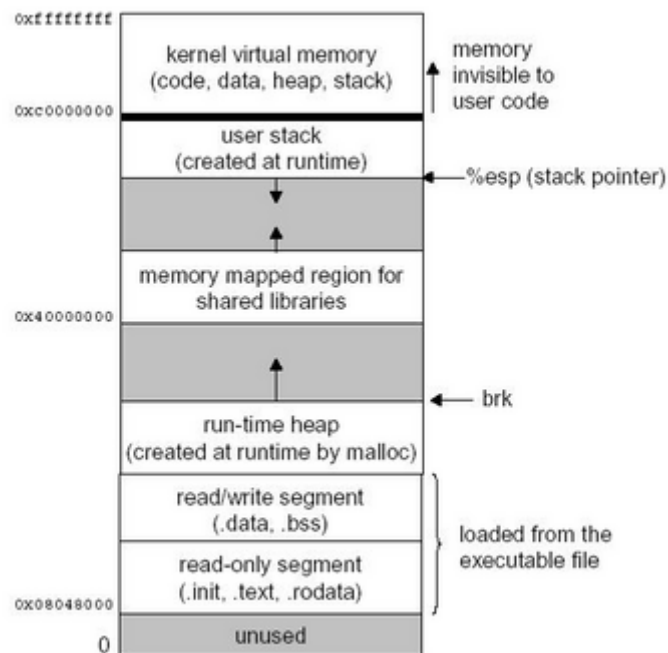


Buffer Overflow

(1) Stack Buffer Overflow

❖ Process Layout

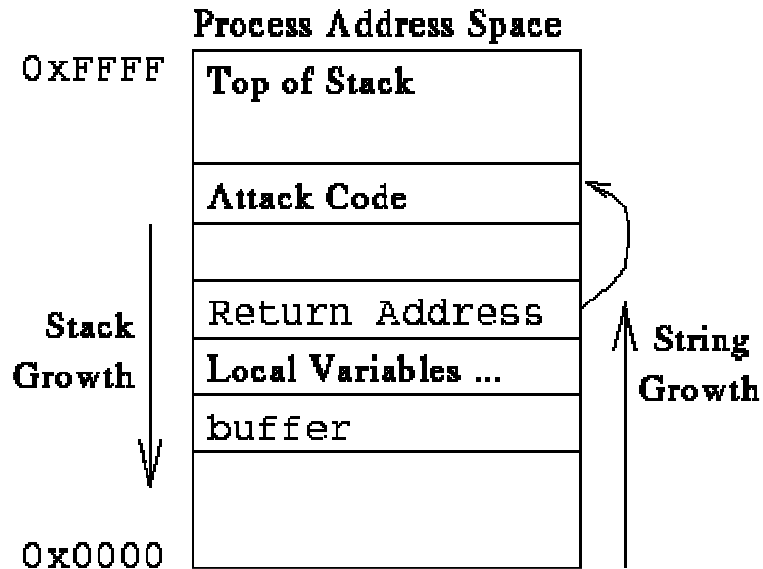
- The figure below shows the memory layout of a Linux process.
- Code and data consists of the program's instructions and the initialized and uninitialized static and global data, respectively.
- Run-time heap is mainly used for dynamically memory allocation (e.g., using malloc/calloc).
- This stack is used whenever a function call is made.



❖ Stack Layout

- Grows from high address to low address (while buffer grows from low address to high address)
- Return address: address to be executed after the function returns
- Frame pointer (FP): is used to reference the local variables and the function parameters

