A (really) simple introduction to buffer overflows

Herbert Bos

Vrije Universiteit Amsterdam

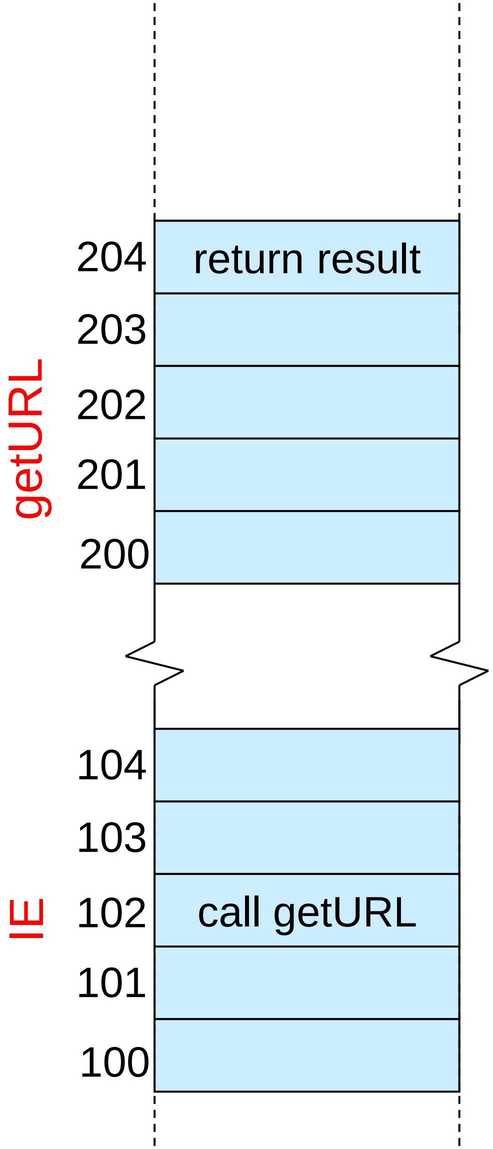
Exploits

- program has a security hole
- exploit = input that abuses the vulnerability

- In this module we will discuss an example: the Buffer overflow

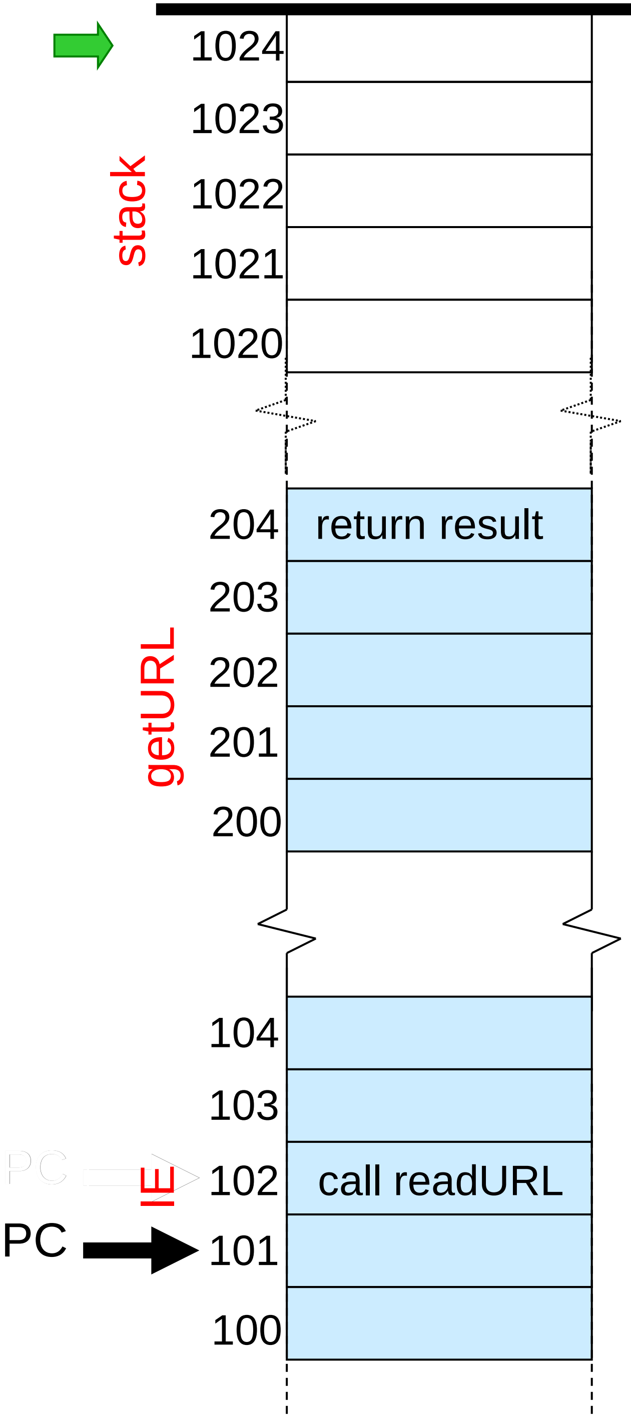
software

- sequence of instructions in memory
- logically divided in functions that call each other
  - function ‘IE’ calls function ‘getURL’ to read the corresponding page
- in CPU, the program counter contains the address in memory of the next instruction to execute
  - normally this is the next address (instruction 100 is followed by instruction 101, etc)
  - not so with function call



