Project 1 - Part 1: Overflowing the Stack

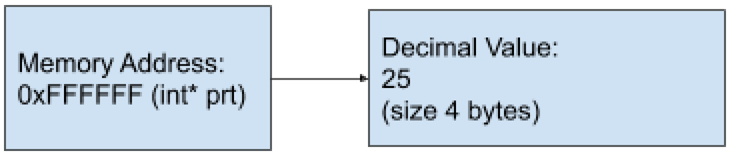
Elsa Rodriguez Plaza

May 20, 2020

✓ Understanding Foundational Concepts

- What is a pointer?

  A pointer is a variable that stores a memory address. For instance, an integer pointer holds the address of a variable that stores a decimal number of 32 bits. The following diagram depicts the concept. [1] [2]



- How is memory managed differently between "C" and "Java"? How is it allocated and how is it deallocated?

  In languages such as C, memory allocation and deallocation is the programmer's responsibility. So, if the C programmer calls a function such as malloc() and afterwards calls free(), the memory will be allocated and deallocated. However, in Java all the objects are stored in memory, so whenever the programmer declares an object this will be allocated in memory. In Java the deallocation happens automatically when the Java garbage collector frees up unused memory. [3]

- What is the difference between "pass by value" and "pass by reference"?

  When passing parameters by value, a change in the parameters' values will not reflect after the function finishes executing. However, when a function is called passing parameters by reference, any changes made to the parameters will persist after the function finishes. [4]

- Why would it be smart to use fixed-width integer types?

  It really depends on the use case. For example, in systems that have memory and space limitations, such as embedded systems it would be smart to declare fix-width integers to preserve these scarce resources. Similarly, in the case of applications that transmit data from one machine to another one, it would be better to define more precise data types, since it would increase the application's performance. [6]

- What does it mean when a function or variable is static in the "C" language?

  A static global variable or static function are not visible outside of the C file where they are defined. In the case of a static variable defined in a function, the value of the variable is preserved across function calls, but these variables are not visible outside of the function that defined them. [15]

- What is the difference between "compile time" and "run time"? Give an example of something that may be allocated at compile time, and describe some mechanisms for allocating memory at runtime.

  "Compile time" is the time when the code is converted to executable [16], while "run time" is the time when the executable is running [16]. For example, an array is allocated at compile time, whereas calls to malloc(), realloc(), and calloc() in the C language allocate memory at runtime. [17]

- What is the difference between an assembler, compiler, and an interpreter?

  An assembler transforms programs written in Assembly language into machine code, which is the code "understood by the computer", represented by 0s and 1s. A compiler is the language-specific processor that compiles the program (written in a high level programming language) and translates it into machine language. The interpreter, scans single statements looking for errors and terminates if it finds errors. [7]

- Is C a compiled language? Is Python? If either answer is no, how is the language processed so code can be run?

  C is a compiled language and the programs are first translated into machine language and executed by the computer's processor. But, Python is both compiled and interpreted, basically when a Python program is executed, the compiler creates byte code and then the Python virtual machine converts it, so it runs on the chosen platform[8] [18]

- What is the GNU compiler? How do you invoke the GNU Compiler?

  The GNU compiler is a collection of tools to execute and compile C, C++, Objective-C, Ada, Go and D programs. It was originally implemented by Richard Stallman, the founder of the GNU Project and it's currently maintained by the developers on the GNU project. The GBU Compiler is called by executing the following command on a Unix system: **gcc <program-file.c> -o <program_executable>**, **program-file.c** refers to the program file that needs to be compiled and the **program_executable** refers to the executable that will be created when the compilation finishes. [9] [10].

- Explain what GDB is, then explain how you would use it to do the following:

  The GDB is a program debugger that allows to debug programs written in Ada, Assembly, C, C++ and much more. You can set breakpoints, you can inspect the processor's registers, change variables values and a lot more. [11].

