Assignment 1A – Buffer overflow & Race conditions

1. Disabling ASLR

> cat /proc/sys/kernel/randomize_va_space

What is the current value?

2

What does it mean?

Address Space Layout Randomization (ASLR) obscures the memory locations of the parts of computer processes by making them non-sequential, and thus more difficult to predict and abuse.

va_space refers to *Virtual Address Space* (VAS). The VAS maps processes in memory to main memory (RAM) and secondary storage (usually disks).

A setting of *0* (zero) disables ASLR.

A setting of *1* (one) enables randomization of the address space, which includes the locations of the call stack, the *virtual Dynamic Shared Object* (vDSO) page, and shared memory areas.

A setting of *2* includes the functionality of setting *1*, and additionally randomizes the locations of data segments. Data segments refer to initialized static variables of programs.

2. Insecure strcpy()

Why is *strcpy()* vulnerable?

As the *strcpy()* function does not include a parameter that limits the size of the string to be copied to the destination, buffer overflow can occur if the source string exceeds the size of the destination array.

Which method is recommended instead of *strcpy()* to prevent buffer overflow?

strncpy(), or *strlcpy()* (if available).

Why is *strncpy()* or *strlcpy()* safer?

Both functions include a parameter to indicate the size of the destination array, and truncates the source string to fit. This prevents the source string from overrunning the destination buffer.

However, *strncpy()* does not ensure the copied string is null-terminated, while *strlcpy()* does. Non-null-terminated strings may be a source of more bugs.

3. Running the code

Why do we need to include the command '*-fno-stack-protector*' while compiling?

The *-fstack-protector* flag will include stack canaries into the compiled code which guard against string buffer overflows.

-fno-stack-protector thus disables the stack overflow security checks for routines with a string buffer.

Since we want to test a stack overflow exploit, we have to compile the code with stack protection disabled, otherwise our code will abort when the canaries detect stack smashing.

What is the significance of *$0x1fc* in the third line '*sub $0x1fc,%esp*'?

'*sub $0x1fc,%esp*' represents a subtraction (*sub*) of the literal value (*$*) 508 (hexadecimal *0x1fc*) from the register (*%*) holding the extended stack pointer (*esp*).

This indicates that the size of the stack frame is 508 bytes, and allows us to carry out a buffer overflow attack by exceeding the stack frame size to overwrite critical values in the stack frame, such as the return address at *4(%ebp)*.

Subtraction is carried out because the stack grows from a high memory address to a low memory address. To reserve space on the stack, 508 is subtracted from the ESP, which was initially assigned the memory location held by the extended base pointer (EBP; indicating the start/anchor of the stack). This effectively points the location of the top of the stack 508 bytes away from the base.

Of the 508 bytes allocated, 500 bytes are to fulfill the requested buffer size (*char buffer[500];*). An additional four bytes from *(%ebp)* to *-3(%ebp)* are taken up by the EBP to optionally store the previous *%ebp*. Four more bytes are taken up by the "unspecified" area of the standard stack frame from *-4(%ebp)* to *-7(%ebp)*. (See Figure 3-15 below; extracted from "Intel386 Architecture Processor Supplement, Fourth Edition," 1997.)

Figure 3-15: Standard Stack Frame

Position	Contents	Frame	
4n+8 (%ebp)	argument word n	Previous	High addresses
	. . .		
8 (%ebp)	argument word 0		
4 (%ebp)	return address	Current	
0 (%ebp)	previous %ebp (optional)		
-4 (%ebp)	unspecified		
	. . .		
0 (%esp)	variable size		Low addresses

No additional stack alignment is required as the compiler is called with *-mpreferred-stack-boundary=2* (alignment of 2^2 = 4 bytes; 508 % 4 = 0).

The assembly code later eventually fills in the *char* buffer from *-500(%ebp)* as can be seen from:

0x080483f9 <+21>: lea -0x1f4(%ebp),%eax

Explain the argument string *$(python -c 'print "\x41" * 508')*.

