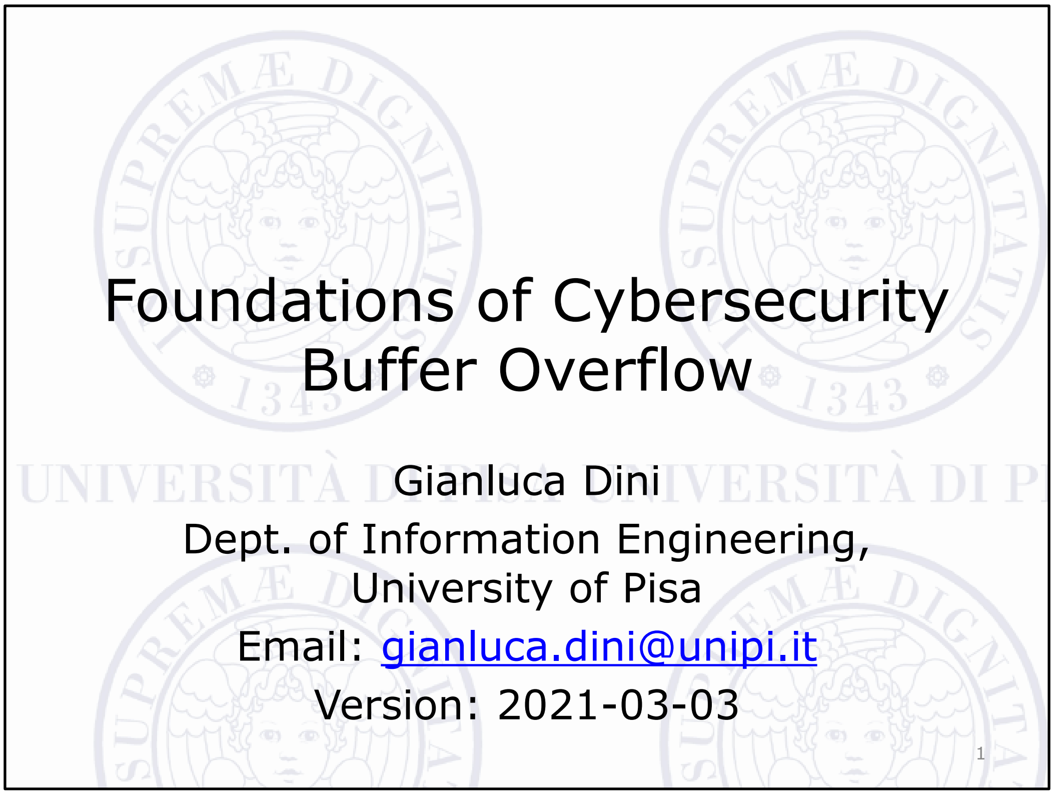
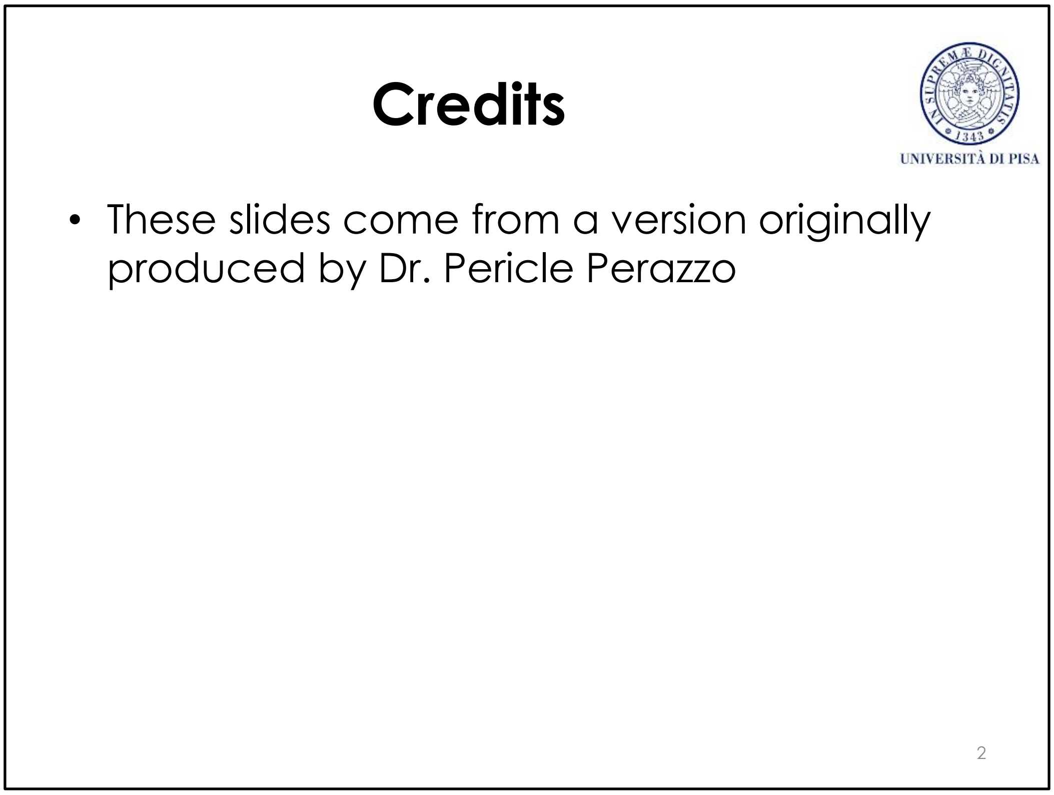


Foundations of Cybersecurity
Buffer Overflow

Gianluca Dini
Dept. of Information Engineering,
University of Pisa
Email: gianluca.dini@unipi.it

