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

Lecture 7:
Call Stack

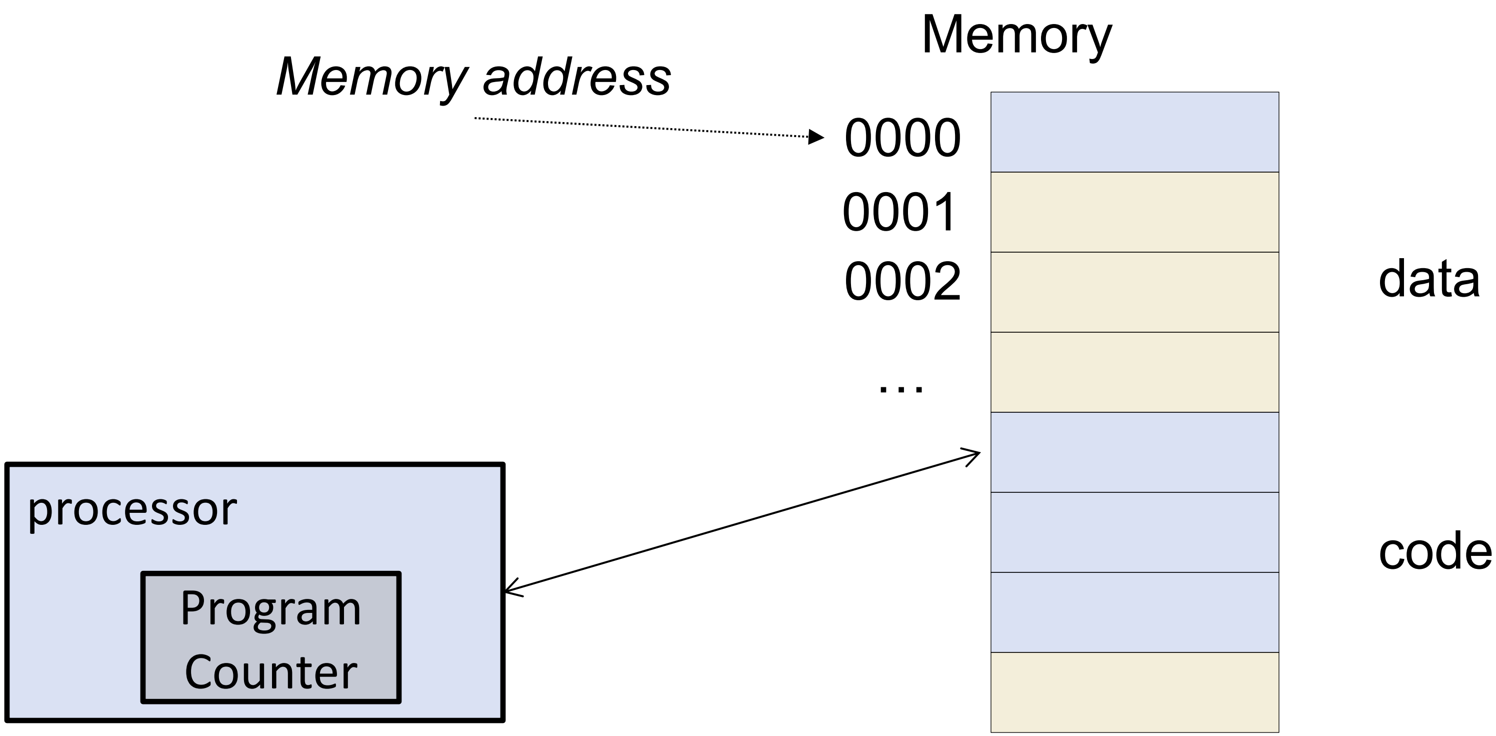
7.1 Background.
7.2 Stack (aka Execution Stack, Call Stack).
7.3 Compromising control flow.

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

7.1 background

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 same memory. There are no clear distinction of code and data. The “program counter” indicate location of next instruction to be executed. (In contrast, the *Harvard architecture* has separated hardware component that stores the code and data).



