Return-to-libc attack

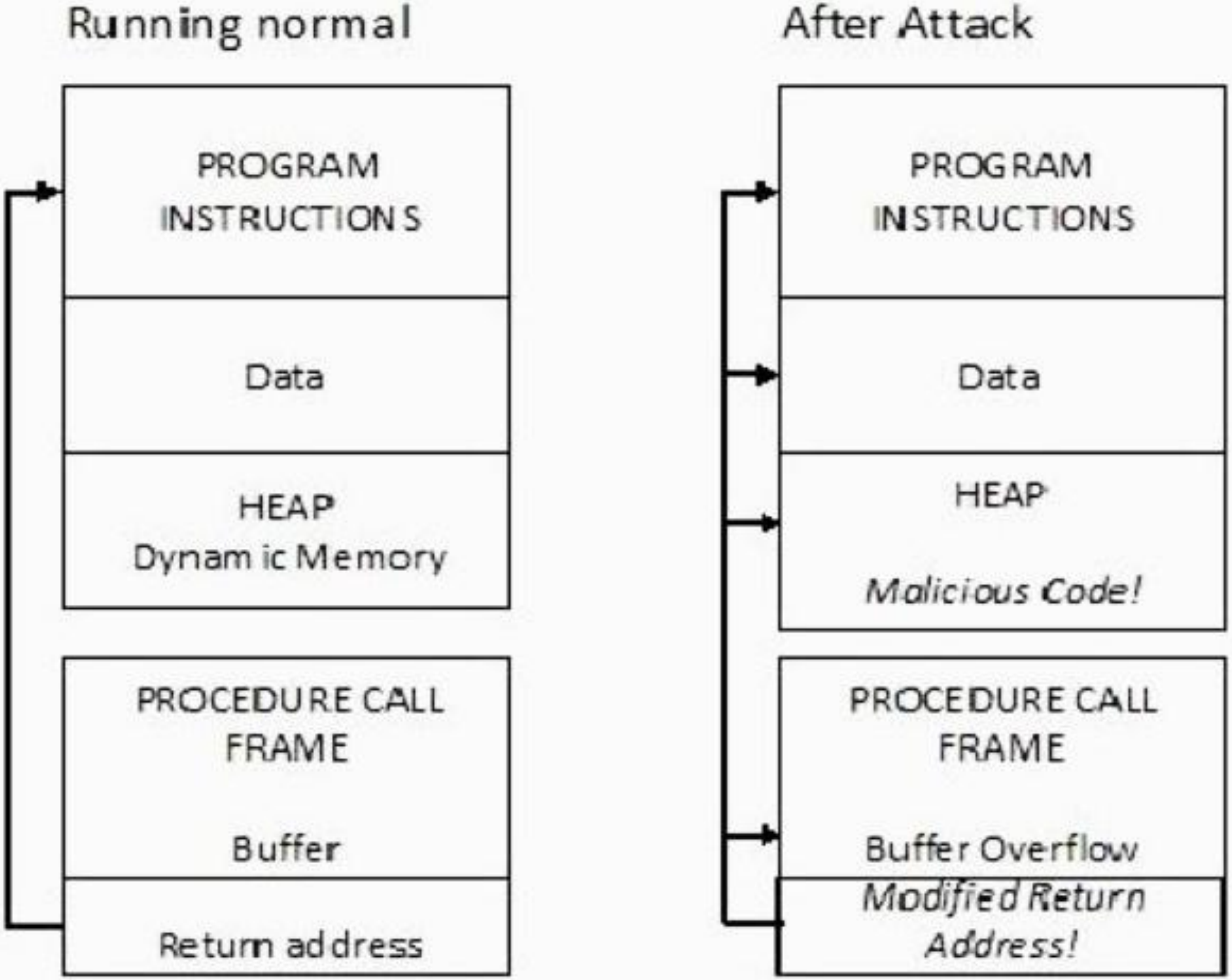
TCS 591 : Unit 2

Buffer Overflow Attack

- A buffer overflow was one of the very first vulnerabilities, so when it was published, back in 1996, information security wasn’t a popular field, and it wasn’t clear how to go about it.
- A buffer overflow is a condition where a variable is overstuffed with data and "arbitrary" (i.e., the hacker's) code is executed. This code can be anything, but ideally, it a command shell or terminal to give **hacker control of the victim system.**

Buffer Overflow Attack

- Buffer overflows are far and away the most dangerous and destructive vulnerabilities within any application or operating system. In its simplest form, a buffer overflow is simply a variable that does not check to make sure that too much data is sent to it (bounds checking) and when too much data is sent, the attacker can send and execute whatever malicious code they want in that address space.