Version: 2021-03-03

1

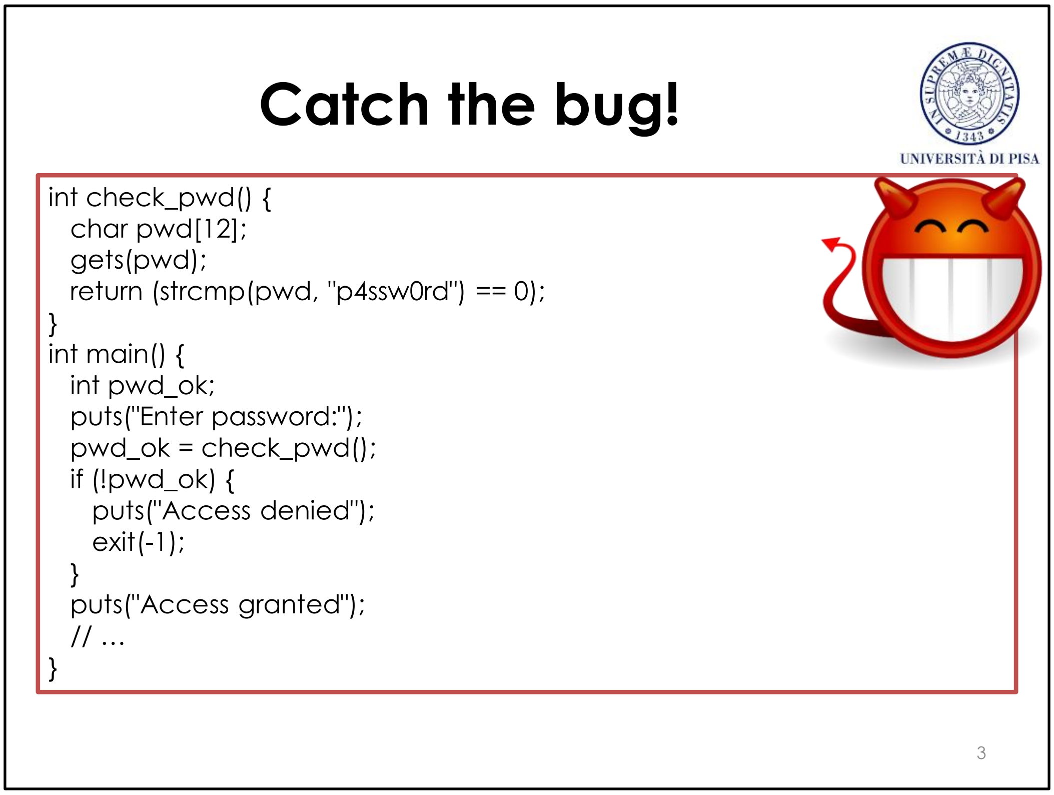


Credits

UNIVERSITÀ DI PISA

- These slides come from a version originally produced by Dr. Pericle Perazzo

2

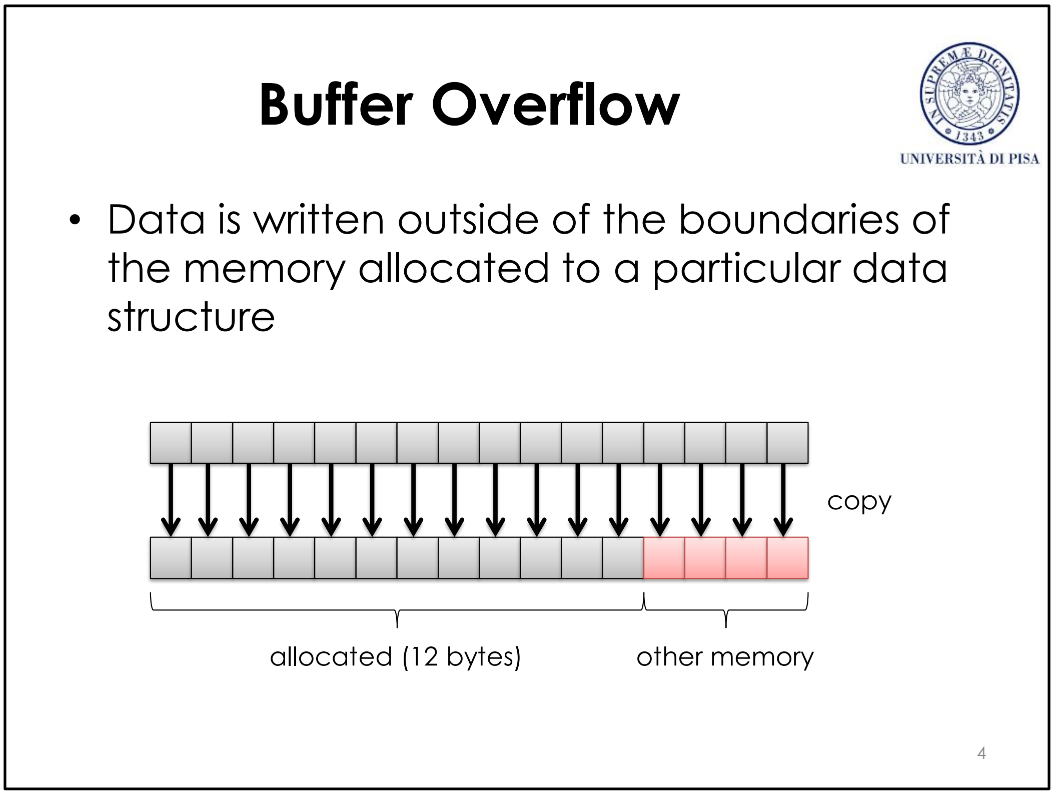


Catch the bug!

```
int check_pwd() {
  char pwd[12];
  gets(pwd);
  return (strcmp(pwd, "p4ssw0rd") == 0);
}
int main() {
  int pwd_ok;
  puts("Enter password:");
  pwd_ok = check_pwd();
  if (!pwd_ok) {
    puts("Access denied");
    exit(-1);
  }
  puts("Access granted");
  // …
}
```

3

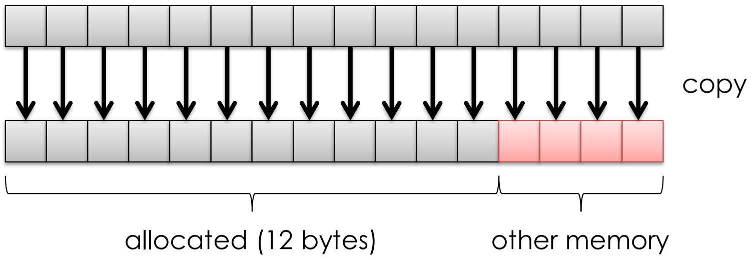
In this slide and the following ones, the red borders mean vulnerable code, whereas the green border means corrected code.
This program asks the user for a password with the function check_pwd() and compares it with the correct password. If the password is wrong, it will print "Access denied" and abort the program, otherwise it will print "Access granted".
If the user inserts a password of 12 characters or more, the gets() function will write beyond the last character of the variable "pwd". This programming anomaly is called *buffer overflow*. What happens in this case is undefined (*undefined behavior*). Some compilers could check for out-of-bound write and raise a hardware exception (which will eventually abort the program). However, in most cases, C/C++ compilers are focused on producing efficient code, so no out-of-bound check will be performed on buffer accesses.



Buffer Overflow

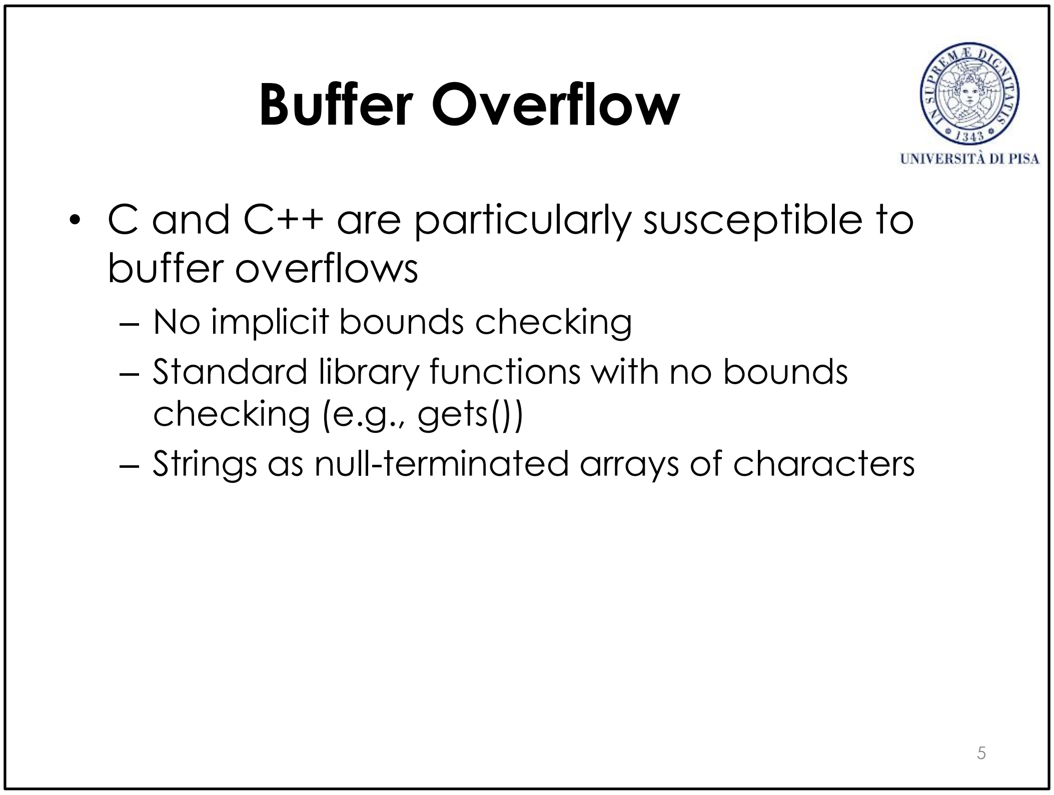
UNIVERSITÀ DI PISA

- Data is written outside of the boundaries of the memory allocated to a particular data structure



4

In this case, the data that overflows the buffer is written in the memory space that happens to be contiguous to the overflowed buffer, containing other data (variables, etc.). Therefore, other data is over-written.

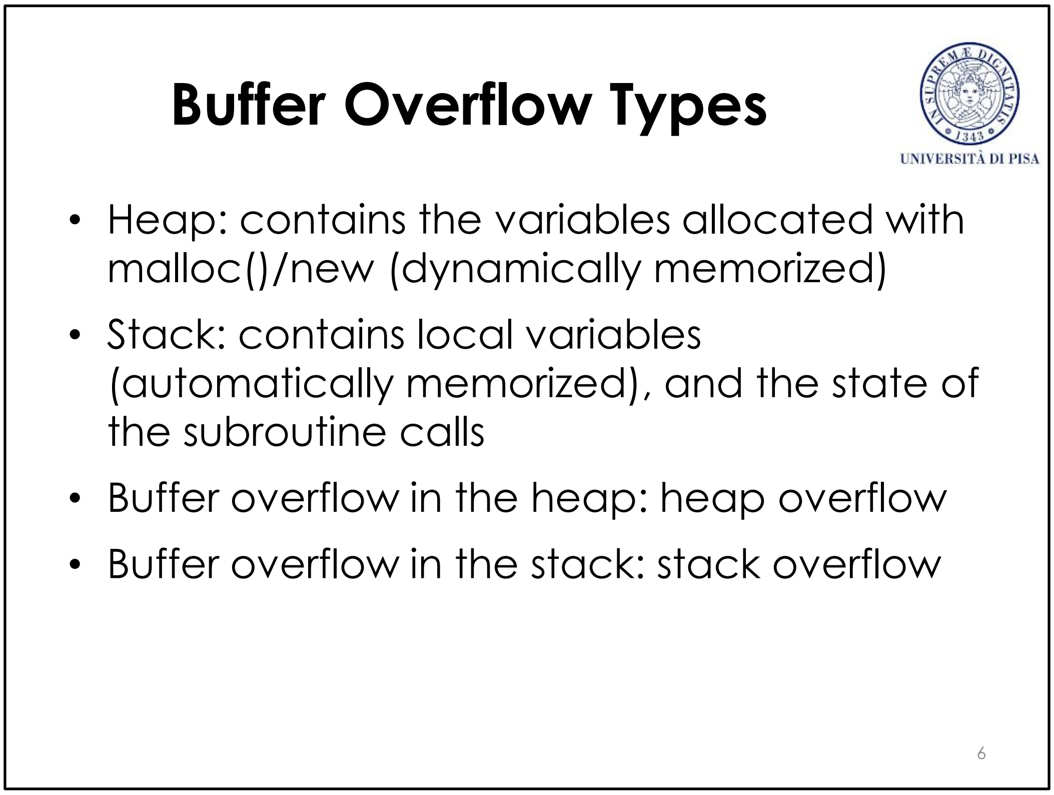


Buffer Overflow

- C and C++ are particularly susceptible to buffer overflows
  - No implicit bounds checking
  - Standard library functions with no bounds checking (e.g., gets())
  - Strings as null-terminated arrays of characters

5

The C and C++ languages are particularly susceptible to bugs leading to buffer overflow vulnerabilities. This is because they do not enforce implicit bounds checking on arrays, and they provide standard library functions (e.g., gets()) which do not enforce bounds checking. Moreover, they implement strings as null-terminated arrays of characters. This makes programming in C/C++ more error-prone than in other languages, for example the Pascal language which implements strings as couples <length, characters>.

