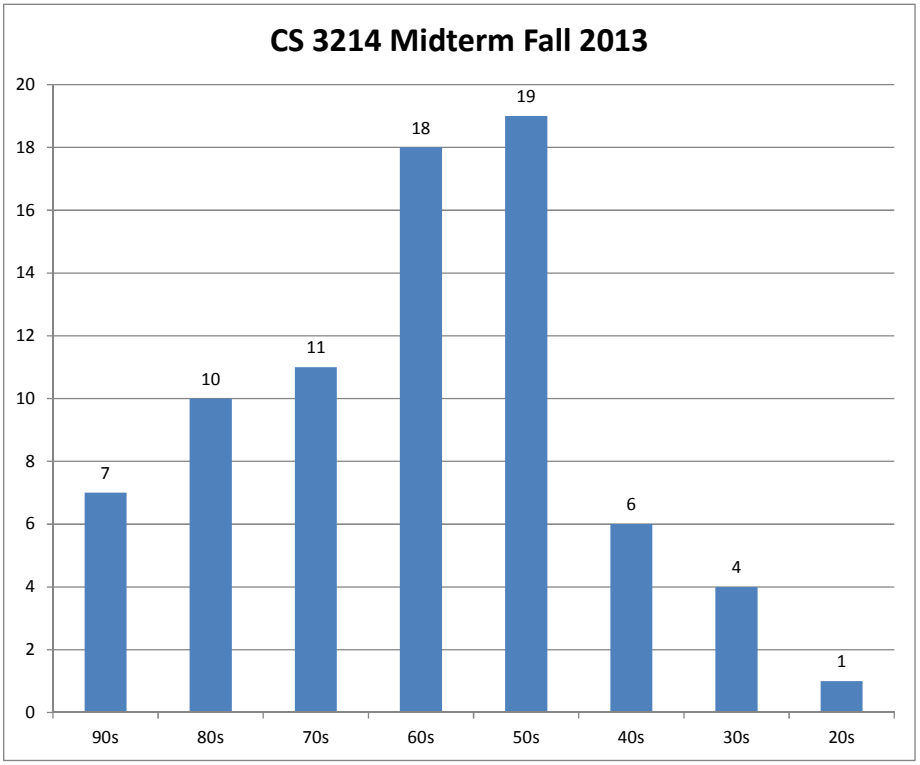
CS 3214 Midterm



The chart shows the distribution of midterm scores across both sections.

The average score was 63.1 and the median score was 64.

CS 3214 Midterm

This is a closed-book, closed-internet, closed-cell phone and closed-computer exam. However, you may refer to your sheet of prepared notes. Your exam should have 12 pages with 3 topics totaling 100 points. You have 75 minutes. Please write your answers in the space provided on the exam paper. If you unstaple your exam, please put your initials on all pages. You may use the back of pages if necessary, but please indicate if you do so we know where to look for your solution. You may ask us for additional pages of scratch paper. You must submit all sheets you use with your exam. However, we will not grade what you scribble on your scratch paper unless you indicate you want us to do so. Answers will be graded on correctness and clarity. The space in which to write answers to the questions is kept purposefully tight, requiring you to be concise. You will lose points if your solution is more complicated than necessary or if you provide extraneous, but incorrect information along with a correct solution.

Name (printed) _____

I accept the letter and the spirit of the Virginia Tech undergraduate honor code – I will not give and have not received aid on this exam.

(signed) _____

#	Problem	Points	Score
I	Programs and Data	26	
II	Processes and Signals	50	
III	Threads and Synchronization	24	
	Total	100	

I. Programs and Data (26 points)

1. (16 points) Consider your experiences in solving the Firecracker phase of the Buffer Bomb. The two vital functions, getbuf() and bang() are shown below:

```
/* Buffer size for getbuf */
#define NORMAL_BUFFER_SIZE 32

int getbuf() ~~$~~ {
   char buf[NORMAL_BUFFER_SIZE];
   Gets(buf);
   return 1;
}

int global_value = 0;

void bang(int val) {
   if (global_value == cookie) {
      printf("Bang!: You set global_value to 0x%x\n",
             global_value);
      validate(2);
   } else
      printf("Misfire: global_value = 0x%x\n",
             global_value);

   exit(0);
}
```

In order to solve the phase, it is necessary to do two things:

- Put the team cookie into the variable global_value.
- Cause the function bang() to be executed.

When analyzing the situation, the student conducts a gdb session, shown below:

```
GNU gdb (GDB) Red Hat Enterprise Linux (7.2-60.el6)
. . .
Reading symbols from /home/sheldon/buflab/bufbomb...

(gdb) break getbuf
Breakpoint 1 at 0x8048c0a
(gdb) break Gets
Breakpoint 2 at 0x8048b50

(gdb) run -u sheldon
```

```
Starting program: /home/sheldon/buflab/bufbomb -u
sheldon
Userid: sheldon
Cookie: 0x28650607
Breakpoint 1, 0x08048c0a in getbuf ()

(gdb) print /a $ebp+4
$1 = 0x556834a4 <_reserved+1037476>          #### A
(gdb) continue
Continuing.

(gdb) backtrace                              #### C
#0  ... in Gets ()
#1  ... in getbuf ()
#2  ... in test ()
. . .

Breakpoint 2, 0x08048b50 in Gets ()
(gdb) print /a *(void**)($ebp+8)
$2 = 0x55683478 <_reserved+1037432>          #### B

(gdb) print /a (void*)bang
$3 = 0x8049012 <bang>

(gdb) print /a (void*)&global_value
$4 = 0x804c1ec <global_value>
(gdb)
```

(a) (3 points) Why is the value displayed by the `print` output labeled A relevant?