  - View the processor registers: By executing the following command: **info registers** [12]
  - Set a breakpoint in code: By executing the following command: **break <line_number, function name or file:line_number>** [13]

  - Find the address of an OS Function: By executing the following command: **info address <Symbol name>** [14]
  - Inspect a memory location: By executing the following command: **x/nfu <address>** where n is how many units to print, f is the format character, u is the unit, expressed in bytes.

✓ Understanding Components

- Address space layout randomization (ASLR)
  - What is ASLR? How does it affect the stack?
    ASLR stands for the Address Space Layout Randomization, it is a technology used for randomizing the stack, heap, shared library address, etc. Randomizing the memory address space layout of a process prevents attackers from knowing the process's stack, heap, and library locations. It affects that stack, since it's address is randomized. [19]
  - How can ASLR be bypassed without turning it off?
    There are other ways to bypass the ASLR without turning it off. One easy, but inefficient way is trial and error (brute force) until the address is found, depending on the degree of the randomization it can take several days. ASLR could also be bypassed by address leaking. Basically, the attacker would get any address from the stack, calculate the difference between the leaked address and a current static address, which would allow him to figure out the offset of anything else in the binary. [20]

- Stack Canary
  - What is a Stack Canary? How does it affect the stack?
    A Stack Canary is a technique used to prevent buffer overflows. Basically, "secrets" (canary values) are placed on the stack and changed every time when the program executes, if these secrets are modified, then the program exits immediately. [19] [21]
  - Are Stack Canaries vulnerable and if so, how?
    It is difficult to bypass Stack Canaries, but not impossible. They can be bypassed by brute forcing and address leaking. There are some vulnerable functions in the C programming language that could leak the Canary values, for example user-controller format strings. [21]

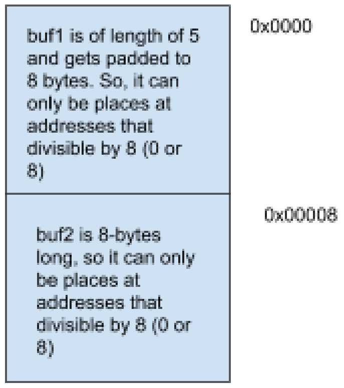
- How does a Stack Buffer Overflow (with no protections enabled) affect the stack? When protection is turned off, an attacker could overwrite the return value of the function on the stack, as well as any parameters' or variables' values.
- Which registers are stack specific and what is the purpose of each (x86 specific for this question) The stack-specific registers are:
    - ESP (stack pointer) points to the top of the stack at all times. The register is increased when something is popped and decreased when something is pushed. [23]
    - EBP (base pointer): Keeps a reference all of the function parameters and the local variables in the current stack frame, some of the literature call it the Frame Pointer [22][24]
- There exists the concept of word-alignment on the stack. Describe what this is and how it affects the stack in a 32-bit OS that utilizes it.

  This means the data that is stored on the stack must be aligned according to the number of bytes it occupies. Stack alignment basically optimizes the operating system, by minimizing the number of extra memory accesses for the processor. [25]

    - Based on this, how would the following lines of code be allocated on the stack?

```
char buf1[5]; // Line 1
char buf2[8]; // Line 2
```

Assuming that that stack is empty, buf1 is 5 bytes long, so it gets padded to 8 bytes. In the case of buf2, it's already 8 bytes-long. Data that is 8 bytes-long will be placed at addresses that are divisible by 8. The diagram below illustrates the concept.


buf1 is of length of 5 and gets padded to 8 bytes. So, it can only be places at addresses that divisible by 8 (0 or 8)
0x0000
buf2 is 8-bytes long, so it can only be places at addresses that divisible by 8 (0 or 8)
0x00008

✓ Compiler Flag Options
  - ○ Research each of the following and give an explanation to how each specifically affects the program/binary. Important: This is referring to the gcc compiler.
    - ■ -O0 (This is the letter O, then the numeral 0): sets the compiler's optimization level. [25]
    - ■ -g: set extra debugging information that in some cases can only be used by the GDB. [26]
    - ■ -fno-stack-protector: disables the stack protection, allowing stack smashing attacks. [27]
    - ■ -z execstack: allows executables stack. [29]

✓ Buffer and Heap Understanding

All of the following questions refers to the address space of a 32 bit Linux distribution OS.