❖ Buffer Overflow Attack



- ❖ How to conduct a successful buffer overflow attack?
 - Injecting the attack code
 - Change the flow of execution
- ❖ Change the the flow of execution (A simple example)
 - We want to skip past the assignment to the `printf` call.
 - `buffer1 + 12`: will point to the return address (12 = 8 + 4, 4 is the size of FP)
 - `(*ret) += 8`: 8 is the distance that needs to be skipped (obtained using a debugging tool).

```

void function(int a, int b, int c) {
    char buffer1[5];
    char buffer2[10];
    int *ret;

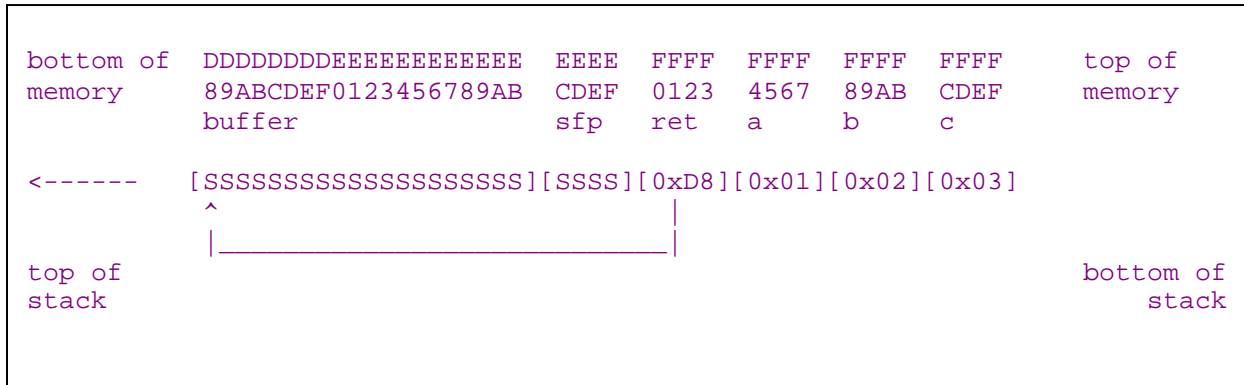
    ret = buffer1 + 12;
    (*ret) += 8;
}

void main() {
    int x;

    x = 0;
    function(1,2,3);
    x = 1;
    printf("%d\n",x);
}

```

- ❖ Cause the program to run an arbitrary code.
 - Place the code in the buffer by overflowing a buffer.
 - Overwrite the return address so it points back into the buffer.
 - The following example: Assuming the stack starts at address 0xFF, and that S stands for the code we want to execute the stack would then look like this:



- ❖ What code to inject?
 - In most cases we'll simply want the program to spawn a shell. From the shell we can then issue other commands as we wish.
 - Shell code (in Alepha One's example, the size of the code is only 45 bytes).

```

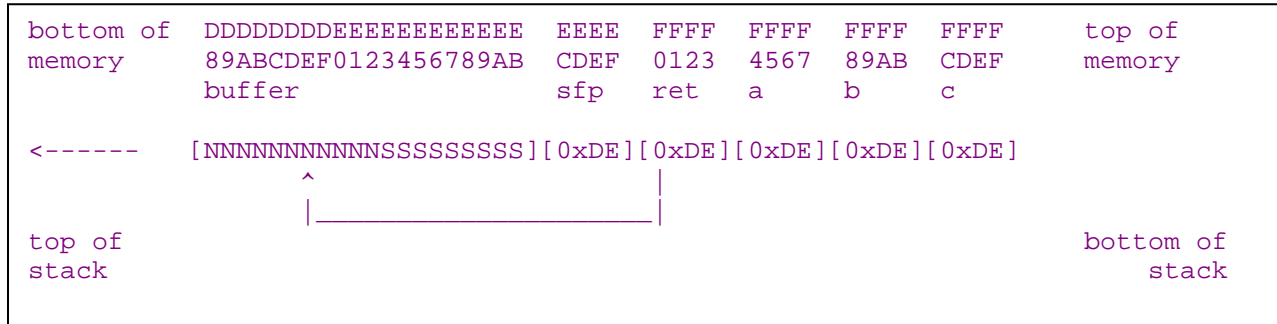
#include <stdio.h>

void main() {
    char *name[2];

    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
    
```

- ❖ Difficulty of writing a shell code
 - The address of the string "/bin/sh" is needed by the execve system call.
 - How do we know this address?
- ❖ Where is the shell code? How to jump to it?
 - We need to know the absolute address of the shell code. How to find it?
 - Guess
 - Stack usually starts at the same address.

