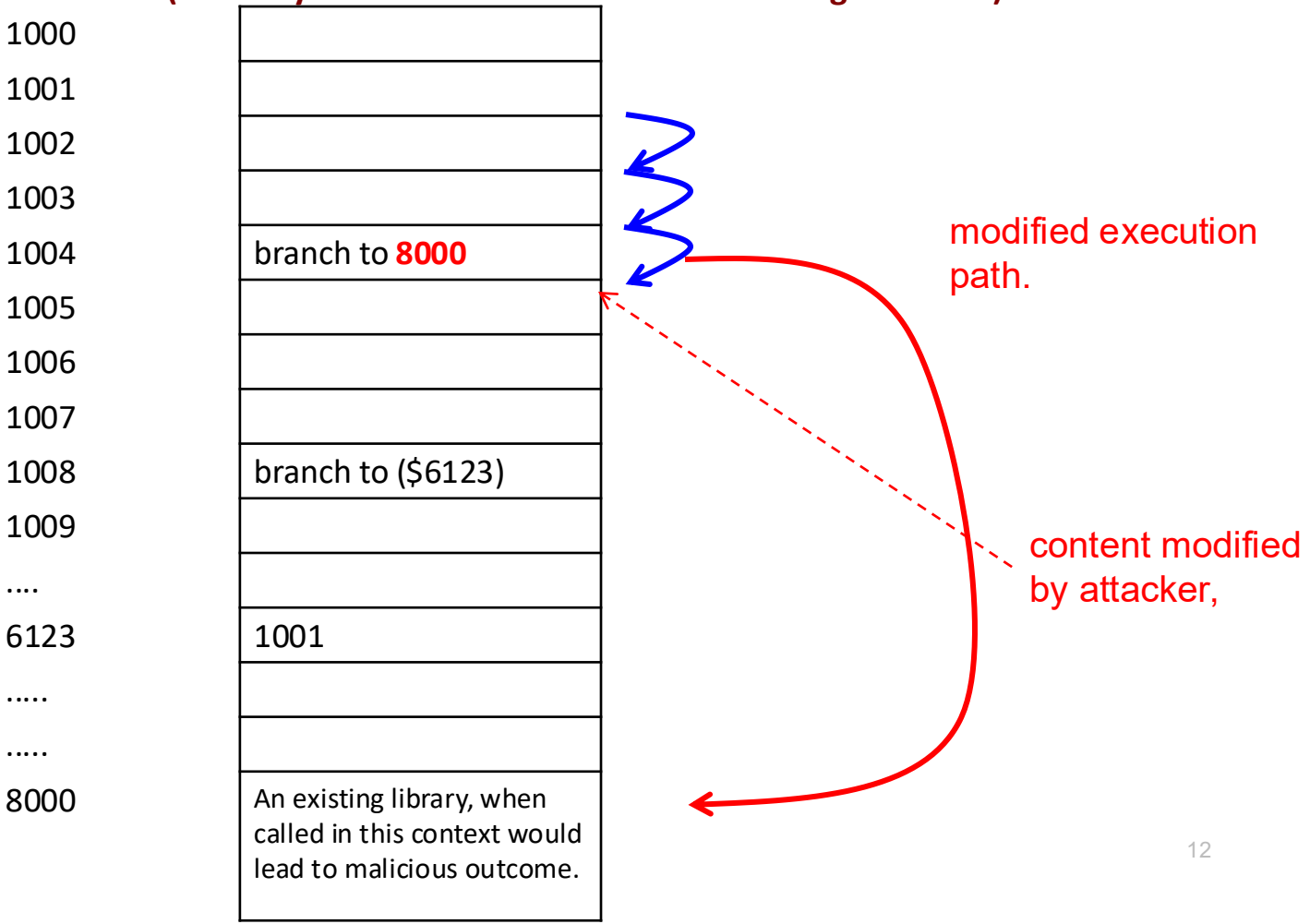




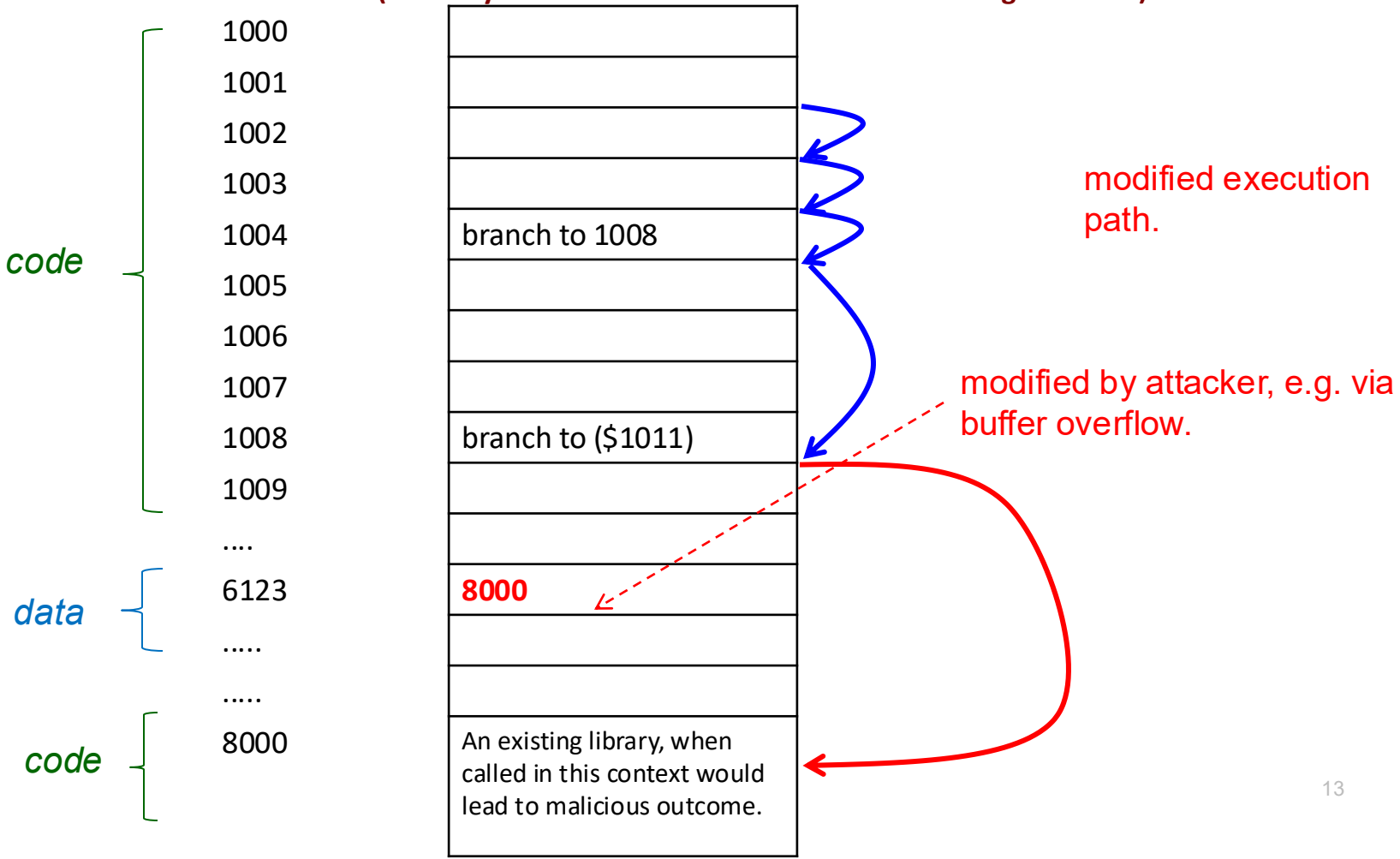
Attack 2a, 2b: Normal control flow before being attacked.



Attack 2a (Memory locations that store the code being modified)



Attack 2b (Memory locations that store the addresses being modified)