- When is memory allocated and de-allocated on the stack in C?

  When a function is called, the local variables, the parameter, the return address, etcetera are allocated on the stack. This data is deallocated when the function finishes execution. [30]

- How is data stored and organized on the stack?

  The stack grows downwards from high addresses to low addresses and it stores local variables, parameters, and return addresses when functions are called. [31]

- Where in memory is the stack located?

  The stack is located in RAM memory, in between the Kernel code and the spare memory, heap and global data. [31]

- When is memory allocated and de-allocated on the heap in C?

  Data is allocated, when the programmer calls functions that allocate memory, for example malloc(), realloc(), and calloc(). Memory is deallocated when the programmer calls the free() function, which frees up the memory. [31]

- Explain how data is stored and organized on the heap using chunks.

  When a programmer calls a memory allocation function, such as malloc() the memory gets allocated by returning the first block that is large enough to satisfy the memory requirements. This block that gets allocated does not need to be consecutive, so the heap could contain "holes". Similarly, when memory is deallocated, the block (chunk) is freed and the heap will contain the empty chunks. [31][33]

- Where in memory is the heap located?

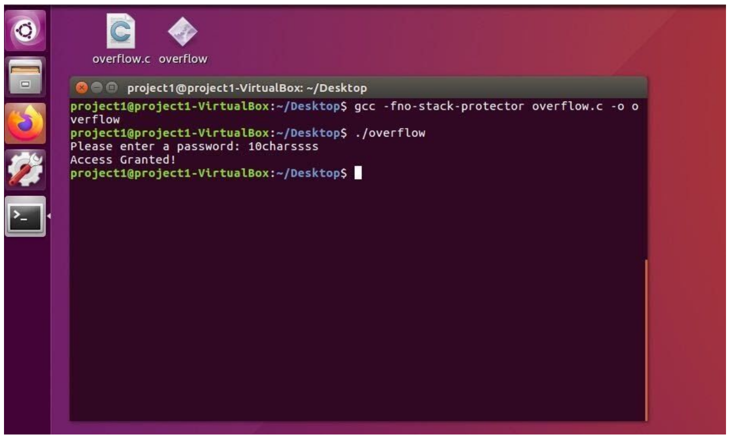
  The heap is located below the Spare memory and above the Global Data (BSS segment) in RAM memory. [31]

- How is the stack used to control program flow when a function is called?

  The stack stores the data associated with the procedure or function calls. Function executions have their own call stacks. Basically, a Stack Frame is created for each function that gets executed. The stack frame contains the local variables, the return function's return address and parameters. The way that the stack controls the execution flow is as follows: [31][32]
  1. Before a function is called, the parameters are pushed onto the stack. [32]
  2. The function's return address is then pushed onto the stack. [32]
  3. Then the frame pointer that contains the previous EBP register value is pushed onto the stack.[32]
  4. Finally, the local variables are also placed in the stack. [32]

✓ Stack Overflow

The following screenshot shows the gcc compiler (-fno-stack-protector) option and input (10 character-long string) that were used to bypass the program's password input requirement.



```
project1@project1-VirtualBox:~/Desktop$ gcc -fno-stack-protector overflow.c -o o
verflow
project1@project1-VirtualBox:~/Desktop$ ./overflow
Please enter a password: 10charssss
Access Granted!
project1@project1-VirtualBox:~/Desktop$ 
```

- Without entering "password", what could you enter in order to display "Access Granted!"?

  First of all, the stack protection has to be turned off; otherwise, we would get a stack smashing error. Finally, after the program is compiled with a fno-stack-protection flag, any password of length greater than 8, will be considered valid by the program as demonstrated in the screenshot above.

- Why is this vulnerability possible and how does it work?