Running *$(command)* in *bash* carries out command substitution. The contents of *command* is executed in a subshell environment, and the output of *command* replaces the *$ (command)* itself.

python -c command executes the *command* string statements using the *python* interpreter.

*'print "\x41" * 508'* when run in *python* will result in the character with a hex value of 41 – the letter *A* – to be output 508 times.

This effectively passes the 508 character long string 'AAA...A' to the *run* command as an argument.

Explain the error code "*Program received signal SIGSEGV, Segmentation fault. 0x41414141 in ?? ( )*".

SIGSEGV stands for *Signal: Segmentation Violation*, more verbosely followed by *Segmentation fault*. This occurs when a program tries to read or write a memory location without access rights.

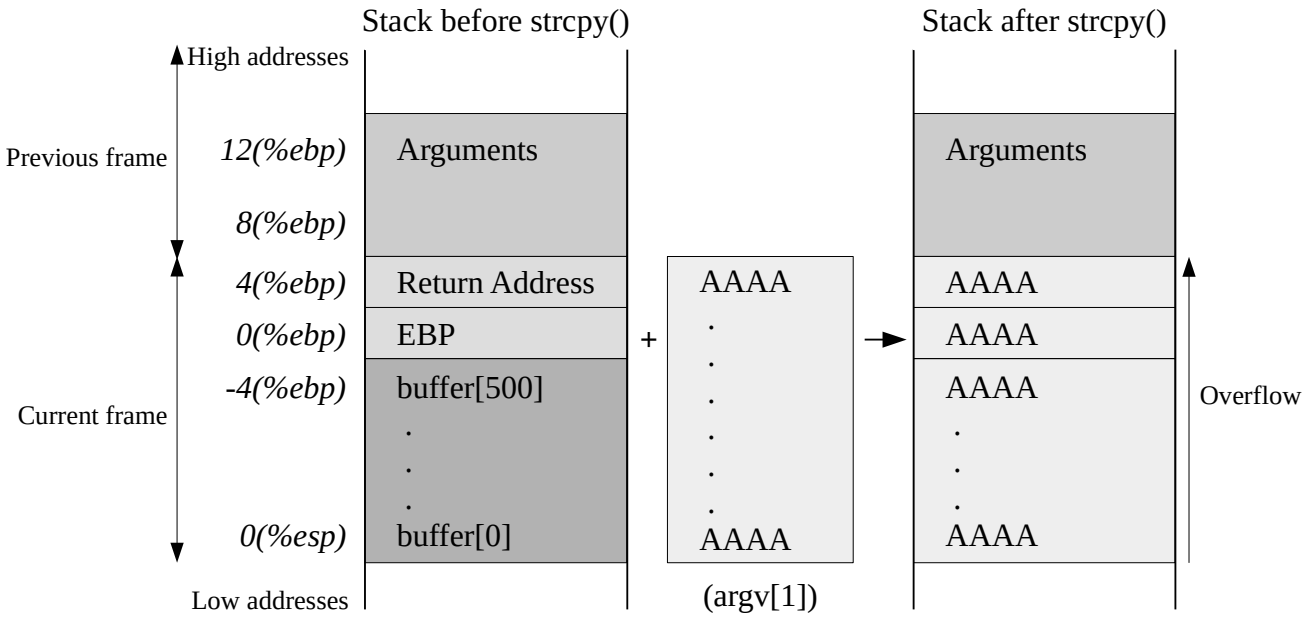
Here we see the trace reporting an erroneous address of *0x41414141* injected by the buffer overflow called by an unknown function *?? ( )*.

Explain the value of EBP & EIP.

By writing 508 'A's into the buffer starting at *-500(%ebp)*, the value of the return address (*RET*) at *4(%ebp)* to *7(%ebp)* is overwritten by *0x41414141* (AAAA).

This causes the Extended Instruction Pointer (EIP) to obtain the value of *0x41414141* as the assembly code reaches the *ret* instruction, and triggers the *SIGSEGV* as it tries to access that memory location.