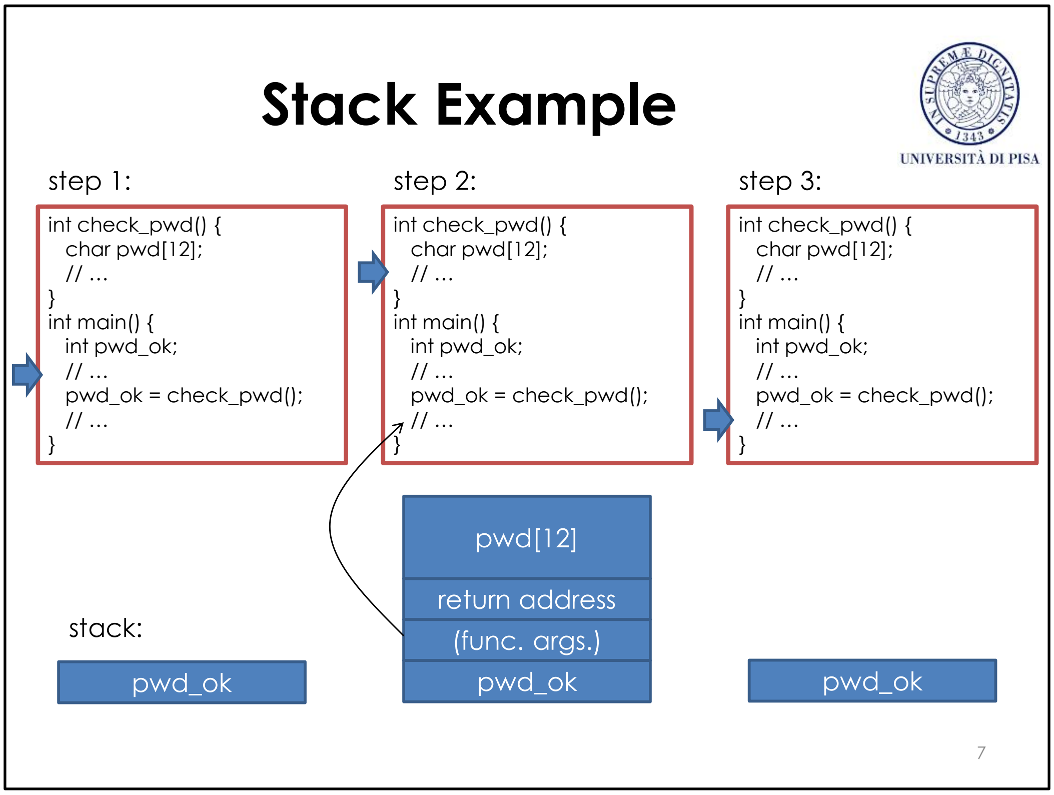


Buffer Overflow Types

- Heap: contains the variables allocated with malloc()/new (dynamically memorized)
- Stack: contains local variables (automatically memorized), and the state of the subroutine calls
- Buffer overflow in the heap: heap overflow
- Buffer overflow in the stack: stack overflow

6

A non-static variable in C/C++ can be allocated either dynamically (with malloc() or new/new[] operators) o automatically (by declaring it as a local variable in a function). The *heap* is a memory space that contains all the dynamic variables, while the *stack* is a memory space that contains all the automatic variables and the parameters and, most importantly, the return addresses of the subroutines. A buffer overflow in the heap is called heap overflow, while a *buffer overflow* in the stack is called *stack overflow*. Both heap and stack overflows can read/change the value of other variables. Stack overflow is generally more dangerous, because it can directly change the execution sequence by changing the return address of the subroutines.



Stack Example

step 1:

```
int check_pwd() {
  char pwd[12];
  // …
}
int main() {
  int pwd_ok;
  // …
  pwd_ok = check_pwd();
  // …
}
```

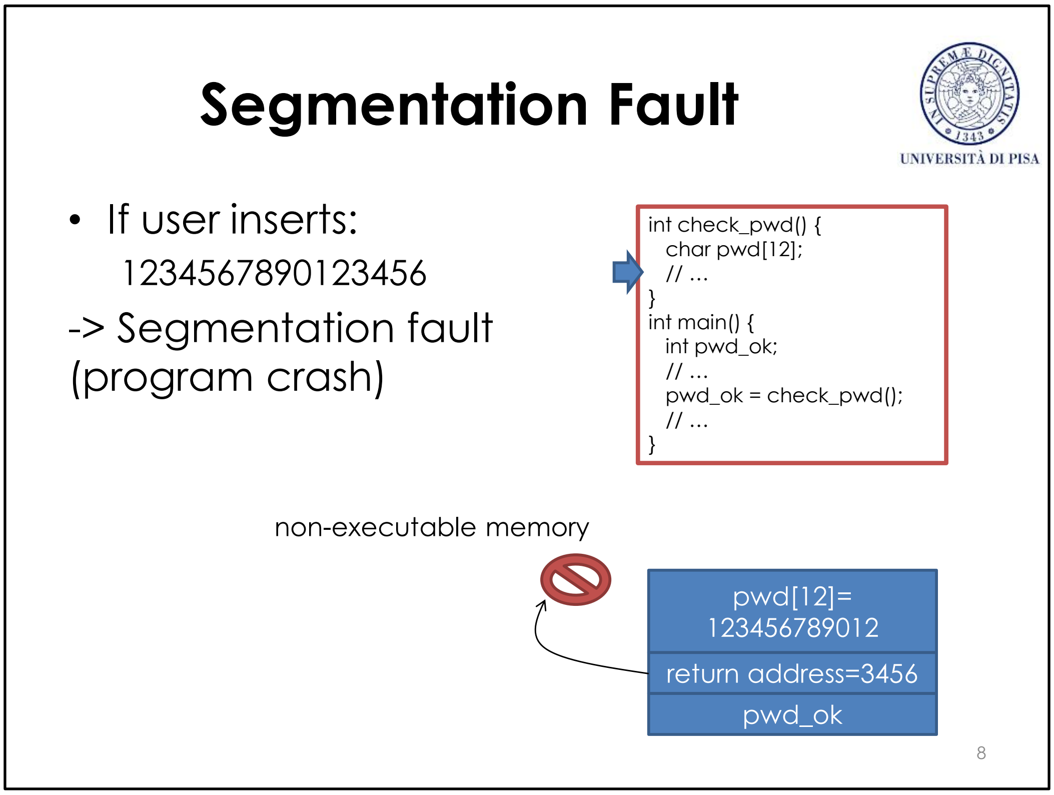
step 2:

```
int check_pwd() {
  char pwd[12];
  // …
}
int main() {
  int pwd_ok;
  // …
  pwd_ok = check_pwd();
  // …
}
```

step 3:

```
int check_pwd() {
  char pwd[12];
  // …
}
int main() {
  int pwd_ok;
  // …
  pwd_ok = check_pwd();
  // …
}
```

The figure shows what happens to the stack when the check_pwd() function is called by the main() function. The blue arrow indicates the current instruction pointer. The stack grows upward and shrinks downward. The function call stores in the stack the possible argument values (in this case: none) and the return address, which is the address to the instruction just after the function calling. Then, the function stores in the stack its possible local variables (in this case: “pwd”). When the function check_pwd() returns, the local variables, the return address, and the possible argument values are removed from the stack, and the execution point starts again from the return address.