  First of all, the vulnerability is possible because the stack protection is turned off (gcc flag fno-stack-protection). Secondly, when we use the **scanf** function to read **str** (password) in, the function does not check for the length of the input buffer, so the value of **access** local variable is overwritten to a value greater than 0. Finally, the last if statement (line 14) grants access to the system, since it is always evaluated to true (access > 0)

- How could you change the code to make this exploit not possible?

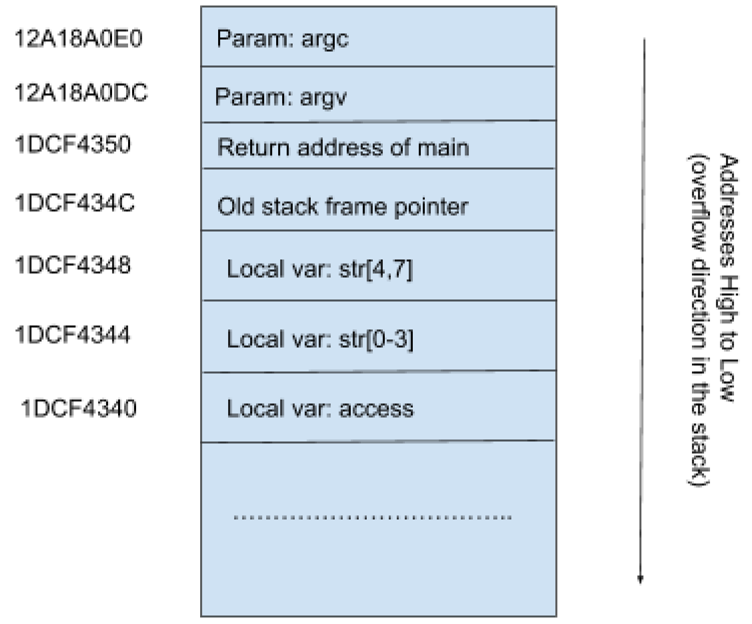
  As detailed in the P1L2 lecture, we could check the input to make sure we reject the invalid inputs. We could also call safe functions to read in input, since it is documented that **scanf**, along with other functions cause stack overflow, due to the lack of input length checks. We could also refactor the code using a safe language, such as Java.

- Would a similar program have the same vulnerability in a strongly typed language? Why or why not?

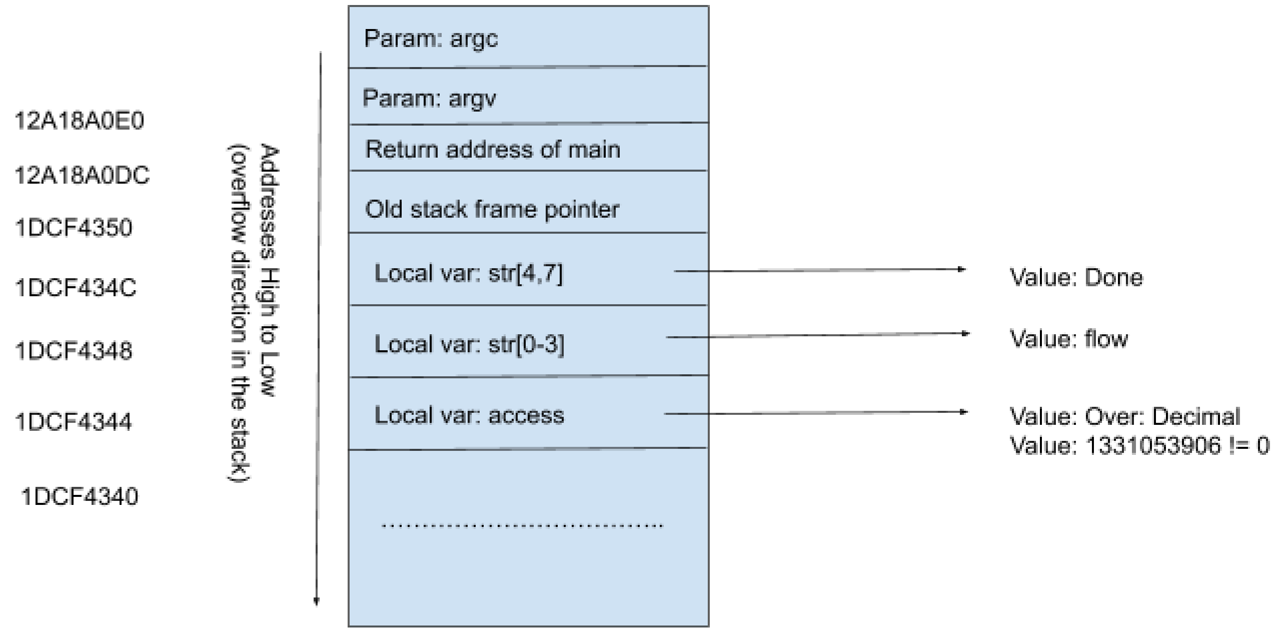
  No, because most of those languages are strongly typed and perform bound checks at runtime, making it impossible for stack overflows to happen.