Attacker plants code that overflows buffer and corrupts the return address. Instead of returning to the appropriate calling procedure, the modified return address returns control to malicius code, located elsewhere in process memory.

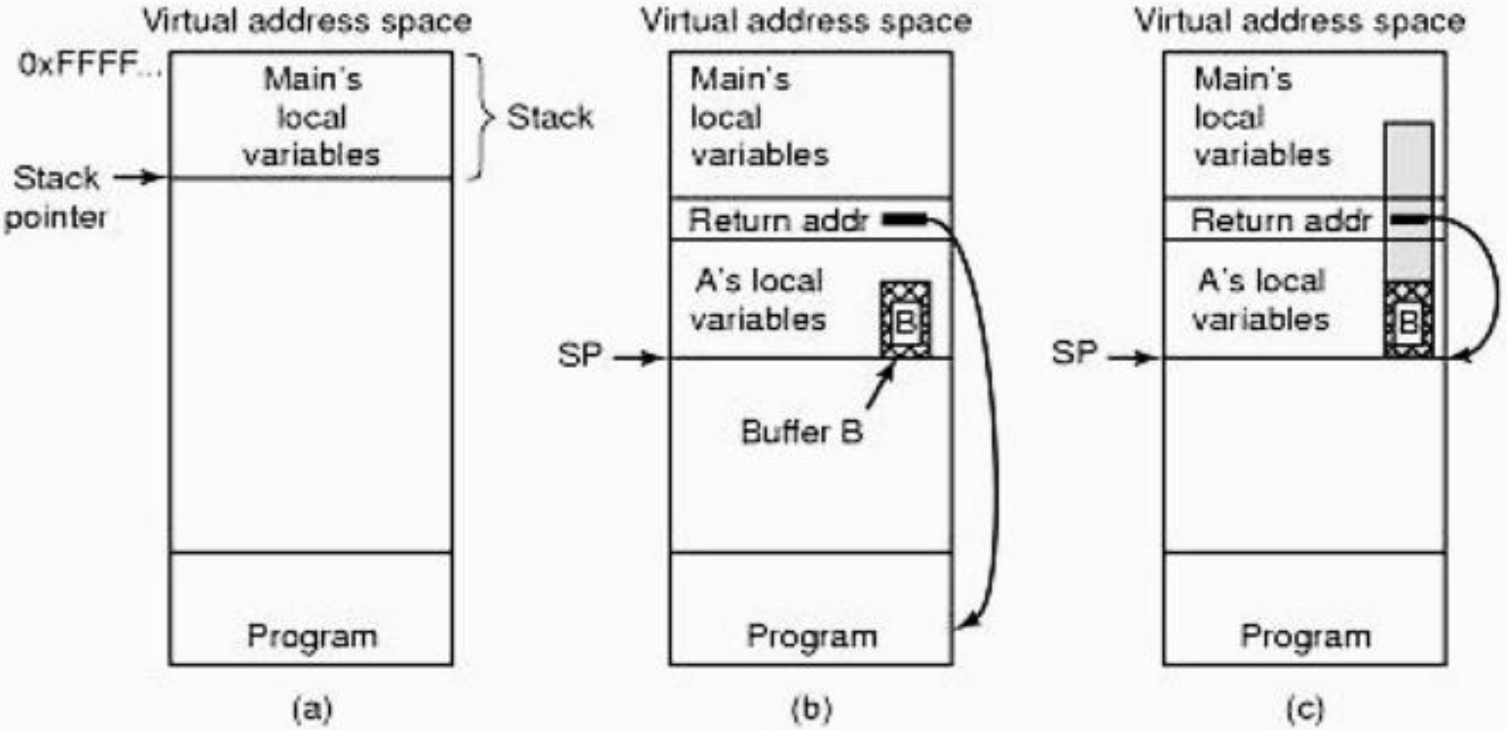
Memory Basic To Understand Buffer Overflow

- To understand buffer overflows, you need to understand a bit about memory. Let's use a simple analogy.
- Let's imagine that our memory is like a large three-ring binder. When a new program executes, it begins to fill up the pages in this three-ring binder with **data (STACK) it needs**, filling it from the **TOP** towards the **Bottom**.
- When the program begins it execution, it requires temporary data that it uses and discards quickly. It then fills the binder with this data from the Bottom toward the Top **(HEAP).**

Stack Vs Heap

- Stack is short term memory, is fixed in size, and is used to store function arguments, local variables, etc.
- Heap is long-term memory and holds dynamic memory.