Segmentation Fault

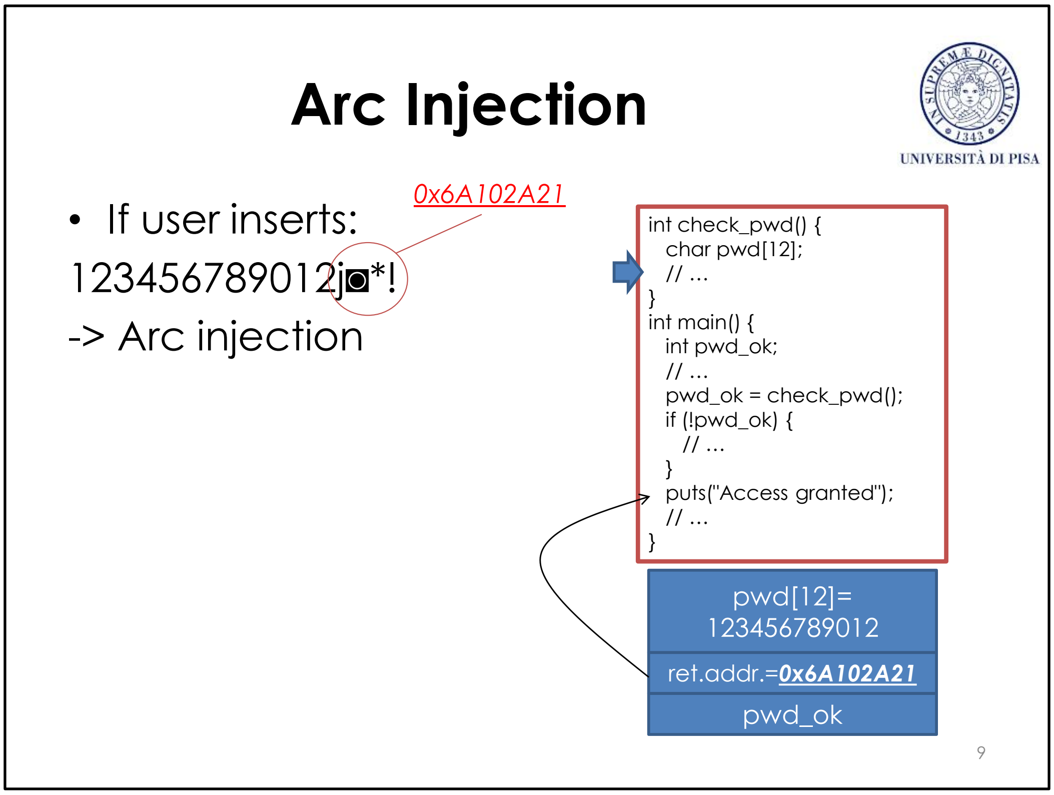
- If user inserts:
  1234567890123456

-> Segmentation fault (program crash)

```
int check_pwd() {
   char pwd[12];
   // ...
}
int main() {
   int pwd_ok;
   // ...
   pwd_ok = check_pwd();
   // ...
}
```

non-executable memory
pwd[12]= 123456789012
return address=3456
pwd_ok
8

If the user inserts a password with a length of 12 characters or more, then the return address will be over-written like in the above example figure. In this case, the overflowed data is interpreted as a return address. When the function check_pwd() returns, the instruction pointer tries to jump to such an address. Let us suppose that the address points to a non-executable part of the memory. This will result in a *segmentation fault* and an abnormal program termination.
In the above example, we assumed a calling convention that do not pushes the *base pointer* (EBP) into the stack for the ease of exposition. In other calling conventions, the EBP is additionally pushed in the stack when a function is called. The EBP usually stands between the return address and the local variables. In this case, a buffer overflow in the "pwd" variable should also overwrite the EBP before overwriting the return address.



Arc Injection

- If user inserts:

123456789012j◙*!

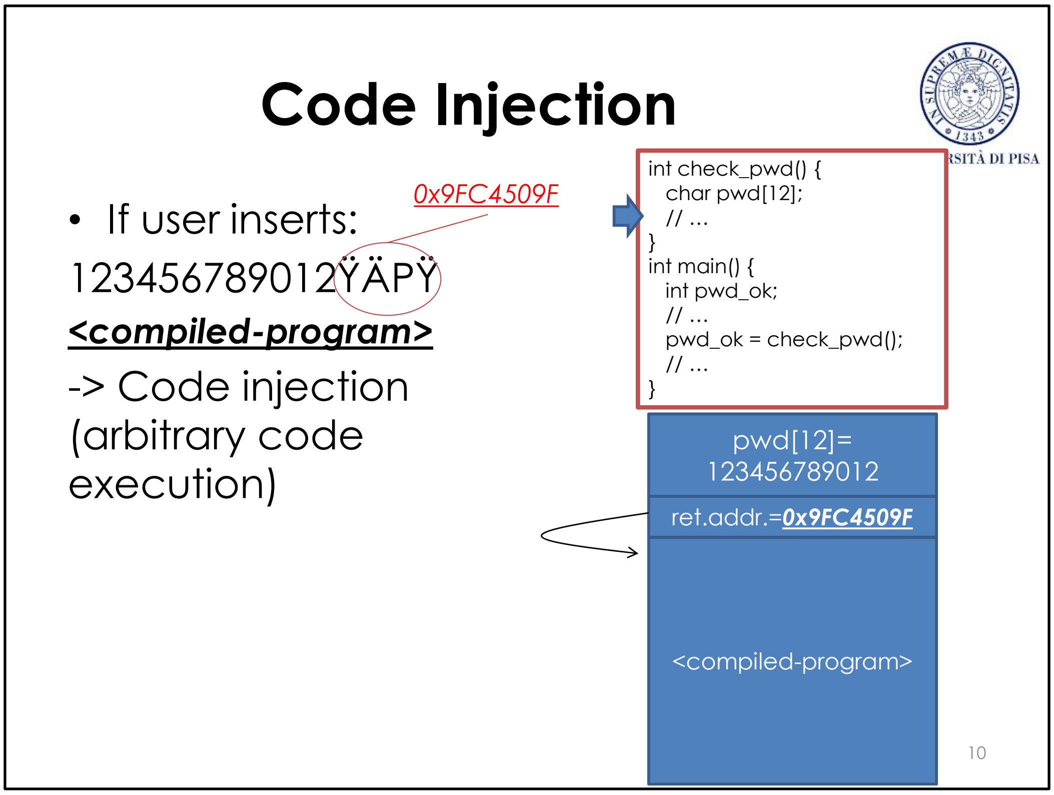
-> Arc injection

0x6A102A21

```
int check_pwd() {
   char pwd[12];
   // ...
}
int main() {
   int pwd_ok;
   // ...
   pwd_ok = check_pwd();
   if (!pwd_ok) {
      // ...
   }
   puts("Access granted");
   // ...
}
```

pwd[12]= 123456789012
ret.addr.=0x6A102A21
pwd_ok
9

Let us suppose that the user inserts the above password. The ASCII characters j◙*! correspond to the address 0x6A102A21 (in hexadecimal digits). Such an address is valid and points to the puts("Access granted") instruction of the main() function. When check_pwd() returns, the execution flow will jump to the instructions implementing the access-grant branch, so skipping the actual password check. Such an attack is commonly known as *arc injection*, because the attacker injects a malicious «arc» in the program execution flow, from the check_pwd()'s return instruction to the puts("Access granted") instruction.



Code Injection

- If user inserts:

123456789012ŸÄPŸ

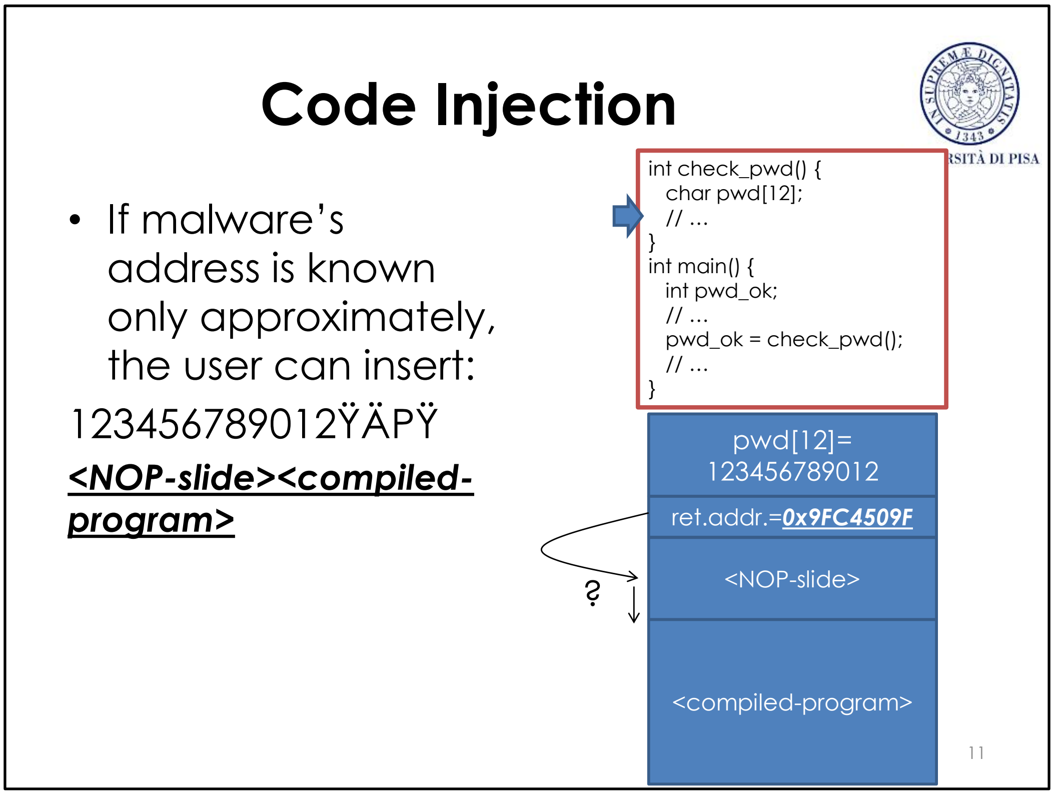
<compiled-program>

-> Code injection (arbitrary code execution)

```
int check_pwd() {
   char pwd[12];
   // ...
}
int main() {
   int pwd_ok;
   // ...
   pwd_ok = check_pwd();
   // ...
}
```

0x9FC4509F
pwd[12]= 123456789012
ret.addr.=0x9FC4509F
<compiled-program>
10

Let us suppose that the user inserts the above password. “compiled-program” represents the binary instructions of a malicious software (*malware*). The ASCII characters ŸÄPŸ correspond to the address 0x9FC4509F in hexadecimal digits. Such an address is valid and points to the first memory location of the injected malware. (Let us suppose by now that the part of the memory which contains the stack is executable.) When check_pwd() returns, the execution flow will jump to the malware instructions. Such an attack is commonly known as *code injection*, and it results in an *arbitrary code execution*. Note that, in the above example, the malware cannot contain '\n' characters. Otherwise, the gets() function will stop accepting input before the malware is completely injected.