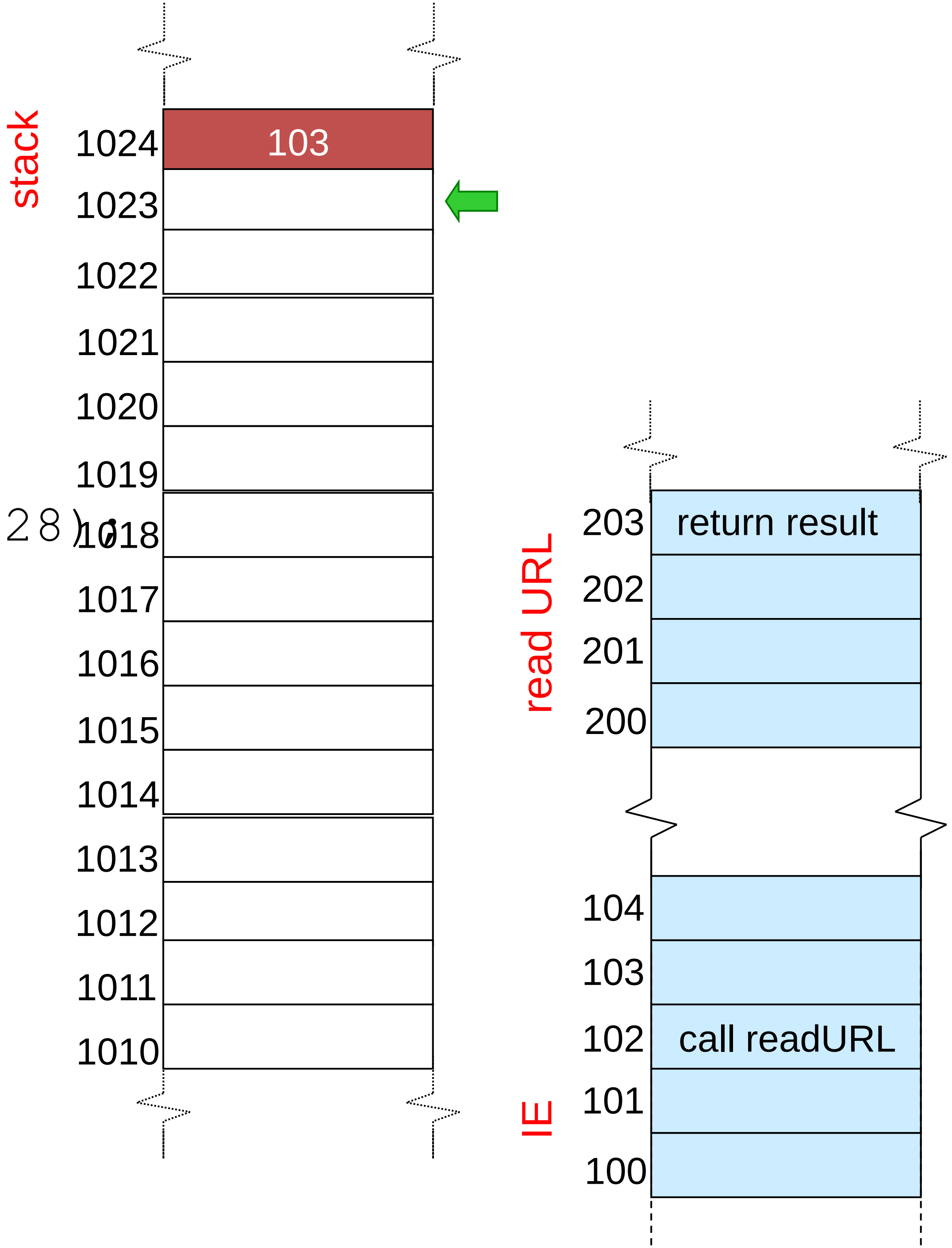
software

- so how does our CPU know where to return?
  - it keeps administration
  - on a ‘stack’



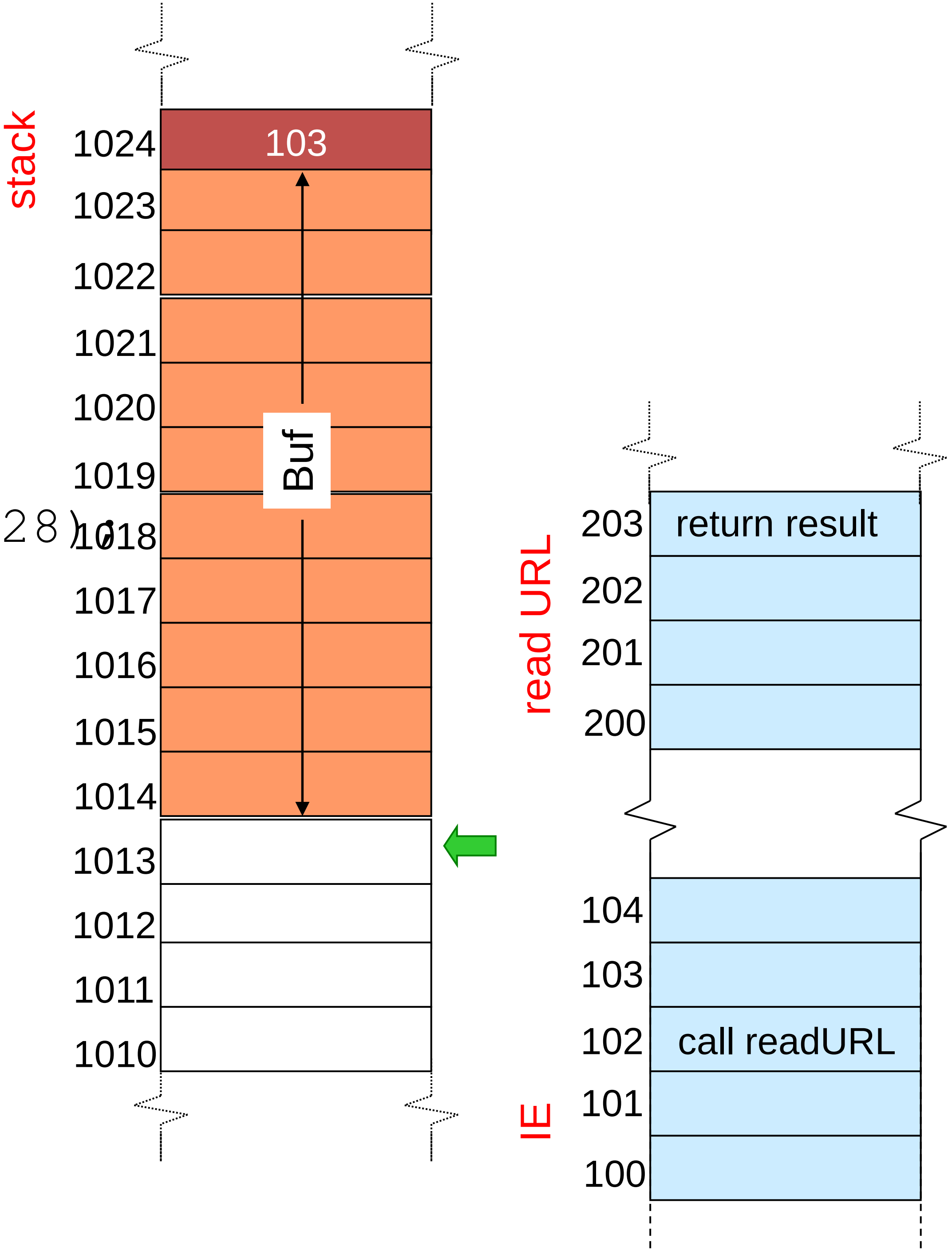
real functions have variables

```
getURL ()
{
   char Buf[10];
   read(keyboard,Buf,128);
   get_webpage (Buf);
}
IE ()
{
   getURL ();
}
```