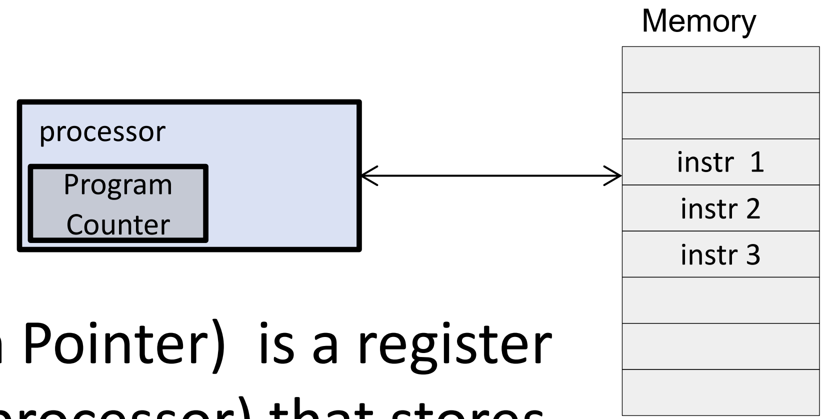
Security implication

- Serious implication: programs may be tricked into *treating input data as code.*

- Other systems, in particular *interpreted languages* (e.g. JavaScript, most scripting languages), also has similar feature *data* can be *code.*

- Such difficulty form the basis of *code-injection attacks!*

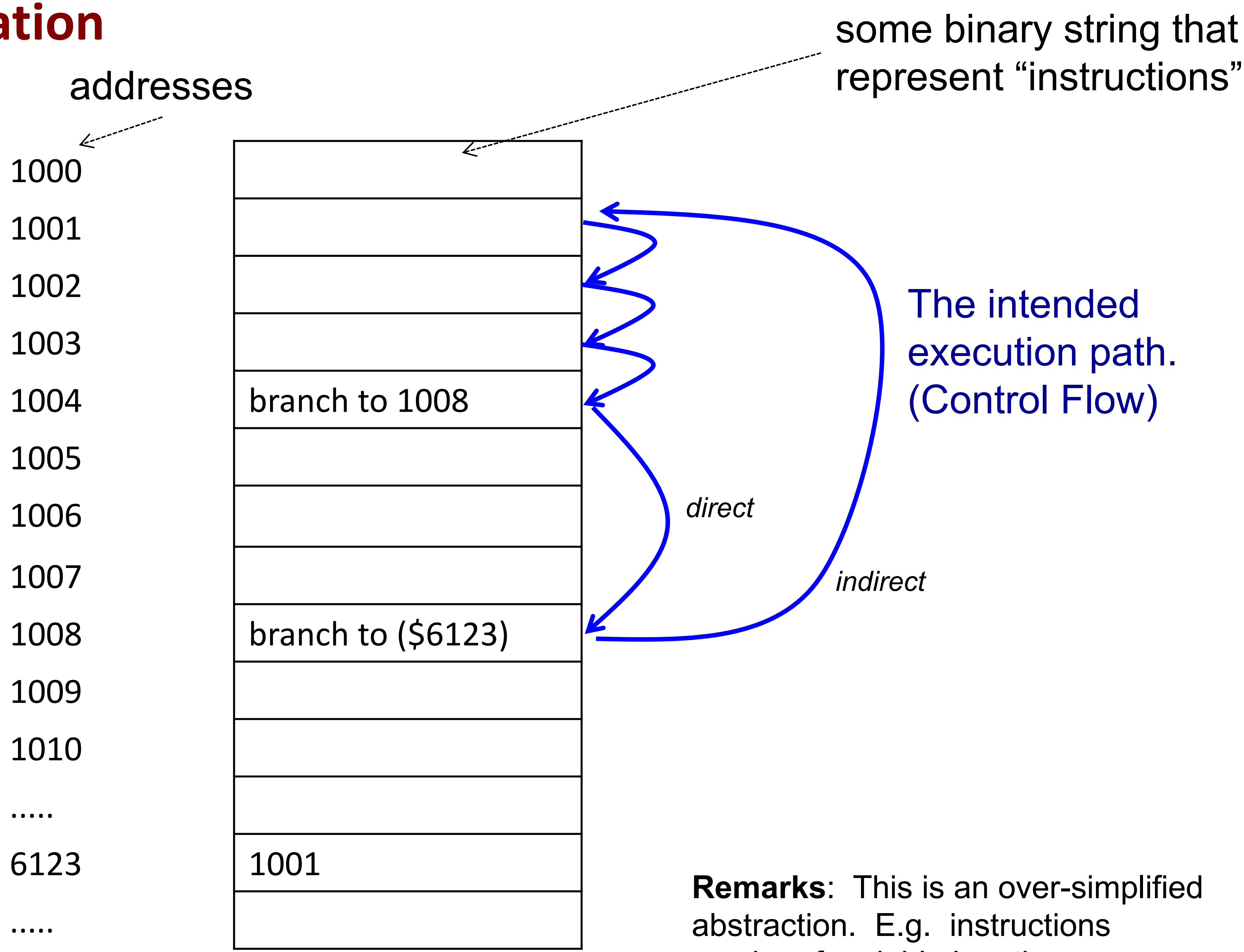
Control Flow



- 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.
- After an instruction is completed, the processor fetches the next instruction from the memory, whose address is specified in the program counter. After the new instruction is fetched, the program counter automatically increases by 1.
- During execution, program counter could also be changed by, e.g*,
  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 Stack
(aka Execution Stack, Call Stack)

