<u>1) After running ls -la in any directory you will see two directories named "." and "..". What are these directories and why are they in every directory?</u>

. (dot) means the current directory you’re in, and .. (dot dot) means the parent directory of the current directory you’re in. If dot dot is used in the root directory, it will refer to itself since it is the parent directory, and doesn't have a parent directory.
They're two special entries in every directory, in most operating systems supporting the hierarchical directory system. When a new directory is created, it's automatically linked to it's parent in the hierarchical directory system, so they will always have the dot dot referring to their parent, and the dot referring to itself.

<u>2) What command would you type to find the owner of a file named "secrets.dat"?</u>

ls -l /path/to/file where the third field in the output is the user and the fourth is the group.
If you only want the owner, not other data: stat -c '%U' </path/to/secrets.dat> If some older Linux systems don't support that ls -ld can be used like this:
ls -ld </path/to/secrets.dat> | awk 'NR==1 {print $3}'

<u>3) What command would you type to change the owner of a file named "secrets.dat" to your username?</u>

chown <myusername> secrets.dat

<u>4) What would be the effect of running the command chmod -R 777 .? Why would you want to be very careful when running this particular command?</u>

chmod changes the permissions of a directory or file, and using 777 as arguments opens up the the directory and all of its files to every user. 777 means every user has the rights to not only access this folder, but also read, write and execute it’s files. The octal permission 777 is equivalent to the symbolic permission ugo=rwx. This is because read is 4, write is 2 and execute is 1, so 7 is 4+2+1, meaning all of the rights; and the first 7 is assigned to the owner, the second to it's group and the last to others, so if we would have 740 it would mean the owner has all the rights, the group only reading rights and others no permissions.

<u>6) What is an environment variable and how could it be used in conjunction with the Shellshock bug to remotely exploit a web server?</u>

Environment variables are available to all the sub-shells of the parent-shell that has the environment variables. Normal variables only live inside that instance of the shell they're declared in. It's important to note that if you create an environment variable by using export varName=value, it will only be available to the shells started from the shell they were exported in; if you open up another terminal you won't find that variable, but you can access the variable if you for example run the command bash inside the one you created the variable in. Once the shell that created the environment variable is closed, the variable is no longer available. If the environment variable is added to the startup shell, it'll be available to the shells on the computer. If you for example write the command export varName=value at the end of the .profile file inside your home directory and restart, you can open a terminal and write echo $varName to get it's value printed.
Shellshock is a vulnerability that exploits systems running older versions of the Bash shell (up to GNU Bash 4.3), and it puts the system at risk by letting commands with higher privileges get executed. The bug/vulnerability causes Bash to execute unintentional bash commands from environment variables. Attackers can remotely send commands to the target host. When it comes to servers, bash isn't that common when it comes to the networking side, but many internal and external services such as web servers do use environment variables to communicate with the server’s operating system. This is where shellshock can be used if the data inputs aren't sanitized, meaning ensuring that code is not part of the input, before execution making it possible for attackers to launch HTTP request commands executed via the Bash shell.

<u>7) Explain how an attacker can exploit the buffer overflow.</u>

Basically all operating systems, and most systems programs are written in the C or C++ programming languages. Neither of these languages checks for array bounds. So there are cases where user input can overflow the maximum size of the array, and some functions that read user input and store them into variables don't limit how much input the user writes. A function called gets is notorious for this issue, and gets is very commonly used. Gets reads the input until it encounters a newline character, and doesn't care about the size of the input. There are other functions than gets that also have this problem, but gets is very commonly used so it's also commonly used as an example when explaining buffer overflow. So if the size of an array is 128, and the user input is 256 and gets is used to read the input into a variable, because gets does not check for buffer bounds violations the overflowing bytes will be stored on the stack overwriting existing data on the locations the overflowing data will be stored on.

