# Stack memory safety

## Summary

In this exercise, we will learn about stack memory safety.

## Prerequisites

Setup an Ubuntu VM as outlined in the VM setup instructions on Blackboard.

## Details

### Part I-Stack memory overview

The SYSV ABI specifies the content of stack frames in stack memory and this includes defined locations for function arguments, stack frame pointers, return pointers, and local variables. Depending on the order of the argument and its type, arguments to called functions can be passed in either registers or stack memory. The table below illustrates the general layout of a stack frame for AMD64 architecture when compiler optimizations are turned off and the base pointer is used for indexing into the stack frame.

Table 1-x86-64 stack frame layout for Linux (lowest address at top; stack grows up here)

|  |  |  |  |
| --- | --- | --- | --- |
| **Position** | | **Contents** | **Frame** |
|  | -128+rsp  …  -8+rsp | Red zone | Current (lowest address)  growth |
| -(M)+rbp    …  -8+rbp | 0+rsp | Local variables in reverse order (i.e., variable 1 is closest to rbp and the last local variable is pointed to by rsp); Here, M is the size in bytes of all the local variables |
| rbp |  | Previous rbp value |
| 8+rbp |  | Return address |
| 16+rbp  …  8n+16+rbp |  | Memory argument eightbyte 7  …  Memory argument eightbyte n | Previous (highest address) |

The next table illustrates several contiguous frame pointers on the stack.

Table 2-Stack layout for multiple stack frames

|  |  |
| --- | --- |
| **Stack frame id** | **Details** |
| Stack frame 4 | Current (lowest address) |
| Stack frame 3 | Previous caller’s  growth |
| Stack frame 2 | Previous caller’s |
| Stack frame 1 | Previous caller’s (highest address) |

Compilers implement control flow with stack frames by inserting snippets of assembly instructions that are appended to the beginning and end of functions. These are respectively called the prologue and epilogue. For our vulnerable(char \*, int) function in this class exercise, gcc generates the snippets of assembly shown in the next two tables.

Table 3-Prologue example

|  |  |
| --- | --- |
| **Instruction** | **Comments** |
| push rbp  mov rbp,rsp  sub rsp,0x90  mov QWORD PTR [rbp-0x88],rdi  mov DWORD PTR [rbp-0x8c],esi | Save previous rbp  Copy current rsp to rbp (current stack frame)  Allocate stack space  Loads argument 1 from rdi to stack  Loads argument 2 from esi (rsi) to stack |

Table 4-Epilogue example

|  |  |
| --- | --- |
| **Instruction** | **Comments** |
| leave  ret | Copies the frame pointer (RBP) into the stack pointer register (RSP), which releases the stack space allocated to the current stack frame.  The old frame pointer (the frame pointer for the calling procedure that was saved by the ENTER instruction) is then popped from the stack into the RBP register, restoring the calling procedure’s stack frame.  Transfers program control to a return address located in RSP |

### Part II-Stack smashing to corrupt stack frame

Buffer overflow vulnerabilities are possible for binaries compiled from C/C++ because, by design, these languages were constructed to trust the programmer.

* Why do you think the creators of the C/C++ language followed this design principle of trusting the programmer?

Buffer overflow vulnerabilities facilitate out-of-bounds copy operations in memory. When the destination for these copy operations is the stack, it is possible to tamper with the stack frame metadata. In today’s exercise, we will overwrite the current stack frame’s previous rbp value and return function pointer (as well as other things higher on the stack after our vulnerable buffer that we are able to overflow).

#### Demo

* Download the project into a local sandbox

$ git clone <https://gitlab.com/underpantsgnomes/softwaresecurity/stacksmashexploit>

* Turn off address-space layout randomization in Ubuntu (ASLR) using

$ echo 0 | sudo tee /proc/sys/kernel/randomize\_va\_space

* Ensure ASLR is disabled by confirming the output of running the following command is 0