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

Functions

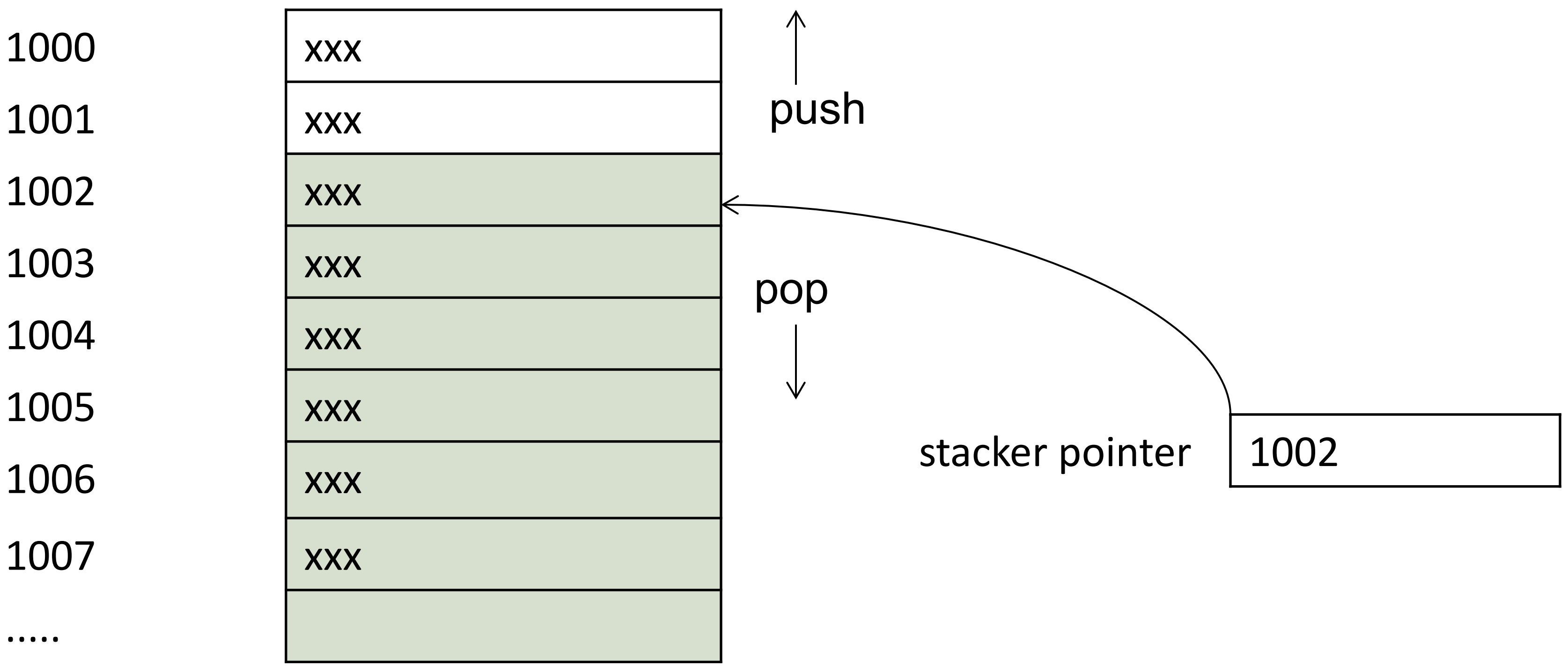
- Subroutine/function/procedure breaks code into smaller pieces:
  - Facilitate modular design and code reuse
- A function can be called from different part of the program, and even recursively.
- ***Question* 1**: how does the system knows where it should return to after a function is is completed?
- ***Question* 2**: where are the function's *arguments* and *local variables* stored?
- These are managed by "call stack".