Draw a diagram similar to the slides to show the contents of the stack and explain the overflow.



By providing an input exceeding the allocated buffer size, *strcpy()* overwrites the EBP as well as the return address (*RET*) in the current stack. This causes the EIP to receive the wrong *RET*. Thus, by setting the *RET* to point to malicious code which we include in the input, we can carry out a buffer overflow attack.

4. Prepare the attack

Why is *NOP* (No Operation) useful in a buffer overflow attack?

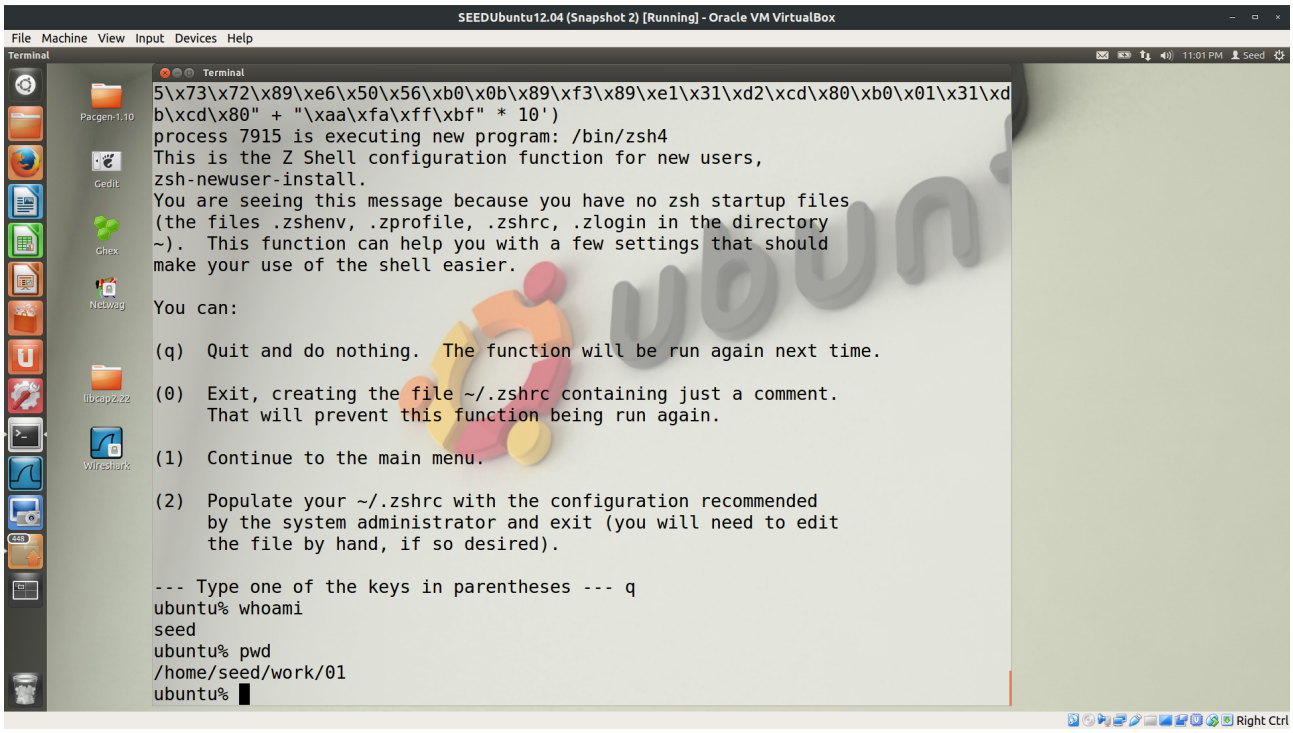
When running buffer overflows on unknown systems, the attacker generally will not know the exact address for payload execution due to variable factors and environments. In order to increase the chance of the overwritten *RET* jumping to and executing the malicious code, a series of *NOP* instructions can be pre-pended to the code being injected to extend the "landing zone" of the *RET* jump. As long as the *RET* jumps to anywhere within the *NOP* sequence, the execution will then "slide" down the *NOP*s and reach the malicious code.