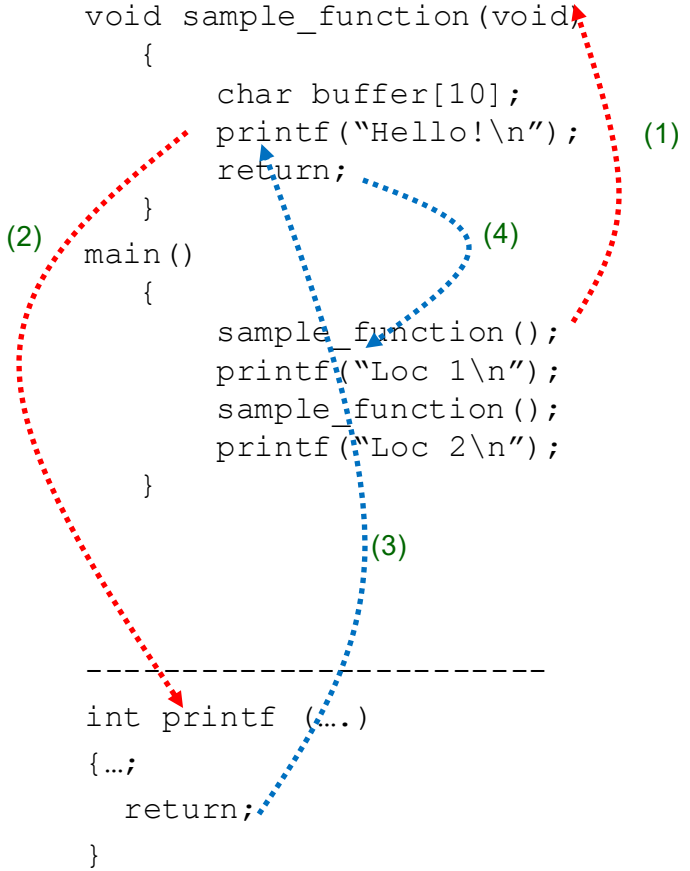


7.3 Stack
(aka Execution Stack, Call Stack)

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

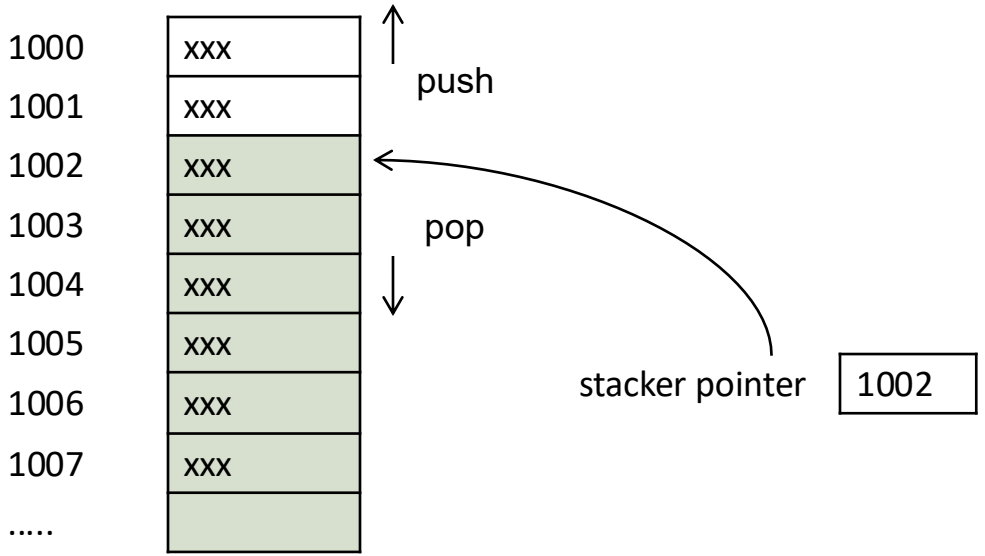
Functions

- A function can be called from different part of the program, and even recursively.
- ***Question* 1**: how does the control flow knows where it should return to after a function is is completed?
- ***Question* 2**: where are the function's "runtime environment" stored. That is, where are the *arguments* and *local variables*? (recap that in a recursive function, a local variable could concurrently hold many values.)
- These are managed by "call stack".



Remark: Stack

- Stack is a data structure resides in the memory.
- A call-stack stores runtime environment for each function instance. The runtime environment of an instance is stored in a single "stack frame". A frame consists of multiple bytes.
- *Stack pointer* is a variable that stores location of the first element. There are two operations in stack: Push and Pop. (Last-In-First-Out).



Note: In this example, the stack grows "upward".

Call stack

- Recap that stack is last-in-first-out.

- During execution, a ***Call Stack*** is maintained to keep track of the runtime environment.

- The runtime environment (let's call it a ***stack frame***) of an instance includes:
  - Parameters to be passed in and result to be returned
  - Return address (i.e. the address to return to after the function completed.)
  - Local variables of functions

  (For efficiency, the stack frame include a pointer to the previous stack frame. This pointer is included for efficiency. Let's ignore the role of this. We mentioned it here since most documents mentioned this)

- (push) When a function is being called, its stackframe is pushed in.
- (pop) When the function completed, the stack frame is poped out.

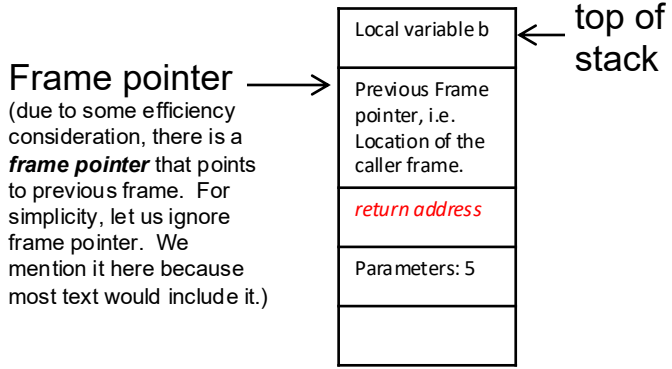
Illustration.