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

------------------------
int printf (….)
{…;
  return;
}
```

Remark: Stack

- Stack is a data structure. It is not a separate hardware.
- A call-stack is a stack that store important information of the function calls.
- The location of the top element is called the *stack pointer.* There are two operations in stack: Push and Pop. (Last-In-First-Out).



Note: In this example, the stack grows “upward”.

Call stack

- Stack is a useful data structure. During execution, a stack is maintained to keep control flow information and passing of parameters. This stack is known as ***Call stack***. Very often, we simply called it "stack".

- During a program execution, a stack is used to keep tracks of
  - Parameters passed to functions
  - Control flow information: return addresses
  - Local variables of functions

- Each call/invocation of a function pushes an activation record (aka ***stack frame***) to the stack, which contains:
  - parameters,
  - return address,
  - local variables, and
  - pointer to the previous stack frame in the stack. (this is included for efficiency. Conceptually, this piece of info is not required to maintain the control flow. Let's ignore the role of this. We mentioned it here since most documents mentioned this)

Illustration.

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

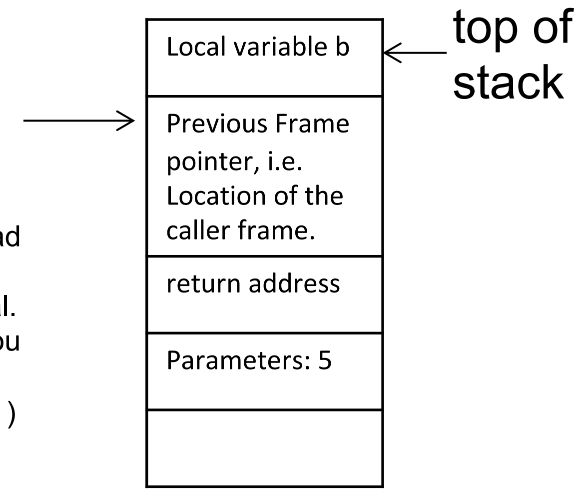
E.g. for the following segment of C program:

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

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

Frame pointer

(due to some efficiency consideration, ***frame pointer*** points here instead of top-of-stack. Conceptually not essential. Mentioned here so that you won’t got confused when reading other documents.)

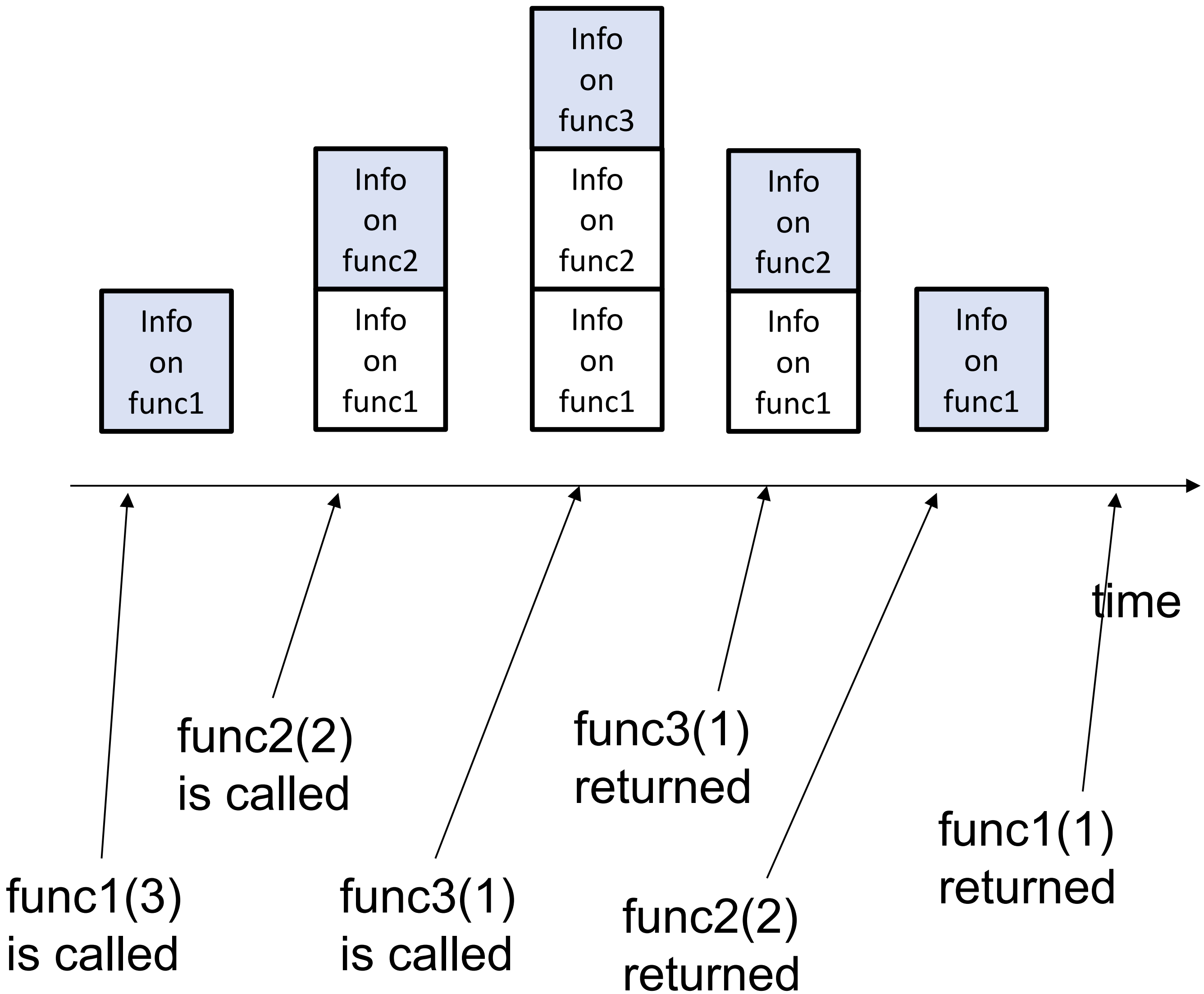


When the function `Drawline(5)` is invoked, the followings steps are carries out:

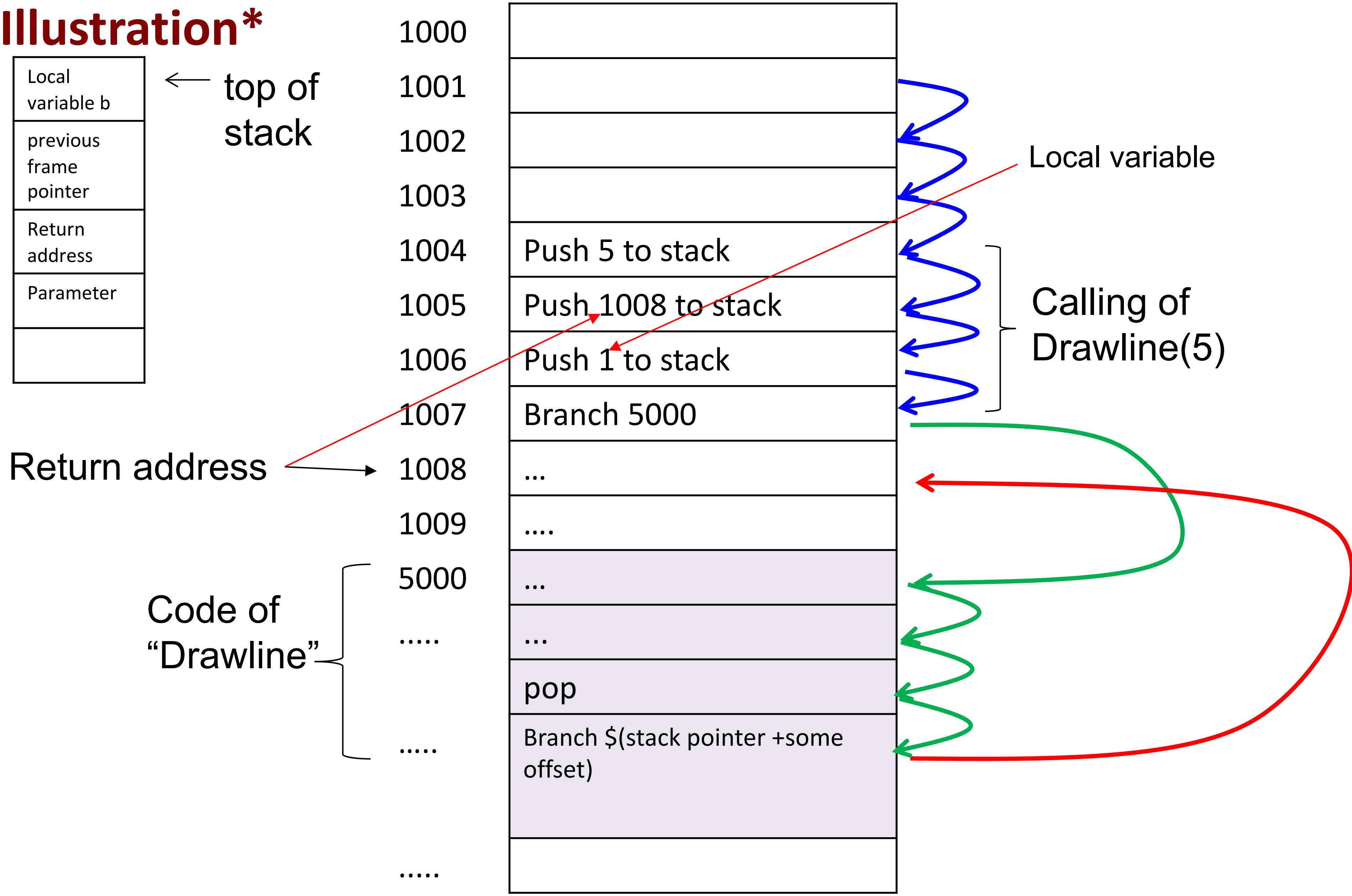
(1) These data are pushed into the stack in this order: the parameter (which is 5), the return address, and the value of the local variable b (which is 1).
(2) The control flow branches to the code of “Drawline”.
(3) Execute “Drawline”.
(4) After “Drawline” is completed, pops out the variable, return address and parameter.
(5) Control flow branches to the return address.