Buffer Overflow



- (a) Situation when main program is running
- (b) After program *A* called
- (c) Buffer overflow shown in gray

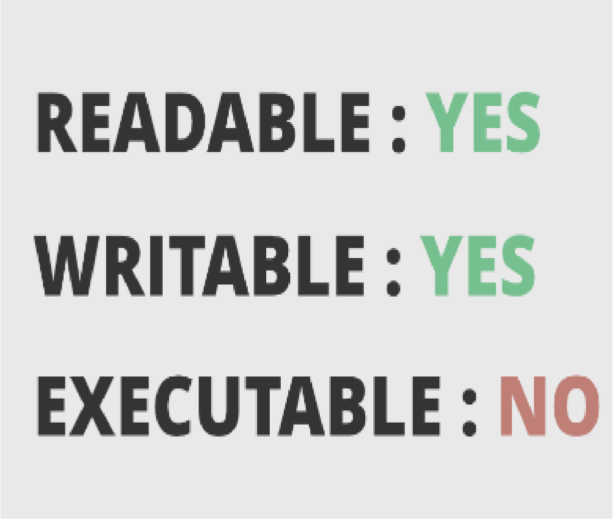
Permissions in Memory Region

- Every memory region has certain characteristics that are enforced on the hardware level. It can be **Readable**, in which case we have permissions to read it, it can be **Writable**, in which case we have permissions to write to it, and what the **security experts** did was introduce another characteristic, whether it's **Executable**, in which case we have permission to run this memory as if it was code.

MITIGATIONS FOR BUFFER OVERFLOW ATTACK

Solution : NX bit

- This is often called the **NX bit, for no-execute**, so we can map the stack, that memory region where buffers and **return addresses** are stored, as non-executable.
- Now, even if an attacker successfully overflows a buffer, overwrites the return address, and diverts the program execution to this buffer, the hardware will refuse to execute this buffer as code, and abort the program.



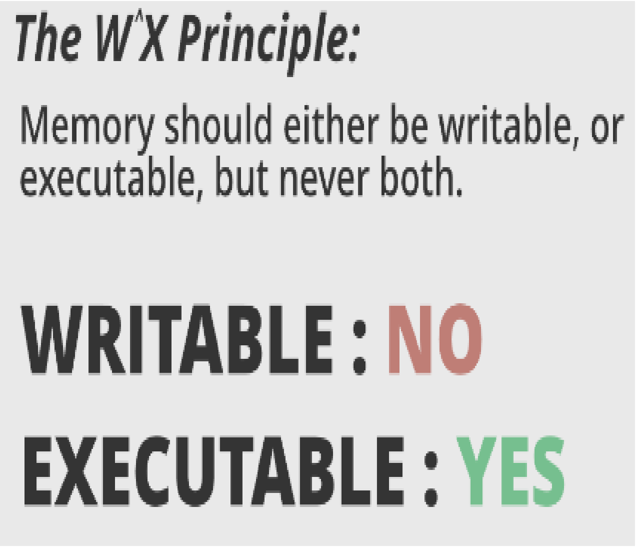
READABLE : YES

WRITABLE : YES

EXECUTABLE : NO

Solution : W^X Principle

- Memory should be either writable or executable, but never both because then an attacker might be able to write some arbitrary code there and divert the program execution to run it



The W^X Principle:

Memory should either be writable, or executable, but never both.