- Stack is usually not very deep: most programs do not push more than a few hundred or a few thousand bytes into the stack at any one time.
- Therefore by knowing where the stack starts we can try to guess where the buffer we are trying to overflow will be.
- Using NOP to increase the chance.

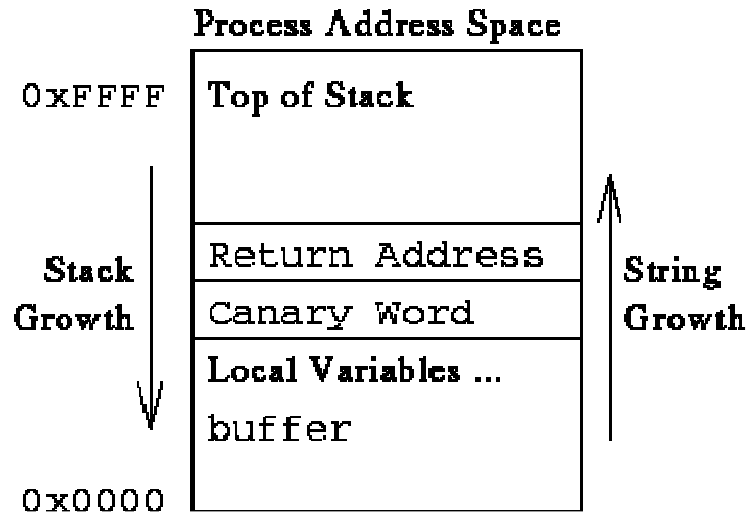


- ❖ Programs that could have buffer overflow problem
 - gets, strcpy, strcat, sprintf, scanf
 - These are safer: fgets, strncpy, strncat, snprintf.
- ❖ How to prevent buffer overflow?

In-Class Discussion

- 1) Let the stack grow from low address to high address, so the return address cannot be overwritten.
- 2) Make the stack non-executable.
- 3) Save the return address to another stack.
- 4) Use hardware protection to protect return addresses.

- ❖ StackGuard
 - Observation: one needs to overwrite the memory before the return address in order to overwrite the return address. In other words, it is difficult for attackers to only modify the return address without overwriting the stack memory in front of the return address.
 - A canary word is placed next to the return address whenever a function is called.
 - If the canary word has been altered when the function returns, then some attempt has been made on the overflow buffers.



- ❖ Stack Shield
 - Copy the RET address in an unoverflowable location (the beginning of the DATA segment) on function prologs (on function beginnings)
 - Check if the two values are different on function epilogs (before the function returns).
 - Need to maintain a stack kind of structure for storing return addresses.

- ❖ Non-executable Stack
 - There are a few occasions that require the stack to be executable: e.g. Linux signal handler.
 - Non-executable stack is now implemented in Linux kernel.
 - **Return-to-libc** attack: alter the return address, but the control is directed to a C library function rather than to a shellcode. For example, `system()` is a library function. It can be used to execute an arbitrary program (e.g. `system("/bin/sh")`). In most of the systems, the location of the C library is fixed, which makes it easy to guess. The parameters of the library function must be put in the stack, i.e., when overwriting the buffer, we should construct the stack frame for the library function. Library functions are NOT stored in the stack.

- ❖ Randomization Approach
 - For the Return-to-libc attack: randomize the library location and makes it difficult to guess.