```
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. Actual implementation includes "return value", and involves the "frame pointer". For more details, see http://www.tenouk.com/Bufferoverflowc/Bufferoverflow2a.html or https://en.wikipedia.org/wiki/Stack_buffer_overflow

8.2 Control flow integrity

Remarks: implications on security.

- We have seen how the call stack stores the the *return address* as data in the memory.

- Code is stored as data.

- Attacker could compromise the execution integrity by either modifying the code or modifying the control flow, in particular, the return address in the call stack.

1 time environment - variables and functions local to that call

Compromising memory integrity → control flow integrity

- Let's assume that the attacker has the capability to read/write some memory (i.e. able to compromise ***memory integrity***). In the next few slide, we show that this can lead to compromise of the ***control flow integrity***.
- One way for the attacker to gain that capability (of writing to memory) is by exploiting some vulnerabilities, for e.g., "buffer overflow"* .
- Nevertheless, it is not so easy for an attacker to compromise memory. In addition, it may come with some restrictions. For e.g. attackers can only write to some particular location, or the attacker can only write a sequence of consecutive bytes, or the attacker can write but not read, etc. In other words, although we assume that attacker can compromise memory, they only have limited and restricted accesses.

Possible Attack Mechanisms

Assuming that the attacker has the capability to *write* to some memory locations and want 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 three attacks above are illustrated in the next few slides.

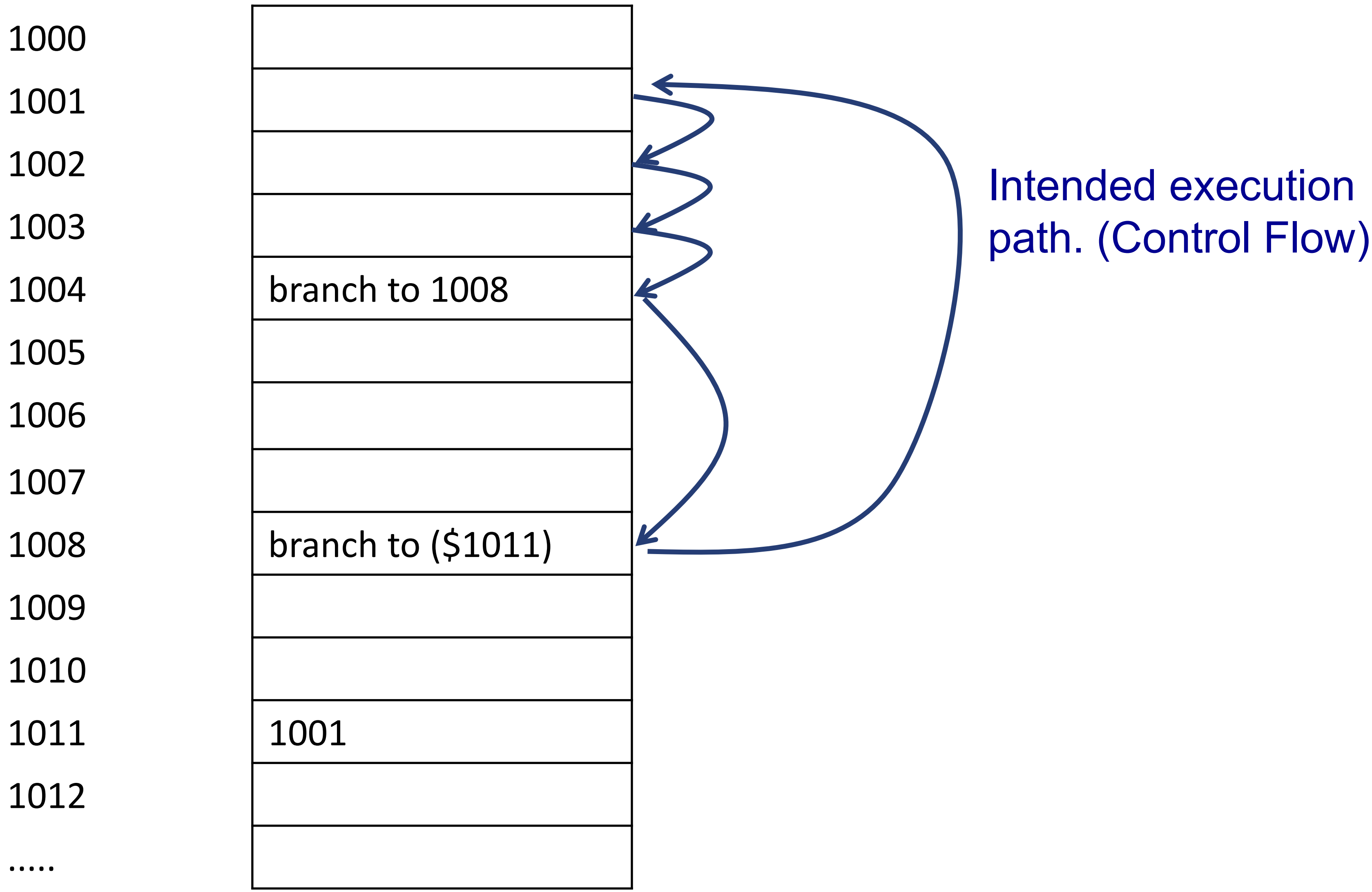
When Attack (2b) can be carried out using ***Stack smashing***. Stack smashing are attack that modify data in the call stack.