Stack Overflow Diagram

The Stack before input "OverflowDone", the memory addresses were picked arbitrarily, but decreased in chunks of 4 bytes.



Stack after input “OverflowDone”. Since the input is 12 characters long, the **access** local variable is overwritten with the word **Over**, which when converted to an integer is **1331053906** (> 0). Therefore, the if statement that grants access (overflow.c line 14) will always evaluate to true when passing in the OverflowDone input.



References

[1] C. Singh, Pointers in C Programming with examples, Feb, 2017. Accessed on: May. 20, 2020. [Online]. Available: https://beginnersbook.com/2014/01/c-pointers/
[2] “Pointers, Stack & Heap Memory, malloc()”. March, 2019. Accessed on: May. 20, 2020. [Online]. Available:
https://people.cs.clemson.edu/~levinej/courses/S15/1020/handouts/lec01/MemoryAndMalloc.pdf
[3] “How Memory is Allocated and Deallocated”. March, 2019. Accessed on: May. 20, 2020. [Online]. Available: https://docs.voltdb.com/PerfGuide/MemoryAllocFree.php
[4] N.Thakur. Differences between pass by value and pass by reference in C++, April, 2019. Accessed on: May. 21, 2020. [Online]
https://www.tutorialspoint.com/differences-between-pass-by-value-and-pass-by-reference-in-cplusplus
[5] Programming languages C. May, 2015. Accessed on: May 21, 2020. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf
[6] M. Barr. Portable Fixed-Width Integers in C, Jan, 2004. Accessed on: May. 20, 2020. [Online]. Available: https://barrgroup.com/embedded-systems/how-to/c-fixed-width-integers-c99
[7] “Language Processors: Assembler, Compiler and Interpreter”. March, 2016. Accessed on: May. 22, 2020. [Online]. Available:
https://www.geeksforgeeks.org/language-processors-assembler-compiler-and-interpreter/
[8] Is Python Compiled or Interpreted? Feb, 2017. Accessed on: May. 21, 2020. [Online]. Available: http://net-informations.com/python/iq/interpreted.htm
[9] GCC and Make. Compiling, Linking and Building. C/C++ Applications, March, 2018. Accessed on: May. 20, 2020. [Online]. Available:
https://www3.ntu.edu.sg/home/ehchua/programming/cpp/gcc_make.html
[10] “GCC, the GNU Compiler Collection”, May, 2020. Accessed on: May. 22, 2020. [Online]. Available: https://gcc.gnu.org/
[11] “GDB: The GNU Project Debugger”, May, 2020. Accessed on: May. 22, 2020. [Online]. Available: https://www.gnu.org/software/gdb/
[12] “Registers”, June, 2014. Accessed on: May. 22, 2020. [Online]. Available:
https://ftp.gnu.org/old-gnu/Manuals/gdb/html_node/gdb_60.html
[13] “GDB CheatSheet”, May, 2020. Accessed on: May. 22, 2020. [Online]. Available:
https://darkdust.net/files/GDB%20Cheat%20Sheet.pdf
[14] “info address command”, Aug, 2017. Accessed on: May. 23, 2020. [Online]. Available:
https://visualgdb.com/gdbreference/commands/info_address
[15] “Static in C”, Sep, 2012. Accessed on: May. 23, 2020. [Online]. Available:
https://www.javatpoint.com/static-in-c
[16] “Runtime vs Compile time”, March, 2016. Accessed on: May. 23, 2020. [Online]. Available:
http://net-informations.com/python/iq/checking.htm
[17]S. Sharma. Compile time and runtime memory allocation, May, 2018. Accessed on: May. 21, 2020. [Online]. Available: https://codeforwin.org/2018/05/compile-time-and-runtime-memory-allocation.html
[18] “Python | Compiled or Interpreted?” March, 2016. Accessed on: May. 23, 2020. [Online]. Available: https://www.geeksforgeeks.org/python-compiled-or-interpreted/
[19] “Buffer Overflows, ASLR, and Stack Canaries
”. May, 2017, 2016. Accessed on: May. 23, 2020. [Online]. Available:
https://ritcsec.wordpress.com/2017/05/18/buffer-overflows-aslr-and-stack-canaries/
[20] I. Smith. New bypass and protection techniques for ASLR on Linux, Feb, 2018. Accessed on: May. 20, 2020. [Online]. Available:
http://blog.ptsecurity.com/2018/02/new-bypass-and-protection-techniques.html