WRITABLE : NO

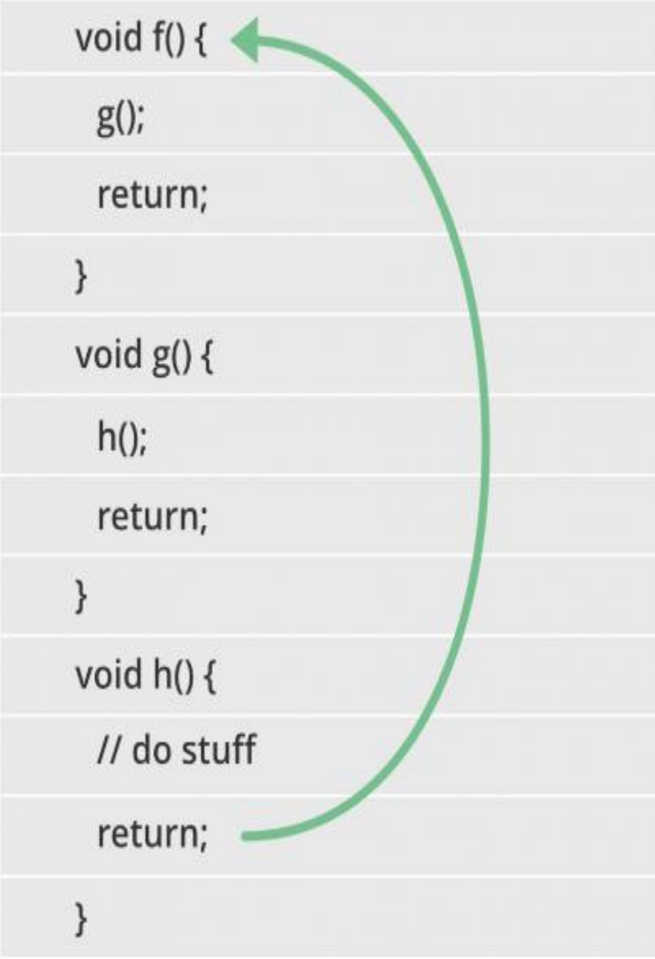
EXECUTABLE : YES

Data Execution Prevention(DEP)

- Useful mitigation is called Data Execution Prevention, or DEP.
- DEP doesn’t stop buffer overflows, but rather prevents code execution on the stack.
- We can still **overwrite functions’ return addresses**, but having lost the ability to write executable code on the buffer, we need to figure out where to jump and what to run instead

Return Addresses

- Every process has code somewhere in memory which defines its execution flow. In fact, this is where function return addresses point to in the first place.
- If function **f** calls function **g**, which calls function **h**, in which we overflow the stack, we can change the return address so instead of returning to **g**, **we skip it and return straight to f**

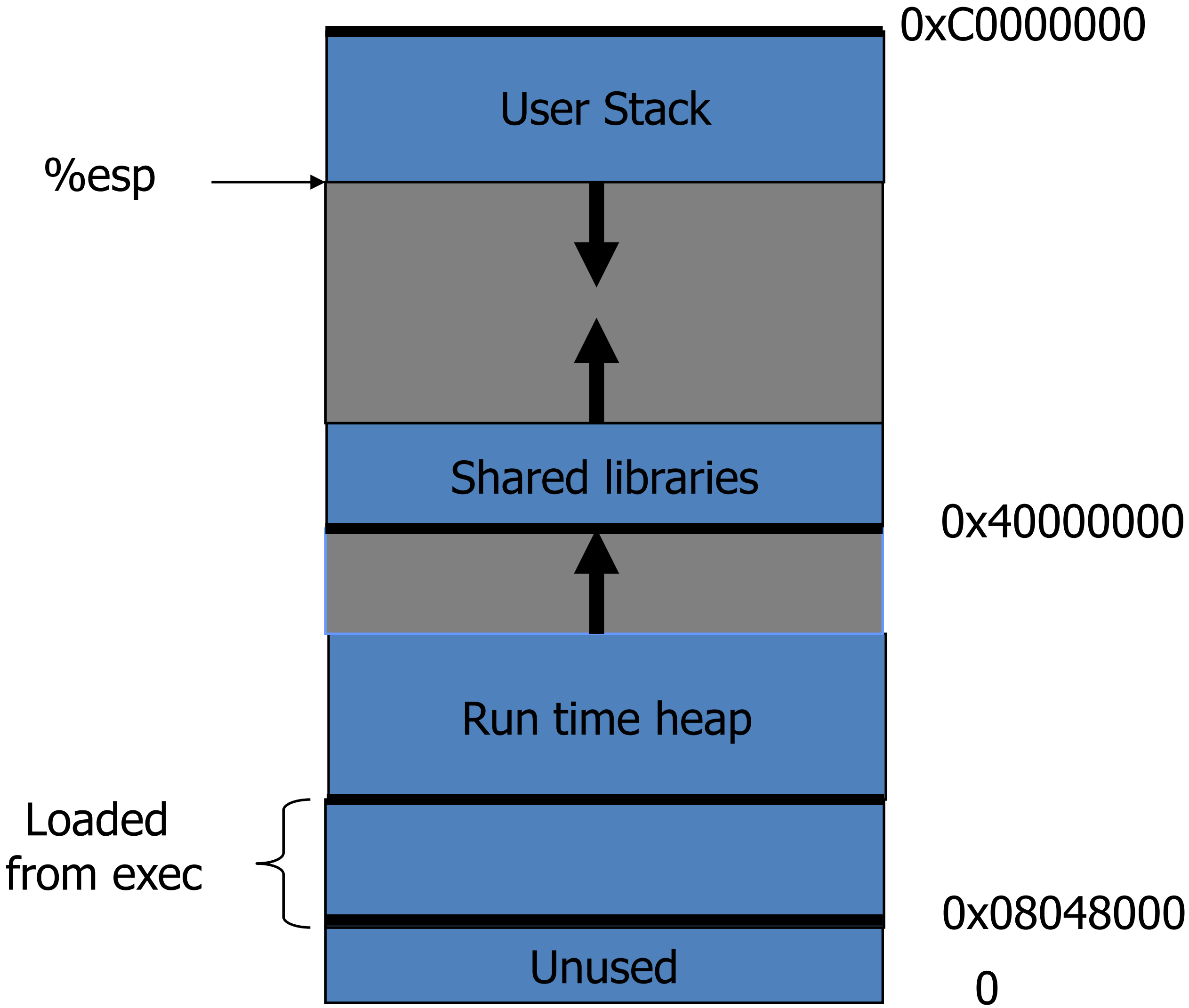


```
void f() {
  g();
  return;
}
void g() {
  h();
  return;
}
void h() {
  // do stuff
  return;
}
```