When a function is invoked, the runtime environment is "pushed" into the stack.

E.g. for the following segment of C program:

```
int Drawline( int a)
{int b =1;
..
}

int main()
{
   Drawline (5);
}
```

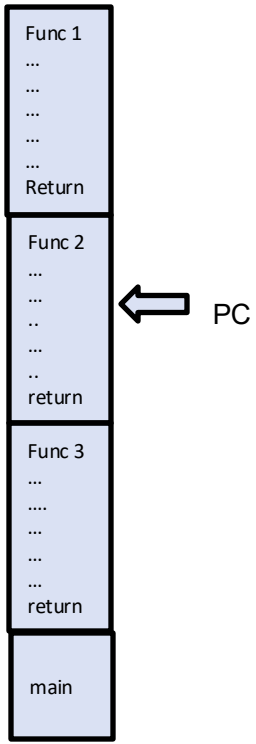


When the function `Drawline(5)` is invoked, the followings steps are carries out:

(1) These data are pushed into the stack in this order:
- the parameter (which is "5"),
- the "*return address*" (i.e. value of program counter), and
- the value of the local variable b (which is 1).

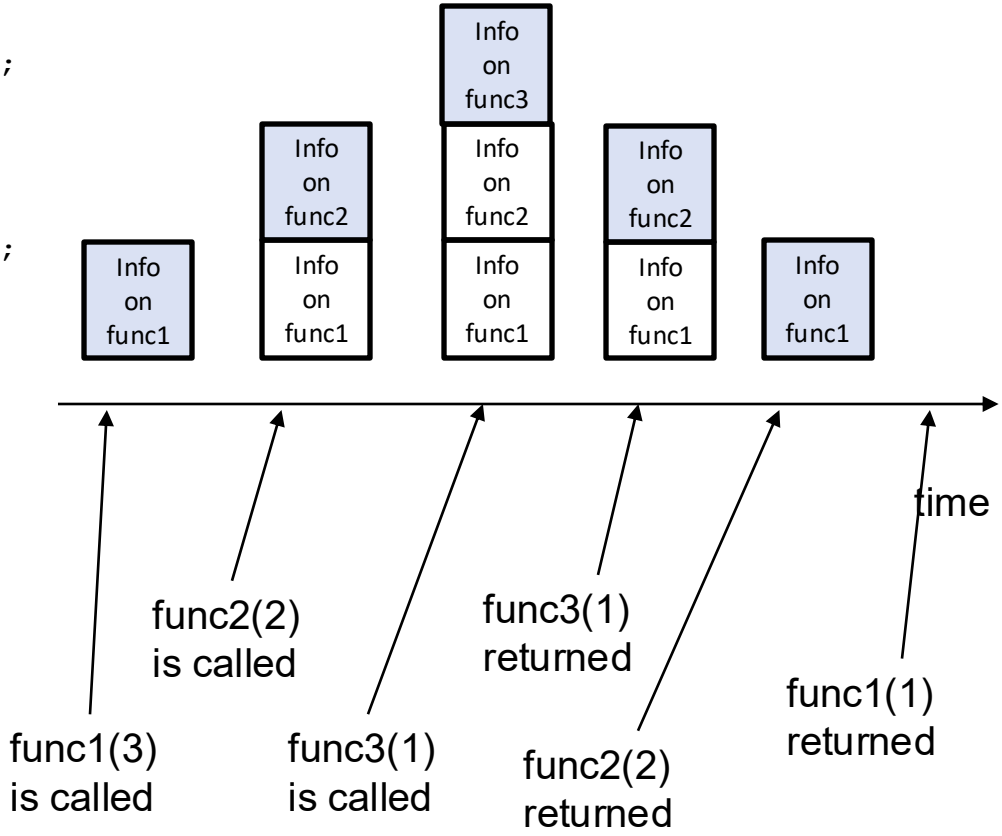
(2) The control flow branches to the code of "Drawline".
(3) Execute "Drawline".
(4) After "Drawline" is completed, pops out the variables, "*return address*" and parameter.
(5) Control flow branches to "*return address*".