Code Injection

- If malware's address is known only approximately, the user can insert:

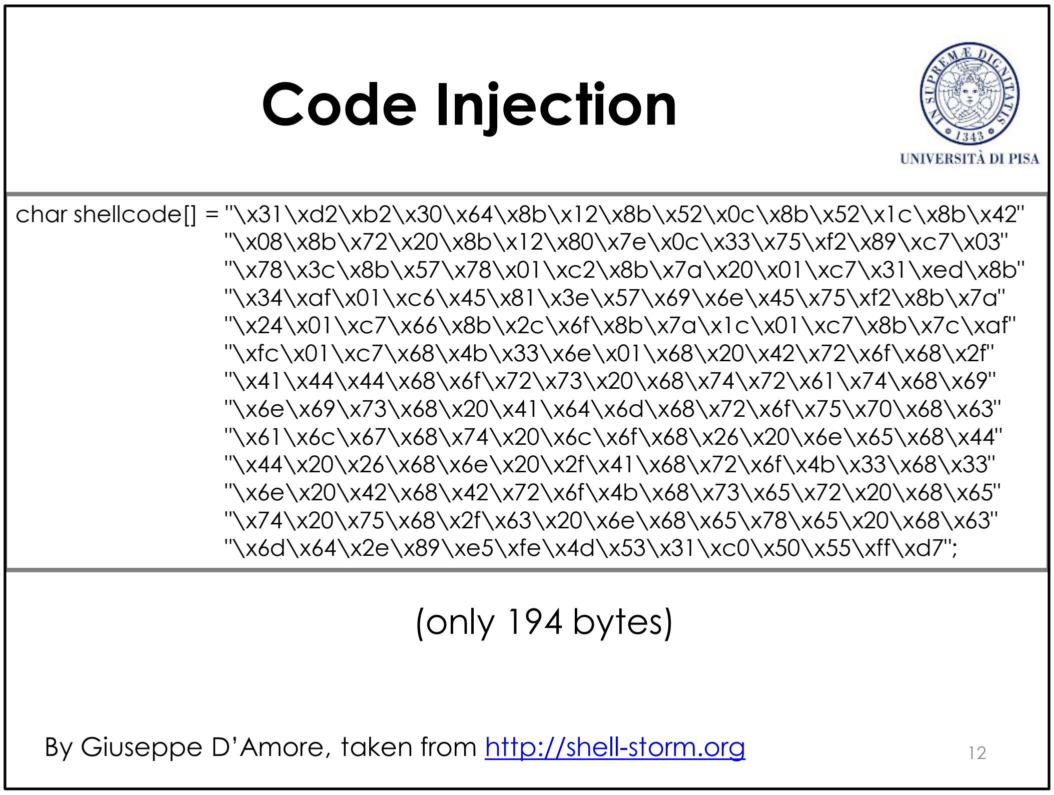
123456789012ŸÄPŸ

<u><NOP-slide><compiled-program></u>

```
int check_pwd() {
   char pwd[12];
   // ...
}
int main() {
   int pwd_ok;
   // ...
   pwd_ok = check_pwd();
   // ...
}
```

pwd[12]= 123456789012
ret.addr.=0x9FC4509F
<NOP-slide>
?
<compiled-program>
11

To mount a code injection, the attacker must know the exact address of the stack, which is not always predictable. To overcome this, the attacker can prepend the malware with many NOP instructions (*NOP slide*). The overwritten return address needs only to jump in the middle of the NOP slide, so it can be approximated. Then, the control flow will drop down to the malware code.



Code Injection

UNIVERSITÀ DI PISA

```
char shellcode[] = "\x31\xd2\xb2\x30\x64\x8b\x12\x8b\x52\x0c\x8b\x52\x1c\x8b\x42"
                   "\x08\x8b\x72\x20\x8b\x12\x80\x7e\x0c\x33\x75\xf2\x89\xc7\x03"
                   "\x78\x3c\x8b\x57\x78\x01\xc2\x8b\x7a\x20\x01\xc7\x31\xed\x8b"
                   "\x34\xaf\x01\xc6\x45\x81\x3e\x57\x69\x6e\x45\x75\xf2\x8b\x7a"
                   "\x24\x01\xc7\x66\x8b\x2c\x6f\x8b\x7a\x1c\x01\xc7\x8b\x7c\xaf"
                   "\xfc\x01\xc7\x68\x4b\x33\x6e\x01\x68\x20\x42\x72\x6f\x68\x2f"
                   "\x41\x44\x44\x68\x6f\x72\x73\x20\x68\x74\x72\x61\x74\x68\x69"
                   "\x6e\x69\x73\x68\x20\x41\x64\x6d\x68\x72\x6f\x75\x70\x68\x63"
                   "\x61\x6c\x67\x68\x74\x20\x6c\x6f\x68\x26\x20\x6e\x65\x68\x44"
                   "\x44\x20\x26\x68\x6e\x20\x2f\x41\x68\x72\x6f\x4b\x33\x68\x33"
                   "\x6e\x20\x42\x68\x42\x72\x6f\x4b\x68\x73\x65\x72\x20\x68\x65"
                   "\x74\x20\x75\x68\x2f\x63\x20\x6e\x68\x65\x78\x65\x20\x68\x63"
                   "\x6d\x64\x2e\x89\xe5\xfe\x4d\x53\x31\xc0\x50\x55\xff\xd7";
```

(only 194 bytes)

By Giuseppe D’Amore, taken from http://shell-storm.org

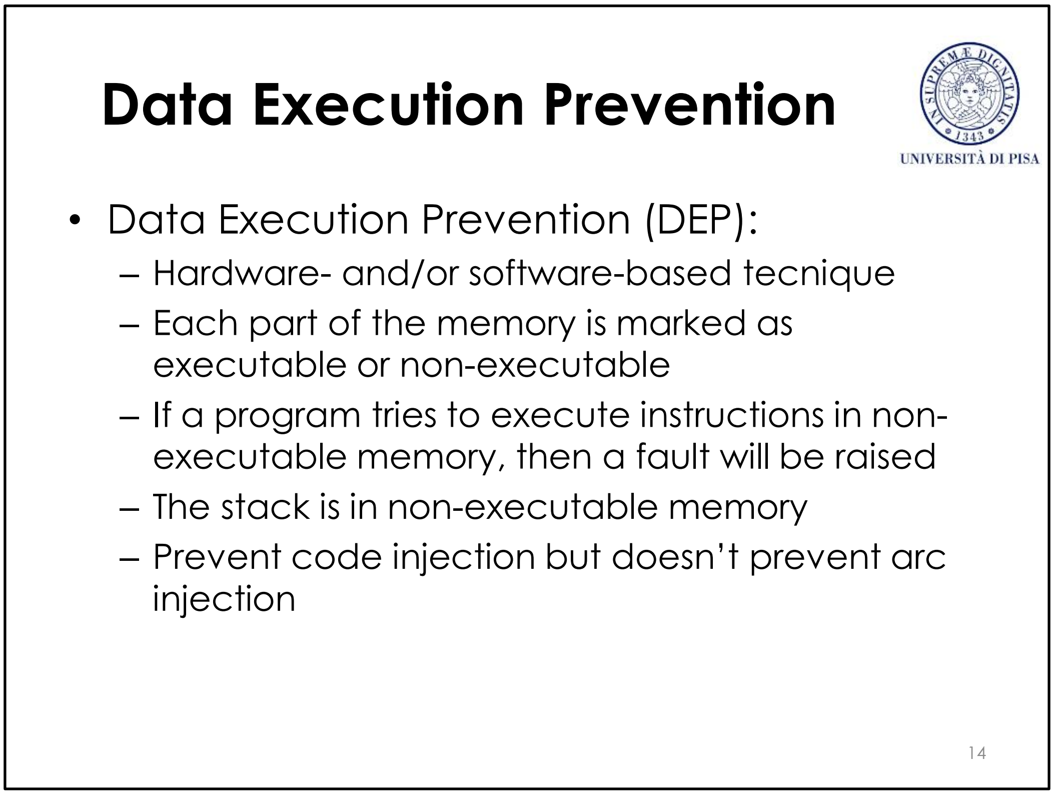
12

The injected code can be a *shellcode*, i.e., a code implementing a (local or remote) command shell by which the adversary can execute arbitrary commands on the victim machine. This shellcode example (taken from shell-storm.org) is only 194-byte-long and is suitable for all versions of Windows platforms. Another possibility is to inject a *download-and-execute* code, which downloads in memory a larger malware from a given Internet location and then executes it. Download-and-execute codes permit the attacker to execute complex malware even with a short buffer overflow.