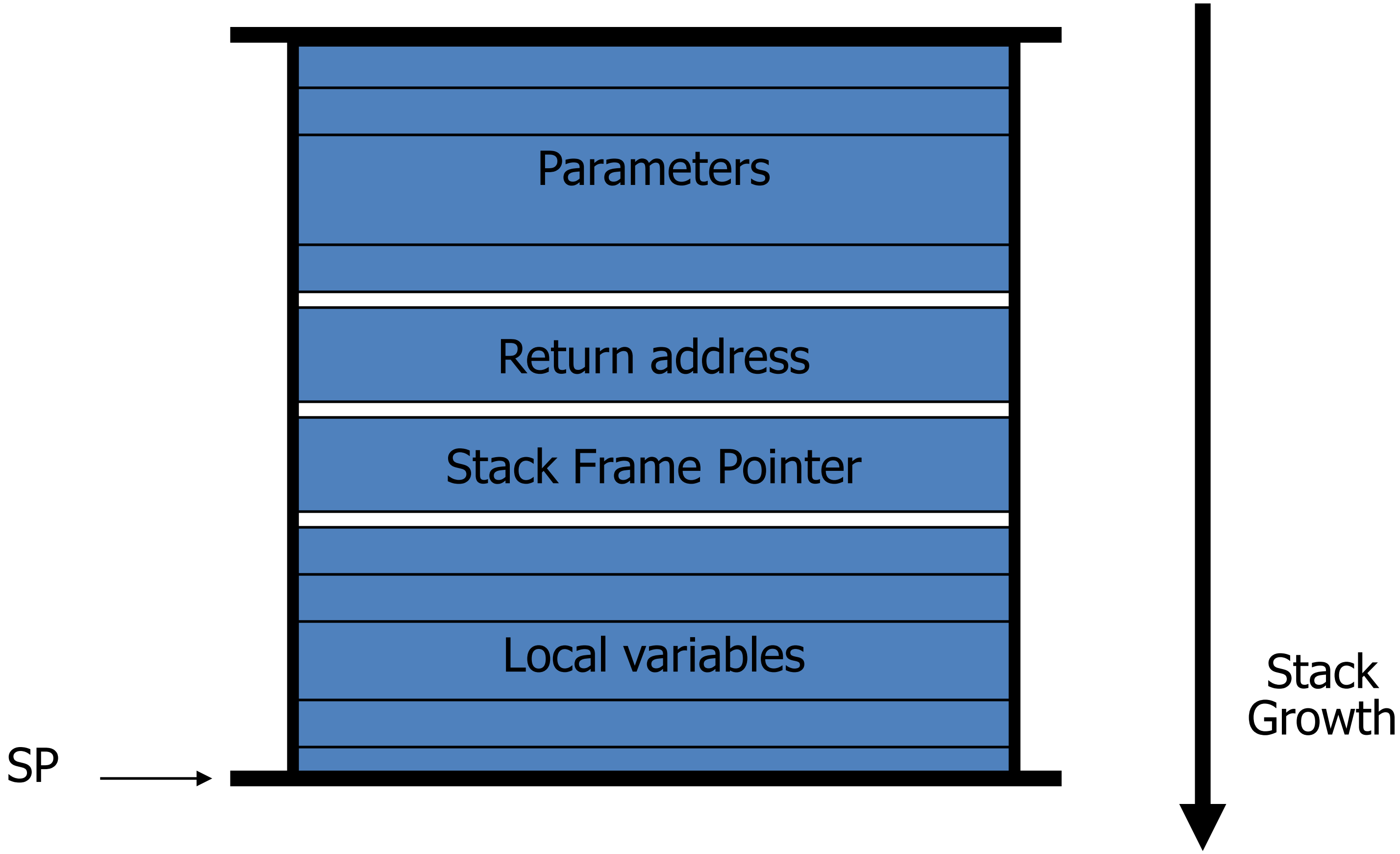
The C standard library, or libc

- Most programs use common external libraries, and first among them, **the C standard library, or libc**. This library defines such basic functions as “memcmp”, “memcpy” and “printf”, the **building blocks** from which all other code is made.
- If we can overwrite return addresses to point elsewhere in the code, and all code includes these building blocks, then we can overwrite the return address

Linux process memory layout



Stack Frame


Parameters
Return address
Stack Frame Pointer
Local variables
SP
Stack Growth

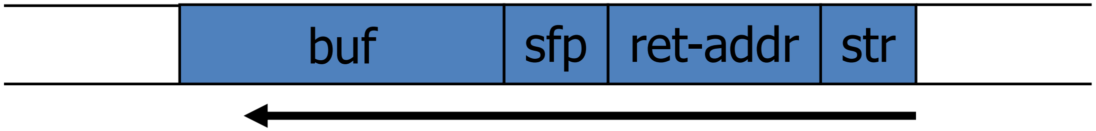
What are buffer overflows?

- Suppose a web server contains a function:

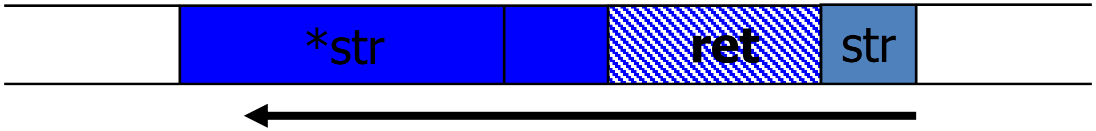
```
void func(char *str) {
   char buf[128];

   strcpy(buf, str);
   do-something(buf);
}
```

- When the function is invoked the stack looks like:

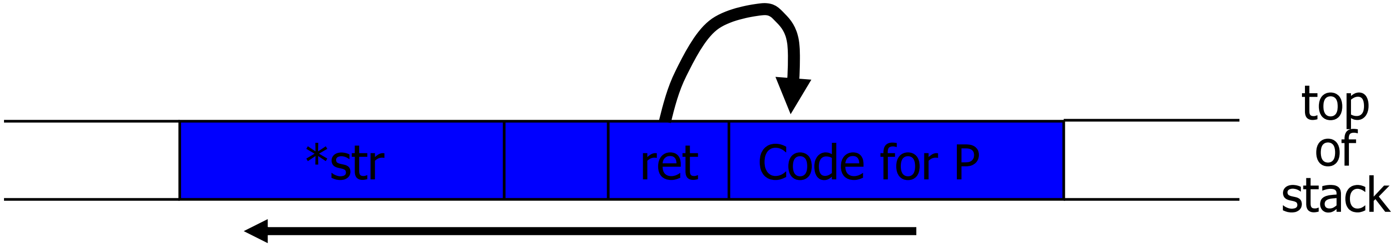


- What if **`*str`** is 136 bytes long? After **`strcpy`**:



Basic stack exploit

- Main problem: no range checking in strcpy().
- Suppose *str is such that after strcpy stack looks like:



Program P: **`exec( "/bin/sh" )`**

- When func() exits, the user will be given a shell !!
- Note: **attack code runs *in stack***.

Some **unsafe C lib** functions

strcpy (char *dest, const char *src)

strcat (char *dest, const char *src)

gets (char *s)

scanf (const char *format, …)

sprintf (conts char *format, …)

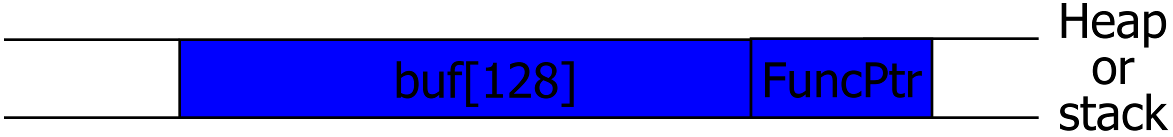
⋮

Exploiting buffer overflows

- Suppose web server calls func() with given URL.
- Attacker can create a 200 byte URL to obtain shell on web server.
- Some complications for stack overflows:
  - Program P should not contain the '\0' character.
  - Overflow should not crash program before func() exits.