[21] “Stack Canaries”, March, 2016. Accessed on: May. 23, 2020. [Online].
Available:https://ctf101.org/binary-exploitation/stack-canaries/
[22] D. Evans. x86 Assembly Guide, Feb, 2006. Accessed on: May. 20, 2020. [Online]. Available:
https://www.cs.virginia.edu/~evans/cs216/guides/x86.html
[23] “ESP register”. May, 2017, 2016. Accessed on: May. 23, 2020. [Online]. Available:
http://www.c-jump.com/CIS77/ASM/Stack/S77_0040_esp_register.htm
[24] S.Friedl. Intel x86 Function-call Conventions - Assembly View, Sep, 2012. Accessed on: May. 23, 2020. [Online]. Available: http://unixwiz.net/techtips/win32-callconv-asm.html
[24] J. Rentzsch. Data alignment: Straighten up and fly right. Feb, 2016. Accessed on: May. 23, 2020. [Online]. Available: https://developer.ibm.com/technologies/systems/articles/pa-dalign/
[25] “gcc -o / -O option flags”, Nov, 2008, 2016. Accessed on: May. 23, 2020. [Online]. Available:
https://www.rapidtables.com/code/linux/gcc/gcc-o.html
[26] “Options for Debugging Your Program”, Dec, 20017, 2016. Accessed on: May. 23, 2020. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Debugging-Options.html
[27] “-fstack-protector, -fstack-protector-all, -fstack-protector-strong, -fno-stack-protector”. Feb, 2005. Accessed on: May. 23, 2020. [Online]. Available:
http://www.keil.com/support/man/docs/armclang_ref/armclang_ref_cjh1548250046139.htm
[28] “Options for Linking”, May, 2020. Accessed on: May. 23, 2020. [Online]. Available:
https://gcc.gnu.org/onlinedocs/gcc/Link-Options.html#index-z
[29] “execstack(8) - Linux man page”, May, 2020. Accessed on: May. 23, 2020. [Online]. Available:
https://linux.die.net/man/8/execstack
[30] “Stack vs Heap Memory Allocation”, May, 2020. Accessed on: May. 23, 2020. [Online]. Available:
https://www.geeksforgeeks.org/stack-vs-heap-memory-allocation/
[31] W. Stallings, L. Brown. Computer Security: Principles and Practices, 4th Edition.
[32] “Buffer Overflow The Function Stack”, May, 2006. Accessed on: May. 23, 2020. [Online].
Available: https://www.tenouk.com/Bufferoverflowc/Bufferoverflow2a.html
[33] “Memory Management: Stack And Heap” , June, 2002. Accessed on: May. 24, 2020. [Online].
Available: http://icarus.cs.weber.edu/~dab/cs1410/textbook/4.Pointers/memory.html