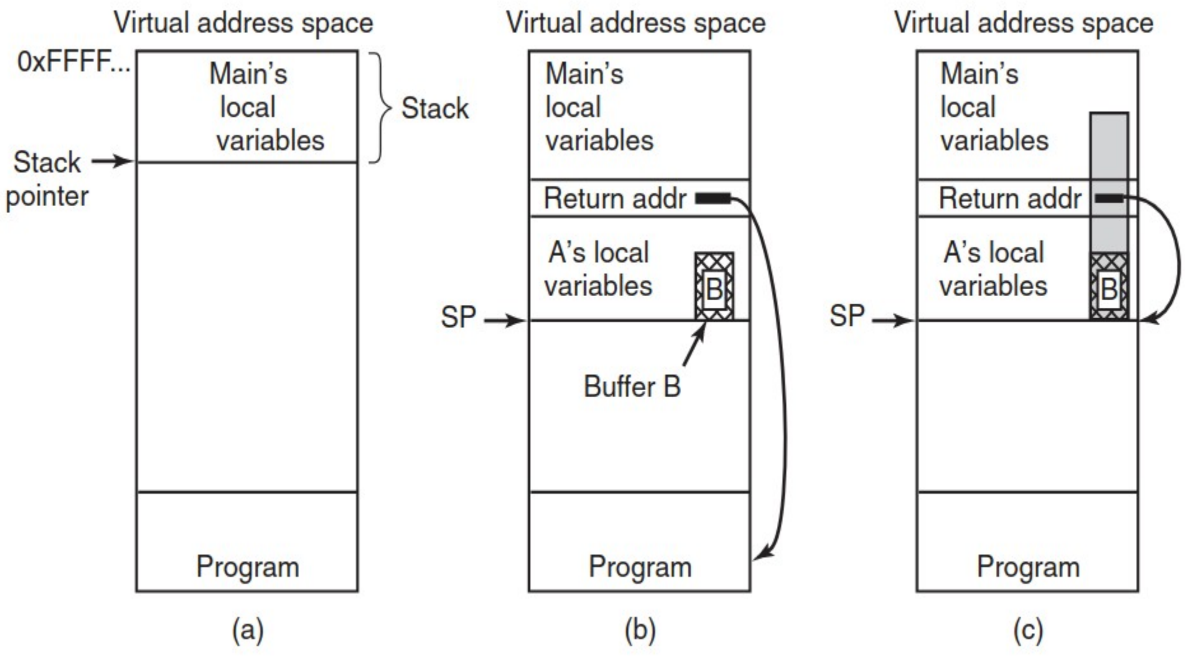
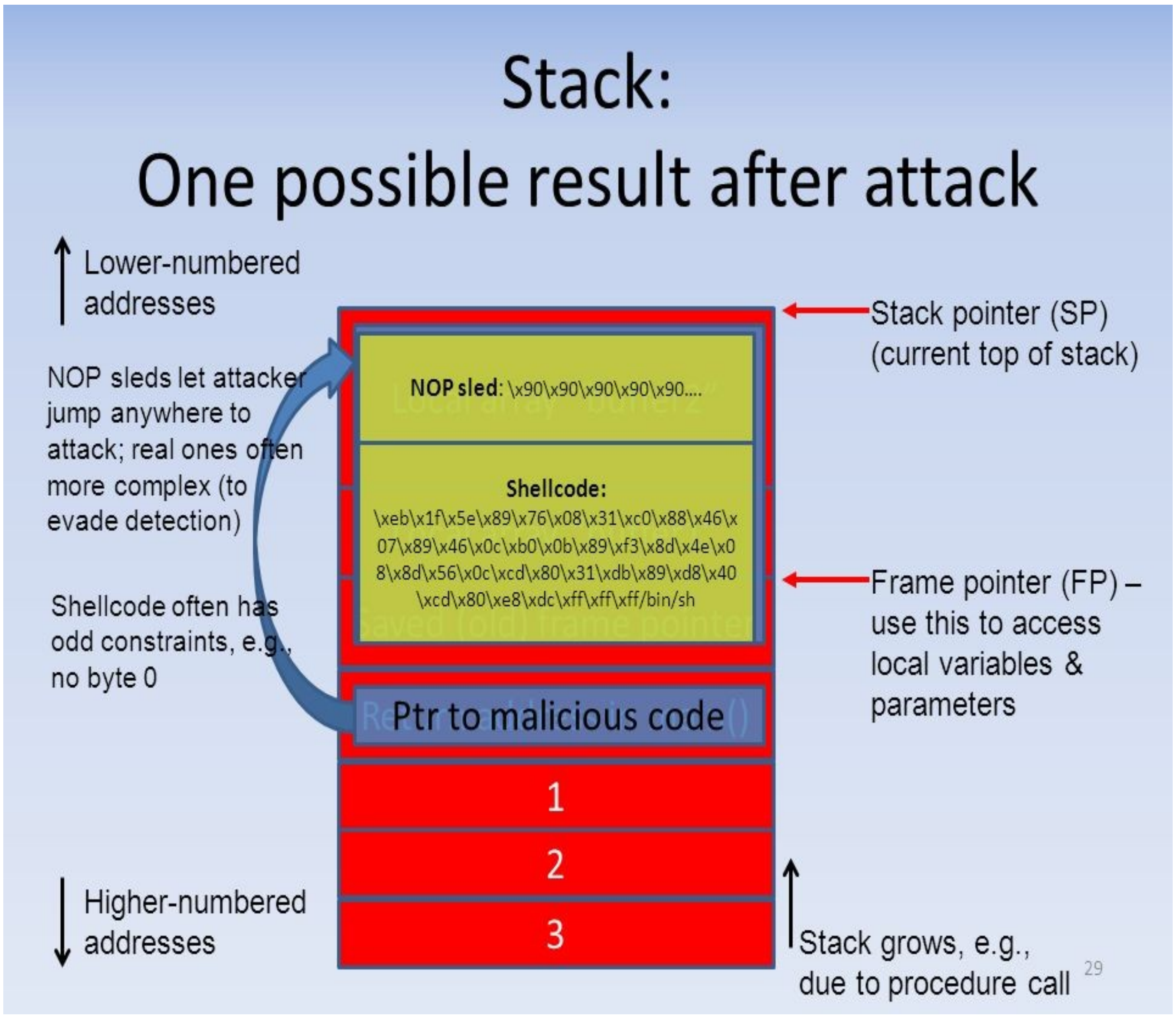