Other control hijacking opportunities

- Stack smashing attack:
  - Override return address in stack activation record by overflowing a local buffer variable.
- Function pointers: (used in attack on PHP 4.0.2)

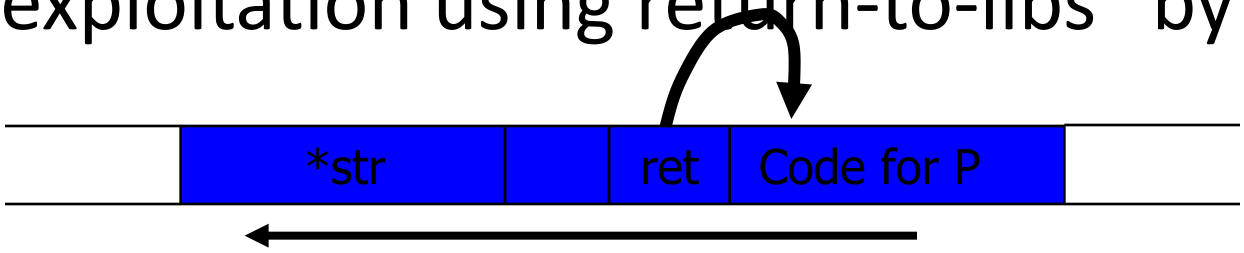


  - Overflowing buf will override function pointer.

- Longjmp buffers: longjmp(pos) (used in attack on Perl 5.003)
  - Overflowing buf next to pos overrides value of pos.

return-to-libc attack

- “Bypassing non-executable-stack during exploitation using return-to-libs” by c0ntex