Buffer Overflow

COUNTERMEASURES

13

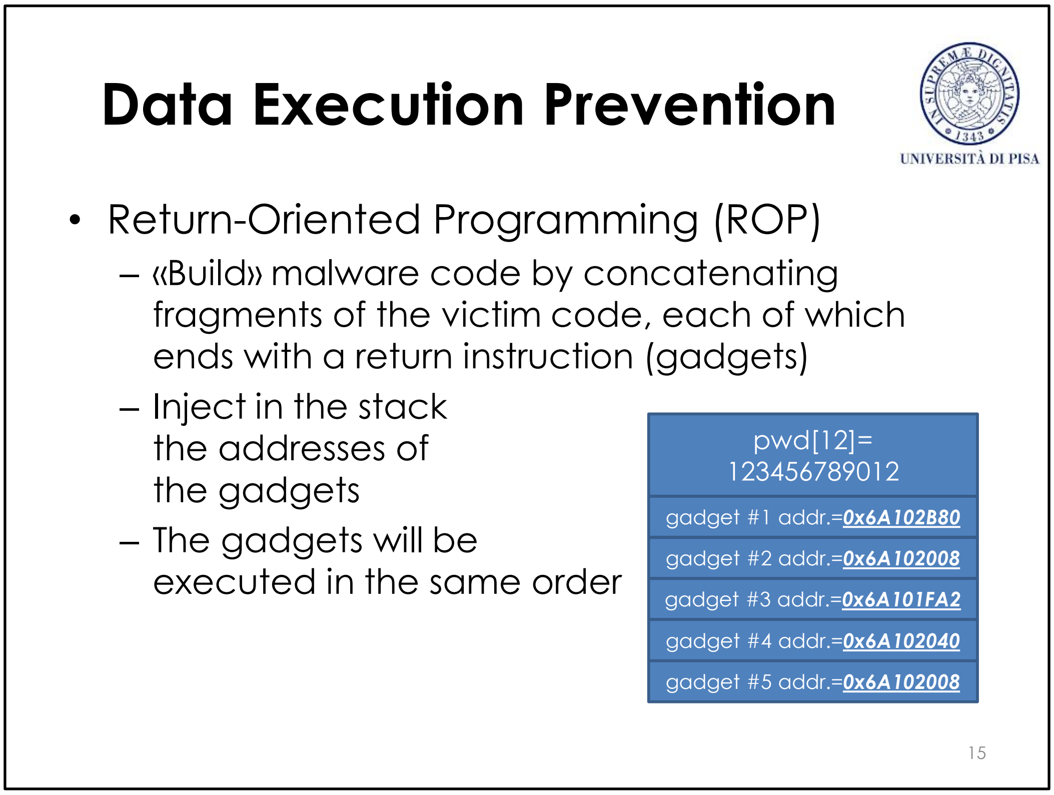


Data Execution Prevention

- Data Execution Prevention (DEP):
  - Hardware- and/or software-based tecnique
  - Each part of the memory is marked as executable or non-executable
  - If a program tries to execute instructions in non-executable memory, then a fault will be raised
  - The stack is in non-executable memory
  - Prevent code injection but doesn't prevent arc injection

14

Starting from 2000's, the major operating systems and compilers adopted a series of stack overflow protections, the most common of which are *Data Execution Prevention* (DEP), *Address Space Layout Randomization* (ASLR), and *Stack Canaries*. None of these protections is definitive, because they do not make the attack impossible, but only more technically challenging.
Data Execution Prevention works by marking each part of the memory as executable or non-executable. The stack is marked as non-executable, so when the victim process tries to execute the malware in the stack, a fault is raised, and the process abnormally terminates. Hardware-based DEP has a negligible impact on performance. Note that DEP prevents code injection, but it does not prevent arc injection.



Data Execution Prevention

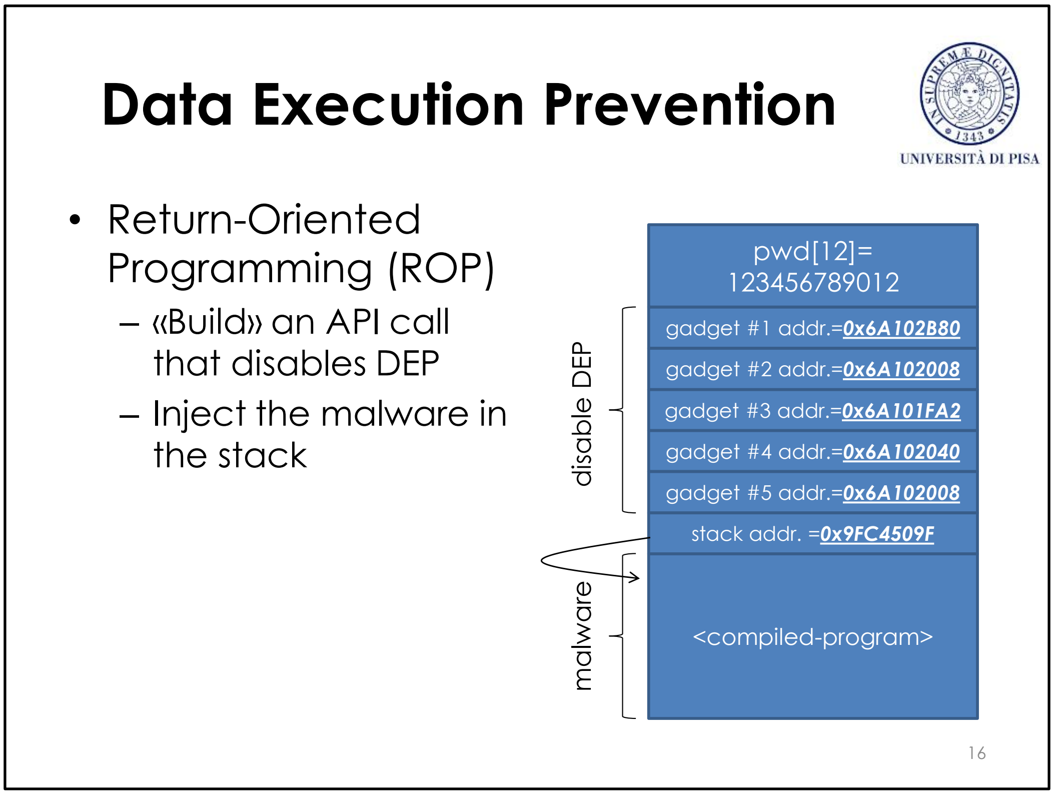
UNIVERSITÀ DI PISA

- Return-Oriented Programming (ROP)
  - «Build» malware code by concatenating fragments of the victim code, each of which ends with a return instruction (gadgets)
  - Inject in the stack the addresses of the gadgets
  - The gadgets will be executed in the same order

pwd[12]= 123456789012
gadget #1 addr.=**0x6A102B80**
gadget #2 addr.=**0x6A102008**
gadget #3 addr.=**0x6A101FA2**
gadget #4 addr.=**0x6A102040**
gadget #5 addr.=**0x6A102008**

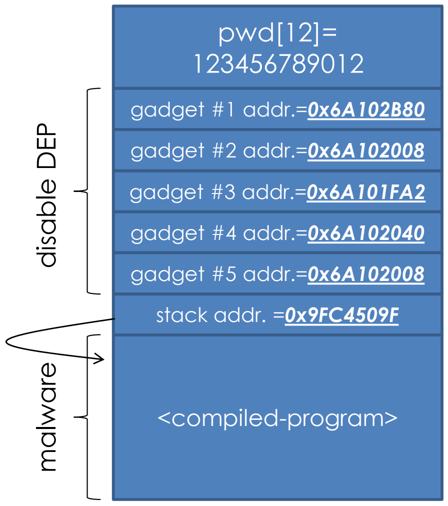
15

DEP can be cheated with *Return-Oriented Programming* (ROP) techniques. In ROP, malware is not injected but «built» with a sequence of victim code fragments, each of which ends with a return instruction (*gadgets*). The attacker injects the addresses of such gadgets in the stack. At the end of every gadget, the return instruction will make the instruction pointer jump to the next gadget. All the gadgets are located in executable memory, thus DEP is bypassed and the victim process does not abnormally terminate.