Attack 1 (replace the code)

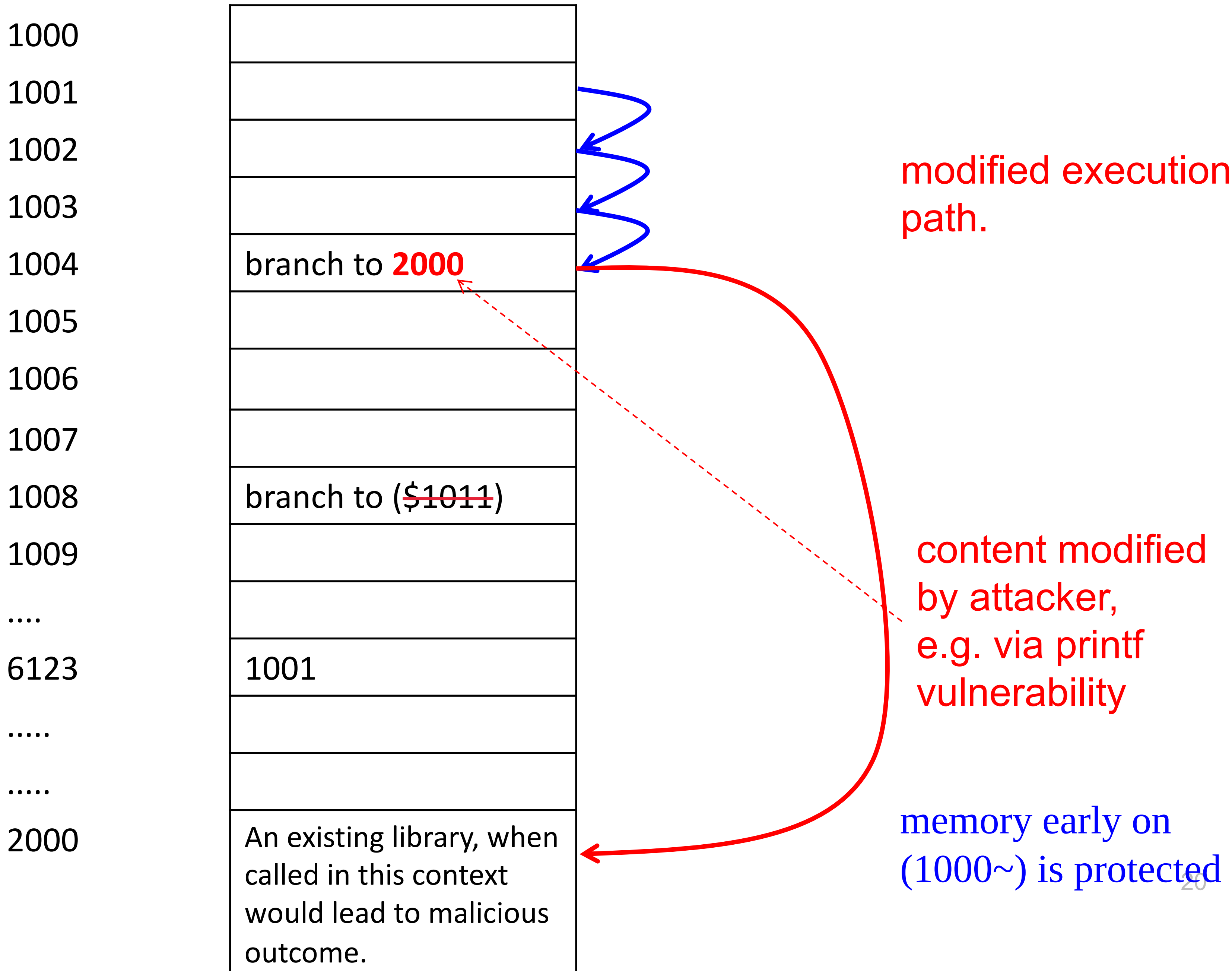
1000	
1001	
1002	
1003	
1004	branch to 1008
1005	
1006	
1007	
1008	Normal code
1009	Normal code
1010	....
1011	
1012	
.....	