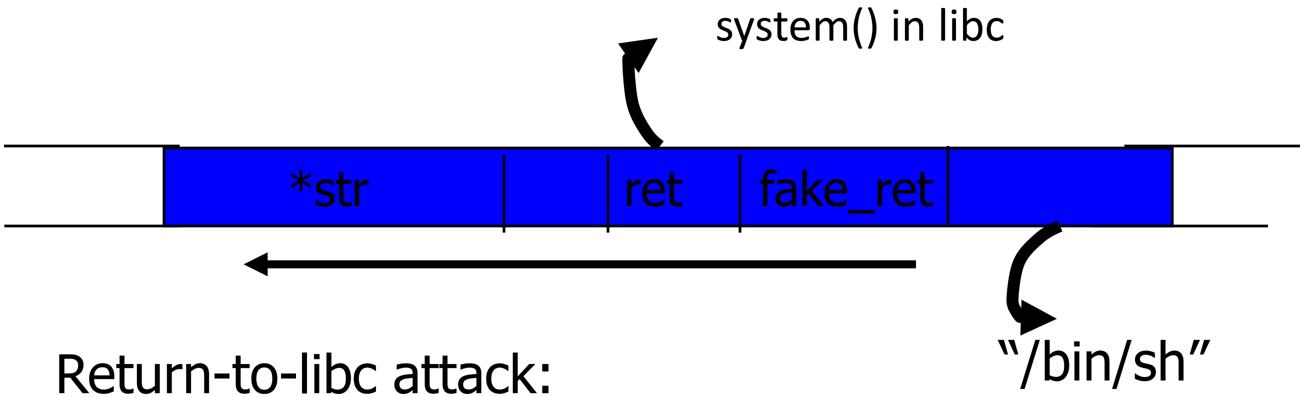


*str

ret

Code for P

Shell code attack: Program P: `exec( “/bin/sh” )`



system() in libc

*str

ret

fake_ret

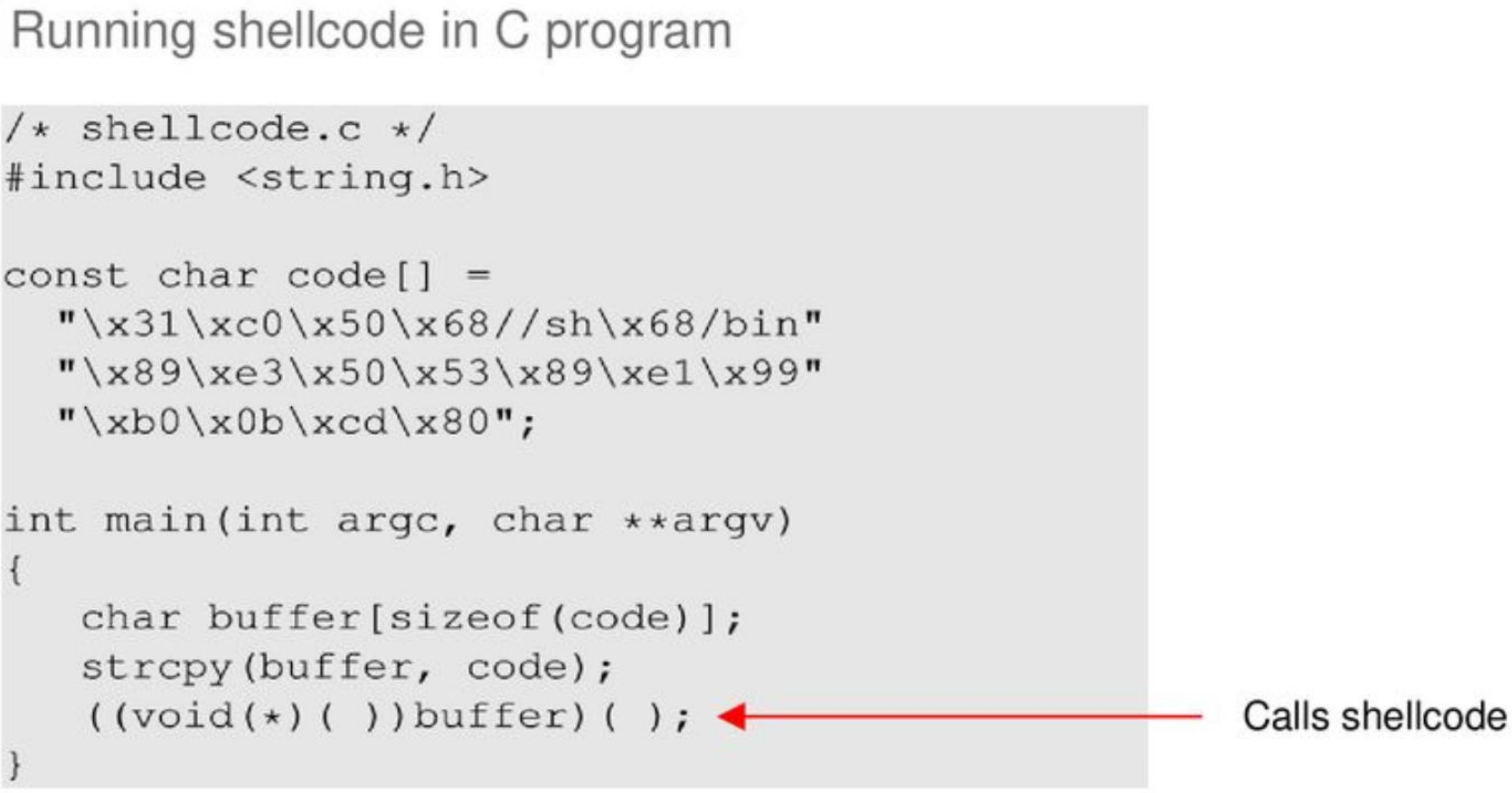
“/bin/sh”

Return-to-libc attack:

What is a Shell code ?

- Shellcode is a special type of code injected remotely which hackers use to exploit a variety of software vulnerabilities. It is so named because it typically spawns a **command shell** from which attackers can take control of the affected system.
- **Shellcode is a set of instructions that executes a command in software to take control of or exploit a compromised machine**

Running Shellcode in C



Running shellcode in C program

```
/* shellcode.c */
#include <string.h>

const char code[] =
  "\x31\xc0\x50\x68//sh\x68/bin"
  "\x89\xe3\x50\x53\x89\xe1\x99"
  "\xb0\x0b\xcd\x80";

int main(int argc, char **argv)
{
   char buffer[sizeof(code)];
   strcpy(buffer, code);
   ((void(*)( ))buffer)( );
}
```

Calls shellcode

What is Command Shell ?

- The shell is the Linux command line interpreter. It provides an interface between the user and the kernel and executes programs called commands.
- For example, if a user enters *ls* then the shell executes the *ls* command. The shell can also execute other programs such as applications, scripts, and user programs (e.g., written in c or the shell programming language).