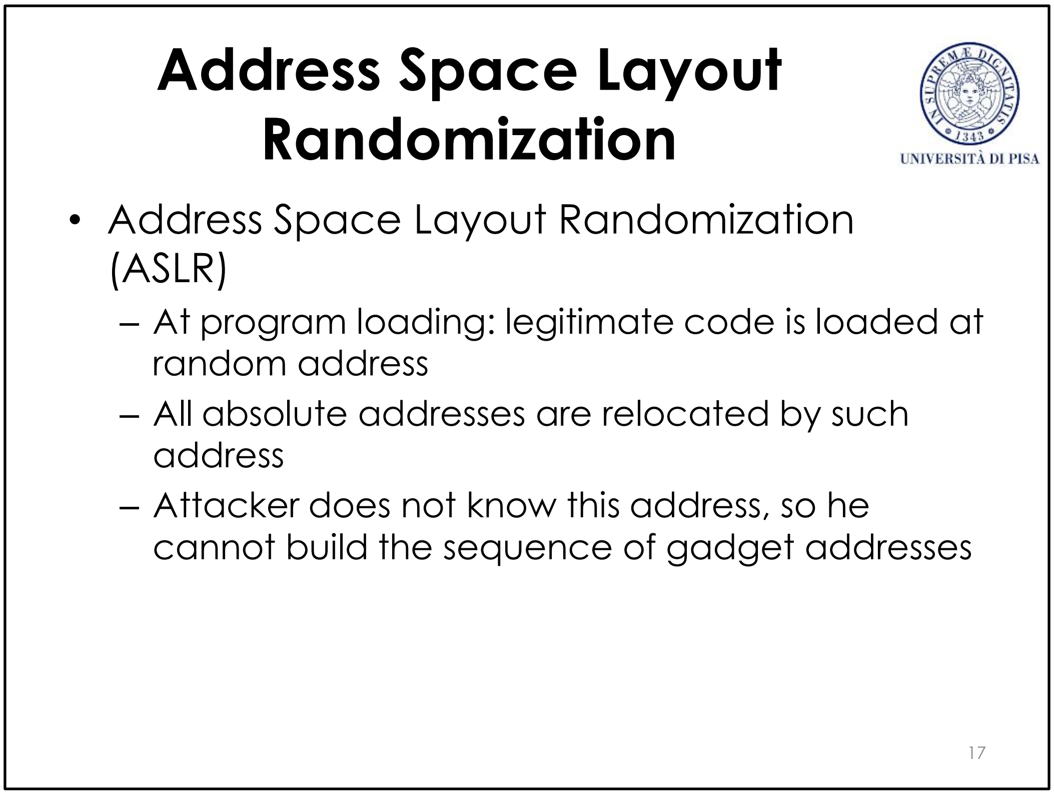
Data Execution Prevention

- Return-Oriented Programming (ROP)
  - «Build» an API call that disables DEP
  - Inject the malware in the stack


pwd[12]= 123456789012
gadget #1 addr.=0x6A102B80
gadget #2 addr.=0x6A102008
gadget #3 addr.=0x6A101FA2
gadget #4 addr.=0x6A102040
gadget #5 addr.=0x6A102008
stack addr. =0x9FC4509F
<compiled-program>
disable DEP
malware

16

DEP can be cheated with *Return-Oriented Programming* (ROP) techniques. In ROP, malware is not injected but «built» with a sequence of victim code fragments, each of which ends with a return instruction (*gadgets*). The attacker injects the addresses of such gadgets in the stack. At the end of every gadget, the return instruction will make the instruction pointer jump to the next gadget. All the gadgets are located in executable memory, thus DEP is bypassed and the victim process does not abnormally terminate.
Depending on the victim code, it could not be possible to build complex malware (e.g., a shellcode) with gadgets. A simpler technique is to use the gadgets to build an API call that disables DEP on the victim process, which is much simpler. After that, the attacker can mount a «classic» code injection, by placing after all the gadgets' addresses an additional address pointing to a location of the stack containing a complex malware.

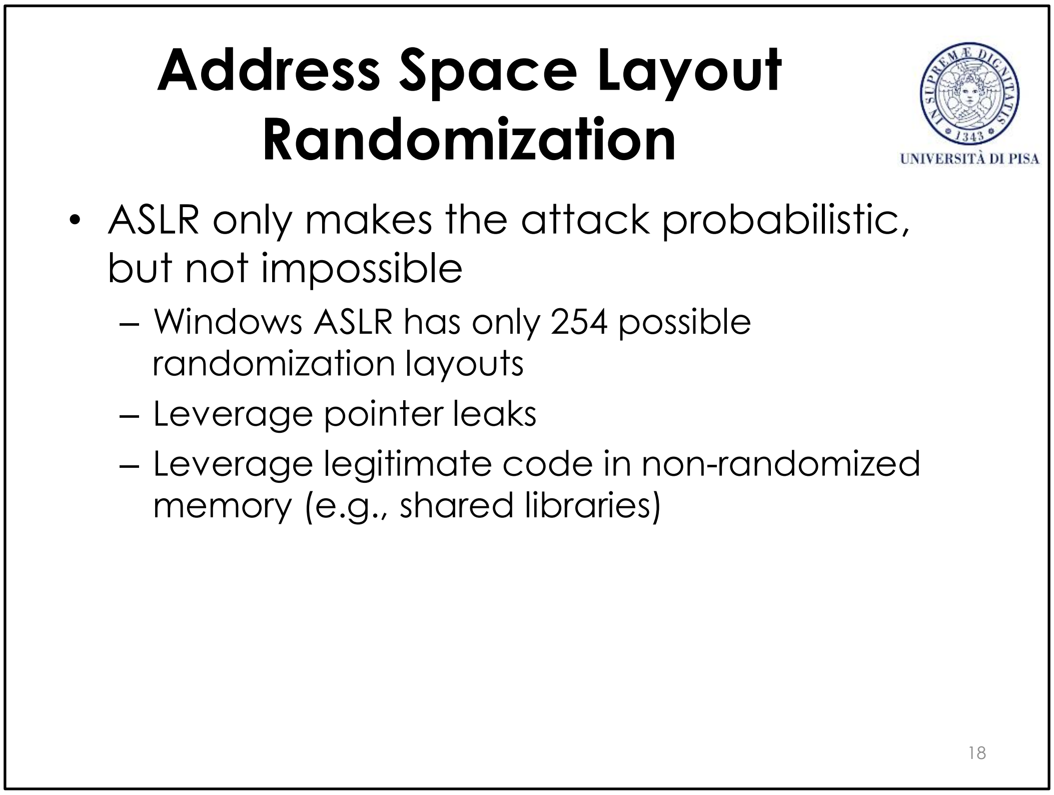


Address Space Layout Randomization

- Address Space Layout Randomization (ASLR)
  - At program loading: legitimate code is loaded at random address
  - All absolute addresses are relocated by such address
  - Attacker does not know this address, so he cannot build the sequence of gadget addresses

17

In old operating systems, when a program was executed, it was always loaded in memory starting from address 0. This makes ROP easy, since the adversary knows deterministically all the addresses of the victim code. With *Address Space Layout Randomization* (ASLR), the operating system decides randomically where to load the program to be executed. All the absolute addresses of the program code are relocated by the same random offset. Since the attacker does not know the relocation address, he cannot perform ROP because he cannot build the sequence of gadget addresses.

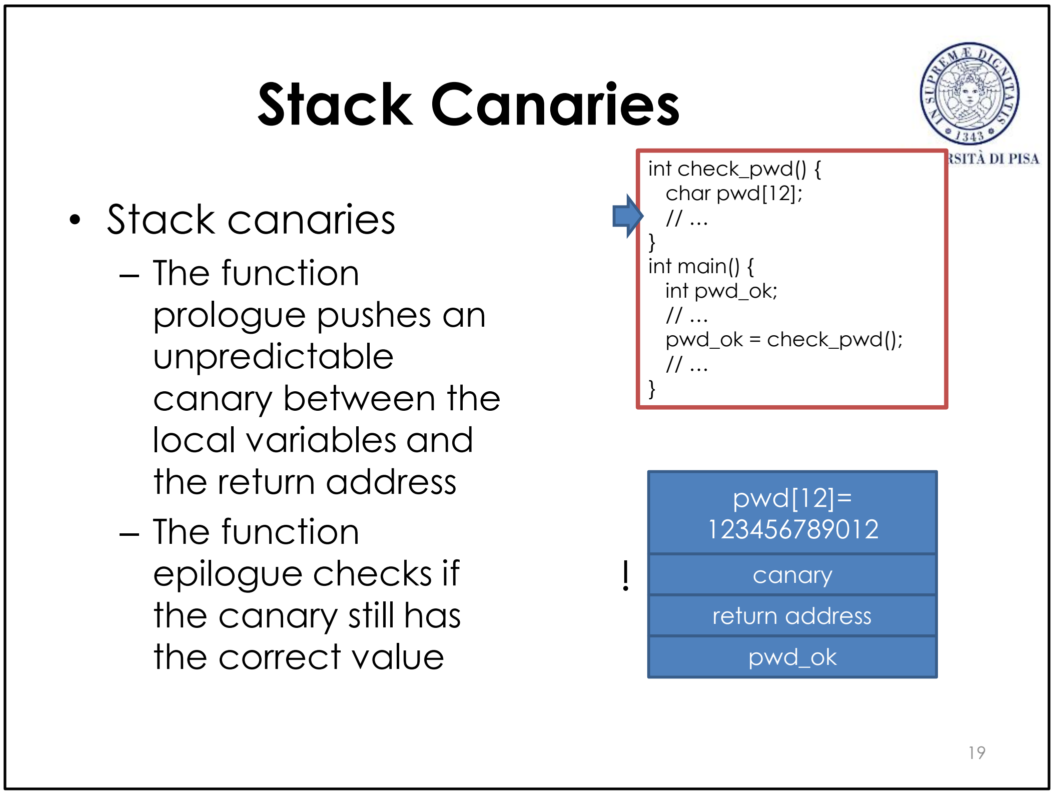


Address Space Layout Randomization

- ASLR only makes the attack probabilistic, but not impossible
  - Windows ASLR has only 254 possible randomization layouts
  - Leverage pointer leaks
  - Leverage legitimate code in non-randomized memory (e.g., shared libraries)

18

ASLR is an additional source of uncertainty for the attacker. However, it does not make the attack impossible, rather probabilistic. Indeed, the possible relocation addresses are few in the typical operating system. For example, Windows has only 254 possible random relocation addresses. The attacker could simply try several times to attack the process, until he guesses the correct relocation address. Moreover, the attacker can leverage *pointer leaks*, that are program bugs that expose the value of pointers. If such pointers point to the code (e.g., a function pointer or the return address of a function call), they reveal the relocation address. Finally, the attacker could do ROP with gadgets from non-randomized memory, for example memory that contains the code of libraries shared between different processes.