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

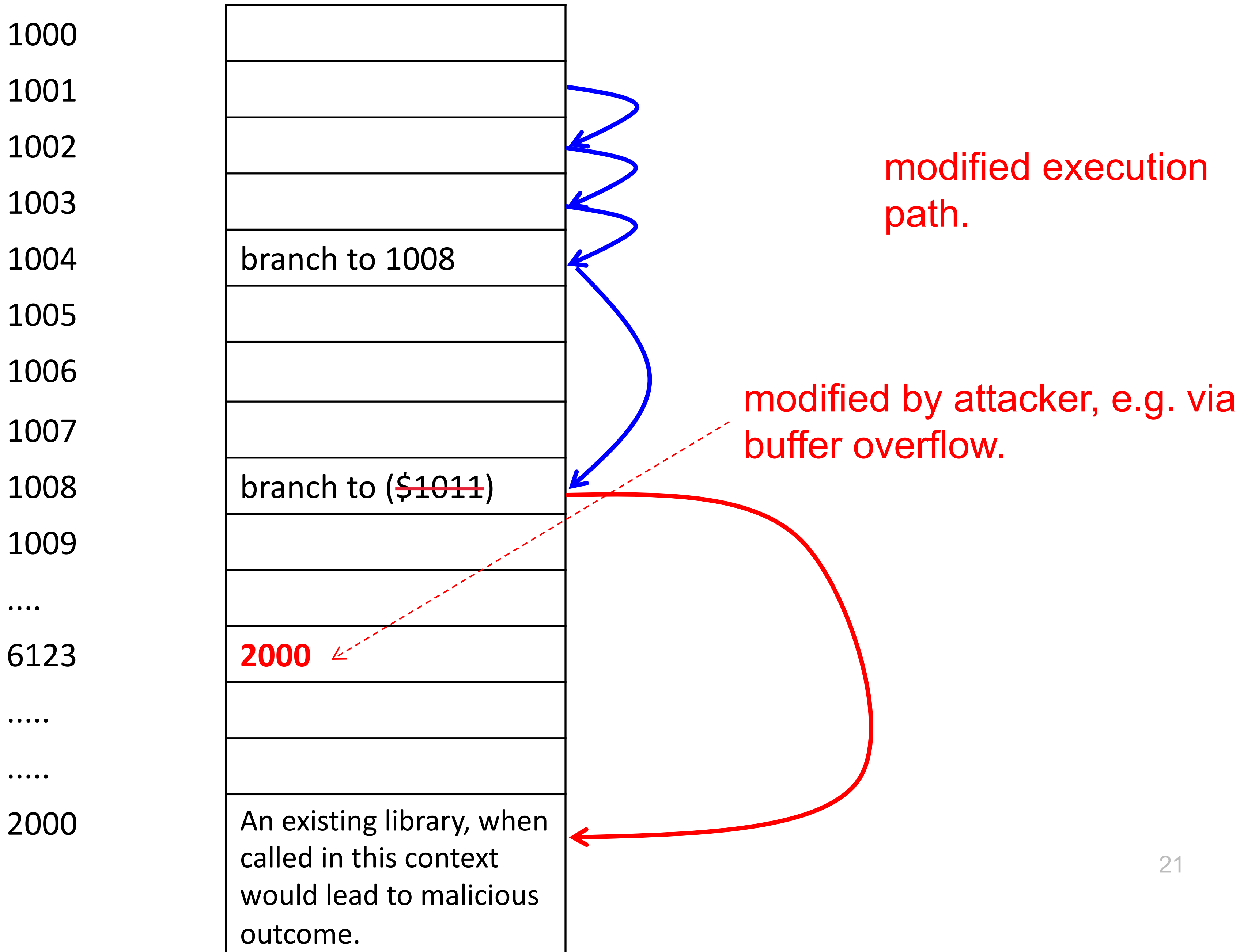
Attack 2a, 2b: Normal control flow before being attacked.



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



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



Remark: Process Memory Layout

Optional

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