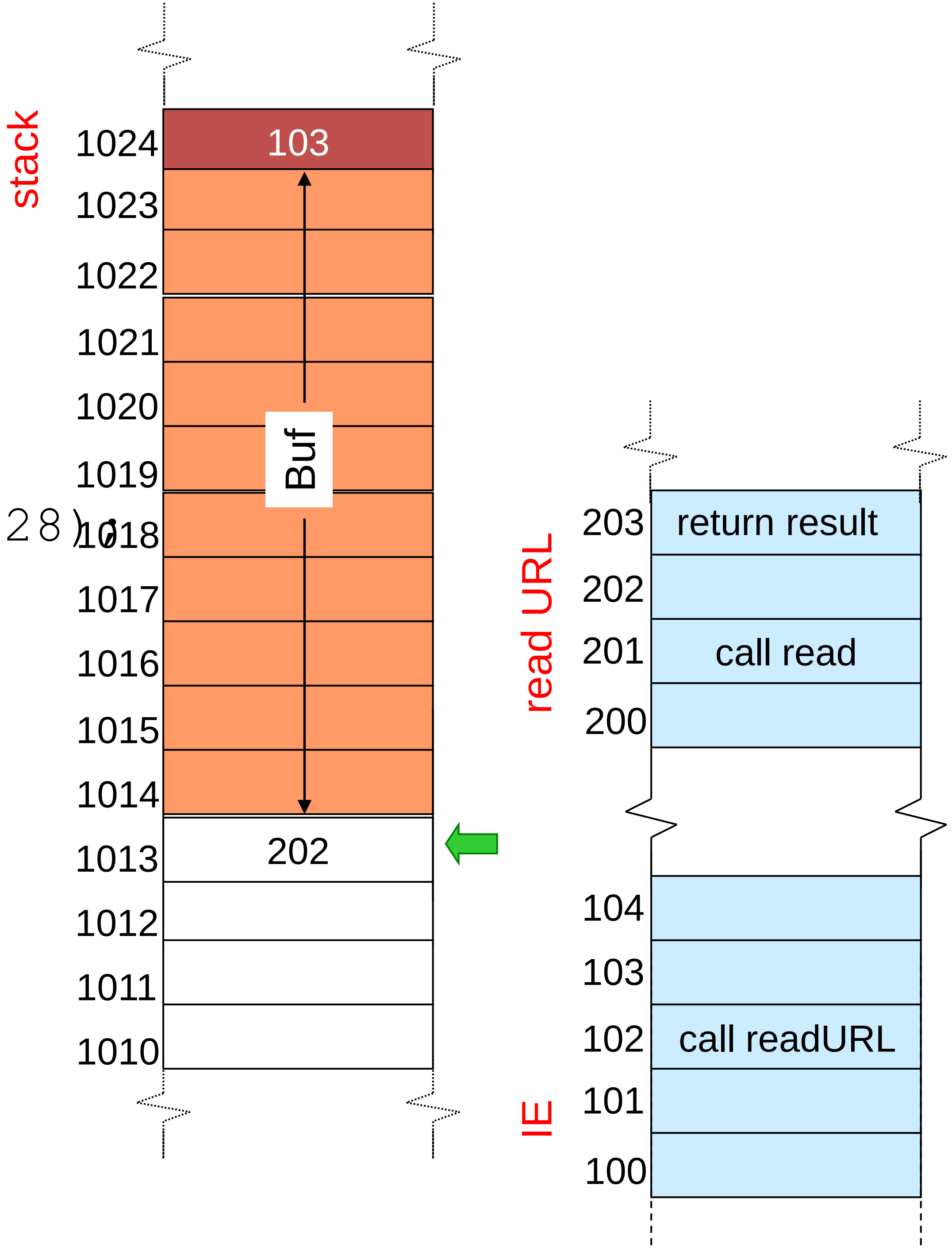
real functions have variables

```
getURL ()
{
   char Buf[10];
   read(keyboard,Buf,128);
   get_webpage (Buf);
}
IE ()
{
   getURL ();
}
```