Why do you have to change the target address to *big endian*?

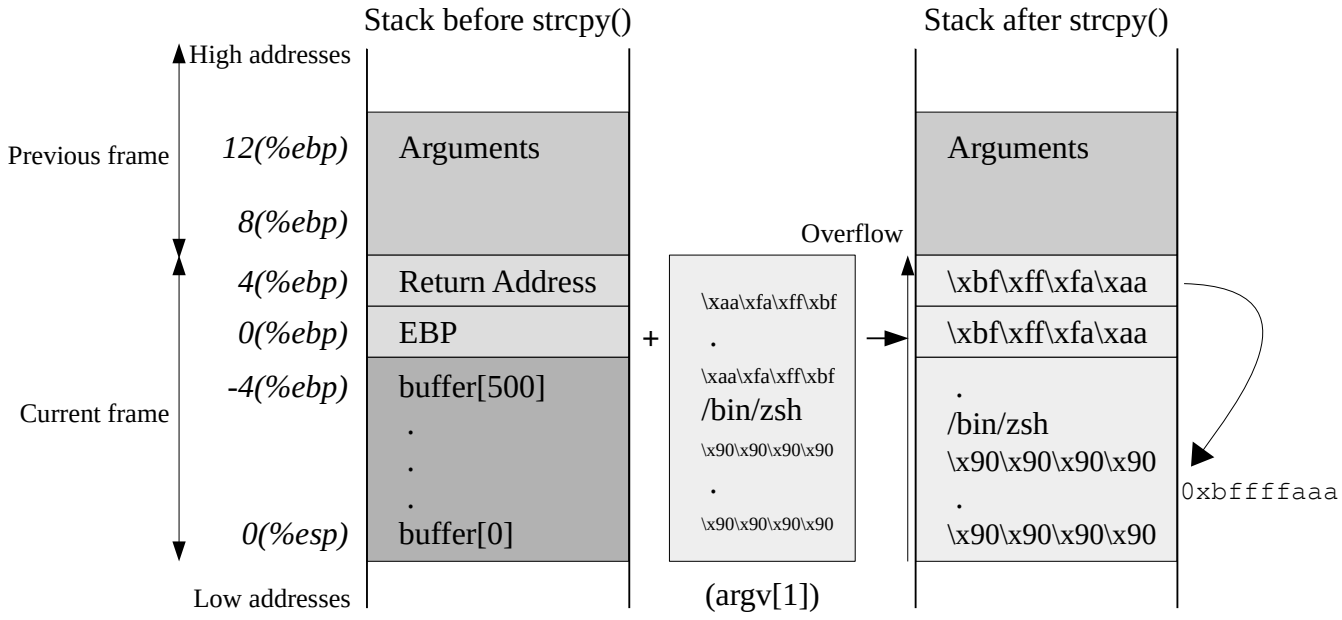
The *big endian* format refers to storing the most significant byte in the lowest address first, with following bytes of decreasing significance stored in higher addresses last.

As the input string grows from a lower address to a higher address, it occupies the stack in the opposite direction which the stack grows. However, on stack return, the 4-byte *RET* is popped and read last in first out (LIFO) in *little endian*. Therefore, the target address must be reversed using the *big endian* format in the input string for the *RET* to return the intended target address.

From the *zsh* shell, run the ‘whoami’ and ‘pwd’. Take a screenshot to show the exploit did run.



Draw the content of the stack to show the exploit and explain how it was able to run successfully.



The *RET* is overwritten with a memory address that falls on the *NOP* slide. As the *strcpy()* function completes, and the *main()* function calls the modified *RET* address, the *EIP* then points back within the same stack, running through the *NOP*s until it reaches the command that executes */bin/zsh*.