$ cat /proc/sys/kernel/randomize\_va\_space

* Import the project in Eclipse
  + File->Open Projects From File System
* Confirm build settings for project has the following settings into the "Other flags" portion of the "Miscellaneous" section of tool settings for the GCC C compiler
  + Compile separately prior to a later linking step (-c)
  + Enable position independent code (-fPIC)
  + Specify to retain frame pointers (-fno-omit-frame-pointer); note: targeted leaf functions (ones that don’t call anything else) might not retain frame pointers even with this setting
  + Disable format string compile time warnings (-Wno-format-security); note: prevents build errors when –Wall is specified
  + Disable Ubuntu’s GCC-SSP enabling of run-time stack overflow verification via stack canaries (-fno-stack-protector)
  + Specify intel dialect for outputted assembly (-masm=intel)
* Confirm build settings for project has the following settings into the "Other flags" portion of the "Miscellaneous" section of tool settings for the GCC C linker
  + Enable executable stack for the linker (-z execstack)
* exploitWrapper.c has two alternative modes, set the sledge-hammer mode by enabling its preprocessor macro and disabling the ball-peen hammer mode
  + Sledge hammer mode passes in a crafted buffer to vulnerable program and a length to copy from the buffer
  + Inspect the exploit code
* Build the project
* Create the debug target configuration
* Select new launch configuration for a C/C++ Application
* Specify the wrapper binary built from the console as the target
* Insert the following three program arguments

|  |
| --- |
| ${string\_prompt: shellcodeaddr:0x7fffffffd680} ${string\_prompt: oldrbp:0x7fffffffd730} ${string\_prompt: bufferlen:0x10} |

* Go to the Debugger tab and specify an environment that uses ~/.gdbinit
* Set a breakpoint on line 15 in vulnFunction::vulnerableUUT.c
* Run the demo
* Continue to the breakpoint and locate the stack frame for vulnFunction::vulnerableUUT.c
  + Locate the local variables on the stack frame
  + Locate the previous rbp and return function pointer on the stack frame
* Step through the buffer overflow vulnerability being exploited at the memcpy and discuss what is happening amongst your team. Identify the buffer overflow vulnerability.
* Determine the minimum size for the command line argument to exploitWrapper, bufferlen, that crashes in vulnFunction::vulnerableUUT and verify this

### Part III-Stack smashing to control rip

* To control the rip via a stack-smashing attack, we want to overflow and overwrite the previous RBP with a valid, dummy address, and the return pointer with our shellcode address. If stack protections are disabled, when the epilogue runs, the shellcode will be executed.
* Set the ball-peen hammer mode in exploitWrapper.c by disabling the sledge-hammer mode via its preprocessor macro and similarly enabling the ball-peen hammer mode
  + Ball-peen hammer mode passes in a crafted buffer to vulnerable program and a length to copy from the buffer. This time, the crafted buffer overwrites the old rbp and return function pointers on the stack frame for vulnFunction::vulnerableUUT.c.
  + Inspect the exploit code and discuss amongst your team
* Run the program in Eclipse and step through the assembly instructions of the shellcode when it is called
  + If it segfaults, you need to adjust shellcodeaddr, oldrbp, and bufferlen arguments passed in to exploitwrapper appropriately (also ensure that ASLR is disabled in OS and also in gdb via ~/.gdbinit)
  + Execute the following in the bash shell when your shell code call works (i.e., when you get the $ bash prompt in your debugging session)

|  |
| --- |
| $ /usr/games/cowsay -f dragon "Grrr!"  < Grrr! >  --------  \ / \ //\  \ |\\_\_\_/| / \// \\  /0 0 \\_\_ / // | \ \  / / \/\_/ // | \ \  @\_^\_@'/ \/\_ // | \ \  //\_^\_/ \/\_ // | \ \  ( //) | \/// | \ \  ( / /) \_|\_ / ) // | \ \_\  ( // /) '/,\_ \_ \_/ ( ; -. | \_ \_\.-~ .-~~~^-.  (( / / )) ,-{ \_ `-.|.-~-. .~ `.  (( // / )) '/\ / ~-. \_ .-~ .-~^-. \  (( /// )) `. { } / \ \  (( / )) .----~-.\ \-' .~ \ `. \^-.  ///.----..> \ \_ -~ `. ^-` ^-\_  ///-.\_ \_ \_ \_ \_ \_ \_}^ - - - - ~ ~-- ,.-~  /.-~ |

* Rebuild the project and enable stack protections. What happens when you run the exploit this time?
* Have a look at the global variables shellcode and spacer in exploitWrapper.c (specifically, note all the \x90 padding bytes)
  + What cpu instruction is \x90?
  + Remove or add a few \x90 padding bytes from the variable shellcode. Then, add or remove the same \x90 padding bytes from the variable spacer and rebuild/rerun the exploit. What is the error you receive and why does it not work?
  + Revert your change to the variables shellcode and spacer. Remove or add a few \x90 padding bytes from the variable spacer only and rebuild/rerun the exploit. What is the error you receive and why does it not work?

## References

* Aleph1, Smashing the stack for fun and profit
* System V Application Binary Interface, AMD64 Architecture Processor Supplement
* 2013, Seacord, Secure Coding in C and C++, 2e, 2.3-2.4 (Buffer overflows from string vulnerabilities)