Figure 9-21. (a) Situation when the main program is running. (b) After the procedure *A* has been called. (c) Buffer overflow shown in gray.

In Fig.9-21(a), we see the main program running, with its local variables on the stack. At some point it calls the procedure A, as shown in Fig.9-21(b).The standard calling sequence starts out by pushing the return address (which points to the instruction following the call) onto the stack. It then transfers control to A, which decrements the stack pointer by 128 to allocate storage for its local variable(buffer B).
In Fig.9-21(c) we see what happens if the user provides more than 128 characters. Many things on the stack can be overwriting, but the most vital part is that it overwrites the return address pushed there earlier from the proceduce A; so now every time function returns it will jump to the overwritten return address. In cases where this happens without malicious intent the characters of the message will not be a valid code address, and since it isn't valid code the result is the program crashing immediately.
In cases where it happens because of an attack, the return message will be written with the goal of subverting the program's control flow. The program will jump to the beginning of overflowing buffer and execute its bytes as code. So the attacker is free to fill the overflowing buffer with machine instructions to execute the attacker's code inside the original program, and taking control over the program. In many cases, a shell is launched, this is often done with the exec system call, resulting in the attacker to be inside the host machine. This type of code is commonly referred to as shellcode, even if no shells are launched in the attack.
Memory may move around a bit during execution of the program, so the exact starting address of the shellcode in the buffer is unknown. To get the program to reach the shellcode address, a common solution is attaching the shellcode with a nop sled: a sequence of one-byte NO OPERATION instructions that don't do any code execution; if the attacker manages to land anywhere on the nop sled when returning to the

overwritten address, the nop sled will lead the code to eventually reach the malicious shellcode in the end. Nop sleds work on the stack and the heap. Malicious JavaScript code in browsers can be used to perform the attack on the heap. The code will allocate as much memory as it can, filled with a long nop sled and a small amount of shellcode. If the attacker is able to divert the control flow, the attacker can hit the nop sled by aiming for a random heap address, this is called heap spraying.


Stack:
One possible result after attack
Lower-numbered addresses
NOP sleds let attacker jump anywhere to attack; real ones often more complex (to evade detection)
Shellcode often has odd constraints, e.g., no byte 0
Higher-numbered addresses
NOP sled: \x90\x90\x90\x90\x90....
Shellcode:
\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh
Ptr to malicious code
1
2
3
Stack pointer (SP) (current top of stack)
Frame pointer (FP) – use this to access local variables & parameters
Stack grows, e.g., due to procedure call
29