(2) Heap Buffer Overflow

- ❖ Introduction of Heap/BSS
 - Global variables
 - Static variables.
 - Dynamic allocated memory.
- ❖ Overwriting file pointers
 - The (set-uid) program's file pointer points to "/tmp/vulprog.tmp".
 - The program needs to write to this file during execution using the user's inputs.
 - If we can cause the file point to point to "/etc/shadow", we can add a line to "/etc/shadow".
 - How to find the address of "/etc/shadow"?
 - Pass the string as the argument to the vulprog program, this way the string "/etc/shadow" is stored in the memory. We now need to guess where it is.

```

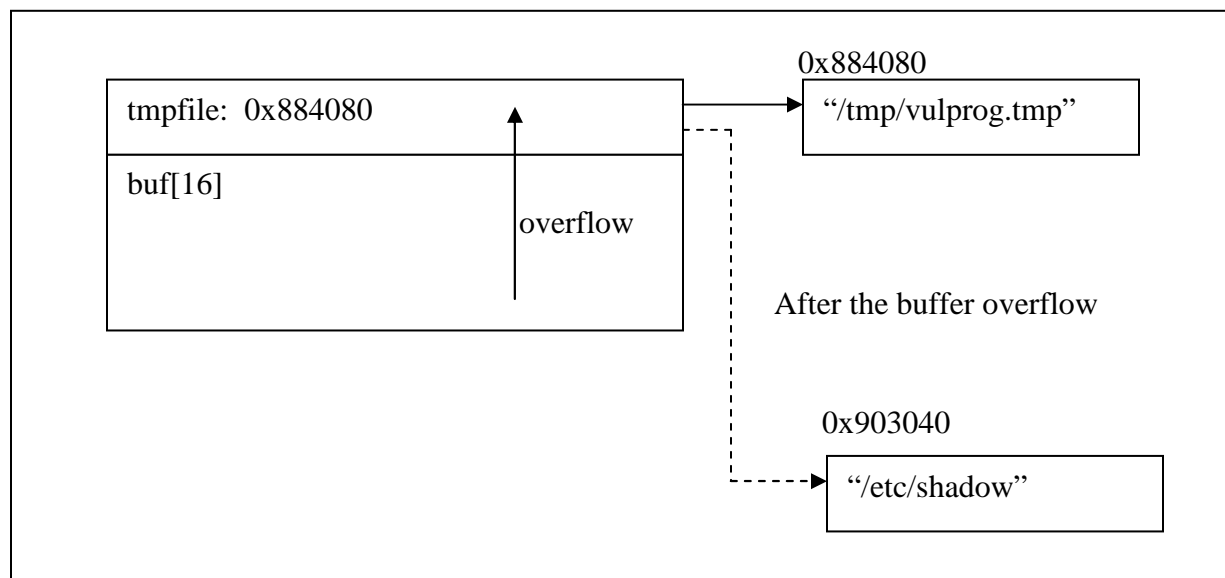
static char buf[BUFSIZE], *tmpfile; /* saved in the heap */

/* argv[1] is provided by the user */
tmpfile = "/tmp/vulprog.tmp";

gets(buf); /* let's cause a buffer overflow here to change
           the value of tmpfile. */

..... Write to tmpfile .....

```



❖ Function Pointer

- A function pointer (i.e., "int (*funcptr)(char *str)") allows a programmer to dynamically modify a function to be called. We can overwrite a function pointer by overwriting its address, so that when it's executed, it calls the function we point it to instead.

```
int main(int argc, char **argv)
{
    static char buf[BUFSIZE]; /* in heap */
    static int (*funcptr)(const char *str); /* in heap */

    ...

    funcptr = (int (*)(const char *str))goodfunc;

    strncpy(buf, argv[1], strlen(argv[1])); /* We can cause buffer overflow
                                             here */

    (void)(*funcptr)(argv[2]);
    return 0;
}

/* This is what funcptr would point to if we didn't overflow it */
int goodfunc(const char *str)
{
    .....
}
```

❖ Overwriting Function Points

- argv[] method: store the shellcode in an argument to the program. This causes the shellcode to be stored in the stack. Then we need to guess the address of the shellcode (just like what we did in the stack-buffer overflow). This method requires an executable stack.
- Heap method: store the shellcode in the heap (by using the overflow). Then we need to guess the address of the shellcode, and assign this estimated address to the function pointer. This method requires an executable heap (which is more likely than an executable stack).

❖ A case study

- The BSDI crontab heap-based overflow: Passing a long filename will overflow a static buffer. Above that buffer in memory, we have a pwd structure! This stores a user name, password, uid, gid, etc. By overwriting the uid/gid field of the pwd, we can modify the privileges that crond will run our crontab with (as soon as it tries to run our crontab). This script could then put out a suid root shell, because our script will be running with uid/gid 0.