memory

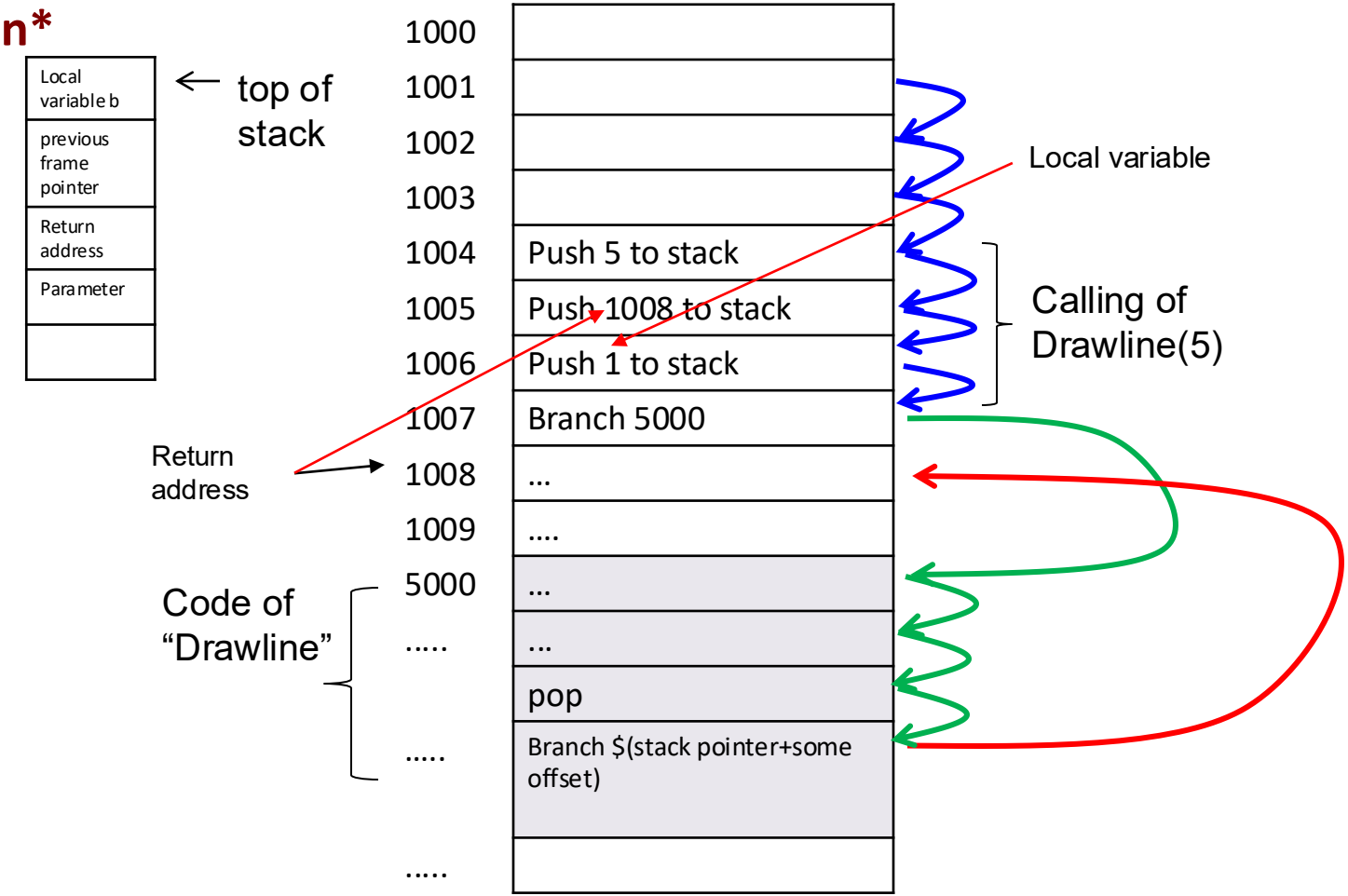



```
int func1( int a)
{int b =253;
        func2( 2);
}
int func2( int a)
{int b =15;
        func3( 1);
}
int func3( int a)
{int b =0;
}


int main()
{
    func1(3);
}
```



Illustration*



*: This slide gives a simplified view. For more details, see http://www.tenouk.com/Bufferoverflowc/Bufferoverflow2a.html or https://en.wikipedia.org/wiki/Stack_buffer_overflow

Security implication.

- Using buffer overflow on the local variables, attacker could modify the stack. (This is called stack smash).

- Stack smash has two effects:
  - If return address is modified, the control flow integrity is compromised.
  - If local variables are modified, the computed result would be wrong.