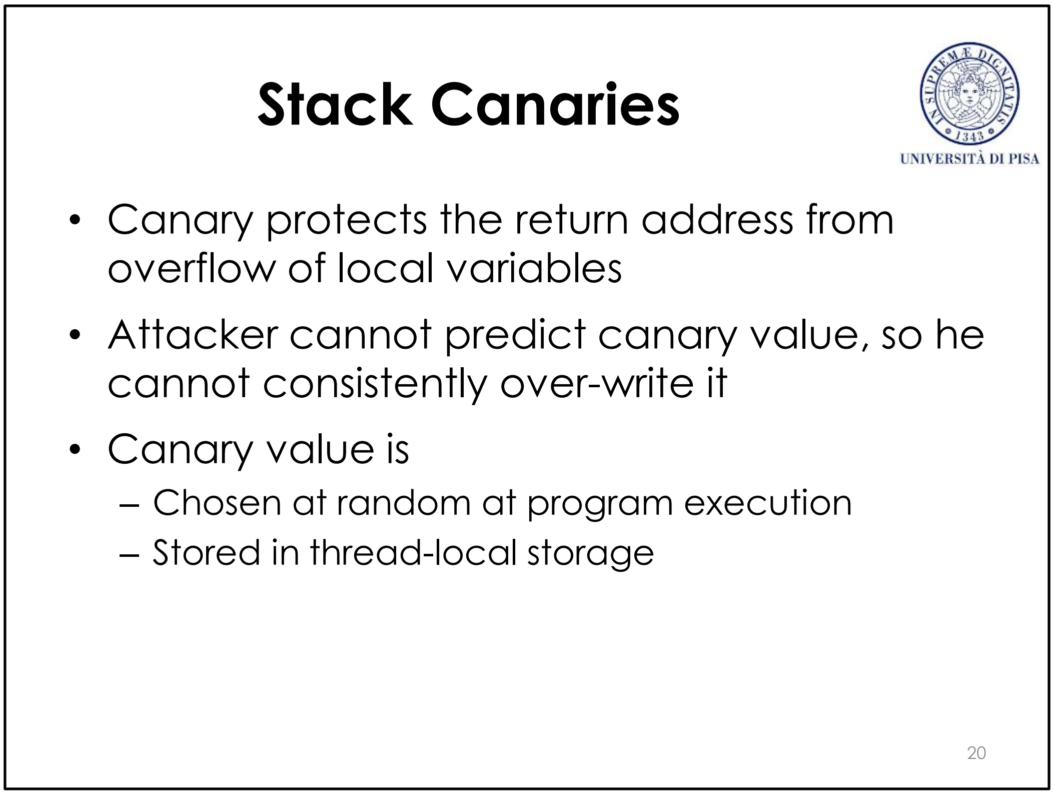
Stack Canaries

- Stack canaries
  - The function prologue pushes an unpredictable canary between the local variables and the return address
  - The function epilogue checks if the canary still has the correct value

```
int check_pwd() {
   char pwd[12];
   // ...
}
int main() {
   int pwd_ok;
   // ...
   pwd_ok = check_pwd();
   // ...
}
```

pwd[12]=
123456789012
!
canary
return address
pwd_ok
19

In the *stack canaries* technique, the prologue of every function pushes a value (*canary*) in the stack between the local variables and the return address. If an attacker wants to overwrite the return address with a buffer overflow, then he must corrupt also the canary. The function epilogue checks if the canary still has the correct value, and it raises an error if it has not.

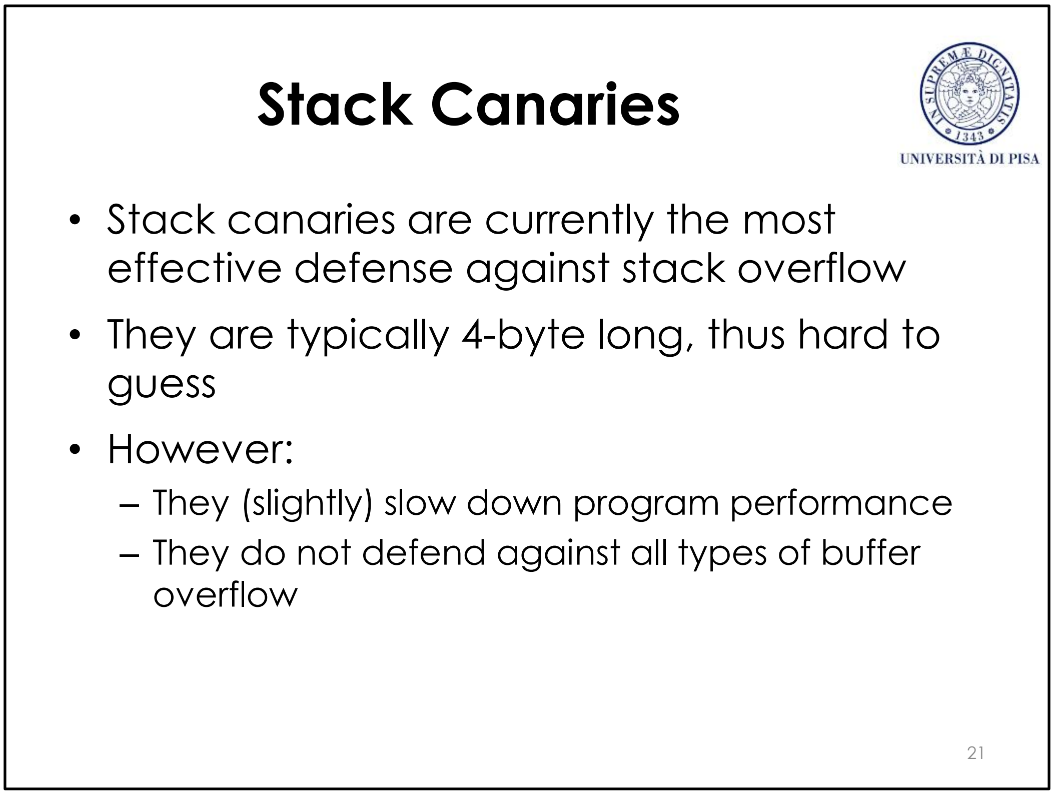


Stack Canaries

- Canary protects the return address from overflow of local variables
- Attacker cannot predict canary value, so he cannot consistently over-write it
- Canary value is
  - Chosen at random at program execution
  - Stored in thread-local storage

20

The canary value must be unpredictable by the attacker. Usually, it is chosen at random at program execution, and stored in the thread-local storage. Stack canaries are a compiler-based protection technique.



Stack Canaries

- Stack canaries are currently the most effective defense against stack overflow
- They are typically 4-byte long, thus hard to guess
- However:
  - They (slightly) slow down program performance
  - They do not defend against all types of buffer overflow

21

Stack canaries are currently the most effective defense against stack overflow, because they are hard to bypass or to guess. Stack canaries are typically 4-byte long, so the attacker has a chance over 2^{32} to guess it. Stack canaries slightly slow down the program performance since they introduce some operations at the prologue and at the epilogue of every function. Some compilers optimize the program by avoiding stack canaries in those function that they consider «safe» from stack overflow vulnerabilities. For example, a compiler could consider safe a function which does not declare local arrays, or declares small local arrays.
Although stack canaries are quite effective and efficient, note however that they can defend only against arc injection and code injection. They cannot defend against those types of stack overflows that do not overwrite the return address. For example, an attacker can use a stack overflow to overwrite variables that are stored in locations contiguous to the overflowed buffer. In this way, the attacker can cause an unexpected behavior of the program without corrupting the return address and the canary. In addition, it is still possible to use a stack overflow to simply produce an abnormal program termination, thus causing a denial of service. Moreover, stack canaries cannot prevent *buffer over-reads*, that is overrunning a buffer's boundary in a *read* operation. A buffer over-read can lead to information leaks. Notably, a buffer over-read in the stack can also leak the value of the stack canary, which could then be used to mount successive stack overflow attacks.