real functions have variables

```
getURL ()
{
   char Buf[10];
   read(keyboard,Buf,128);
   get_webpage (Buf);
}
IE ()
{
   getURL ();
}
```



stack

1024	103
1023	
1022	
1021	
1020	
1019	
1018	
1017	
1016	
1015	
1014	
1013	202
1012	
1011	
1010	

read URL

203	return result
202	
201	call read
200	

IE

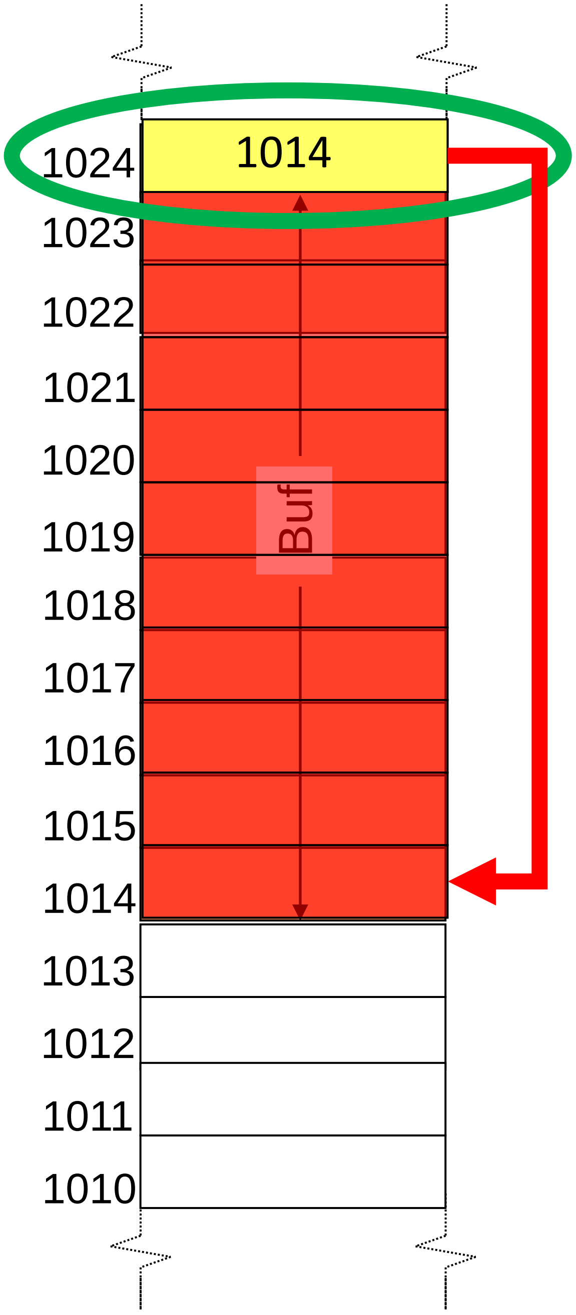
104	
103	
102	call readURL
101	
100	

what is next?

- we have learned a lot
- but where are the vulnerabilities?
- and how do we exploit them?

Exploit

```
getURL ()
{
   char Buf[10];
   read(keyboard,Buf,128);
   get_webpage (Buf);
}
IE ()
{
   getURL ();
}
```



That is it, really

- all we need to do is stick our program in the buffer