$epb + 4 points to (not IS) the return-to address that will be used when getbuf() returns.

We need to replace that with the entry point for the exploit, and then with the entry point for the function bang().

(b) (3 points) Why is the value displayed by the `print` output labeled B relevant?

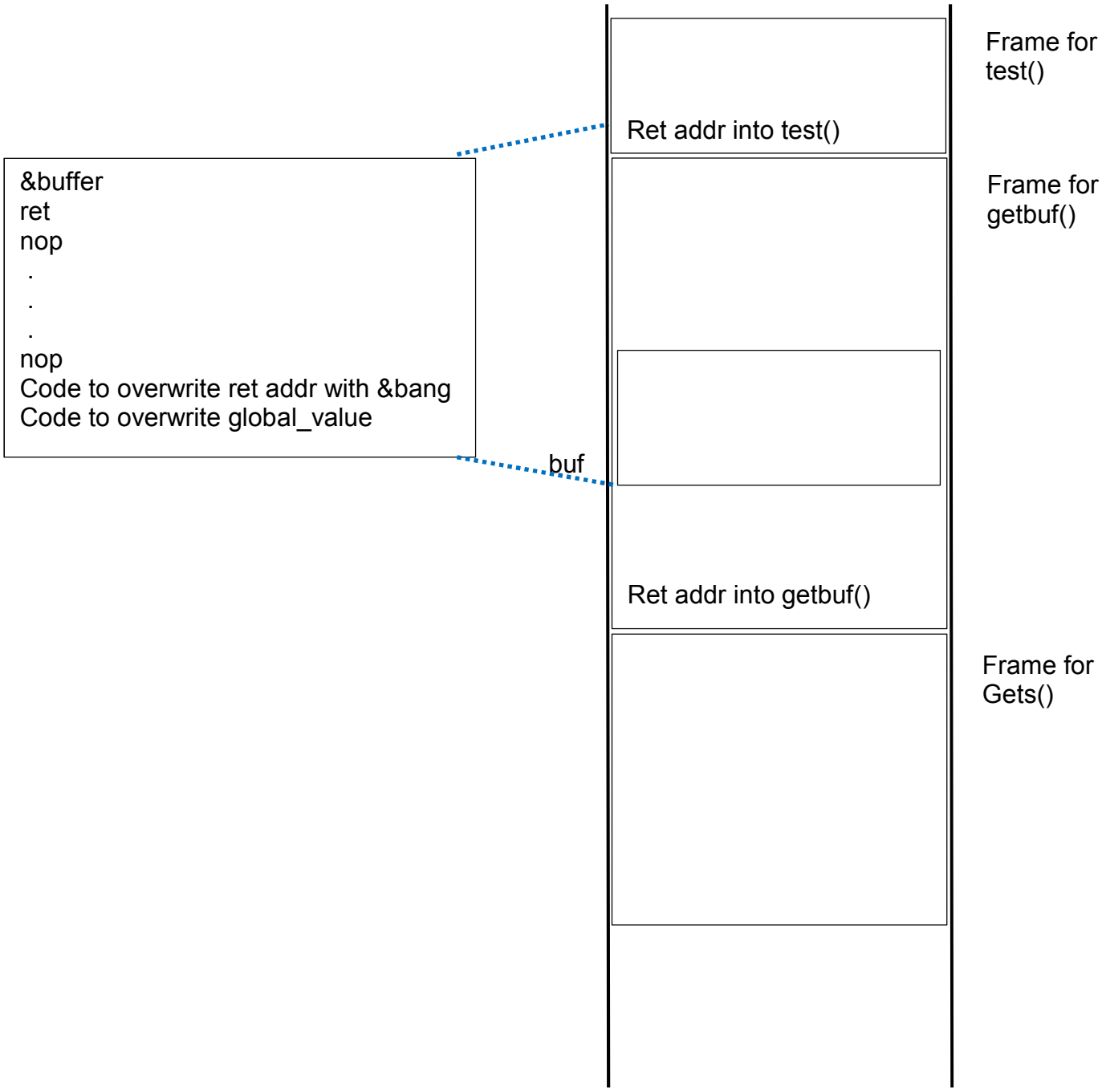
$epb + 8 points to the first parameter to Gets(), which is the address of the buffer to which Gets() will write. Since we dereference in this case, the value printed is actually the parameter, which is the address of the buffer.

We need to:

- **Write our exploit into the stack frame for getbuf(), starting at this address**

- **Calculate the "distance" between the beginning of that buffer and the return-to address from the previous question, so that we know how much data we need to send Gets() in order to overwrite the return-to address for getbuf() with the entry point of the exploit.**

Draw a diagram of the state of the relevant parts of the stack at the point of the backtrace labeled C in the gdb session. Annotate your stack diagram to explain your answers to the following two questions.



Note: the answers to the two parts are intertwined, so I'm showing a unified solution. That's how your answers were evaluated as well.

(c) (5 points) How can the student achieve the goal of placing his cookie into `global_variable`? Be specific. A fully-satisfactory answer will refer to the stack diagram you drew above, and include addresses and code.

(d) (5 points) How can the student achieve the goal of triggering a call to `bang()`? Be specific and complete. A fully-satisfactory answer will refer to the stack diagram you drew above and include addresses; code is not necessary.

There are a few things to note. We cannot, by overflowing the buffer, alter the fact that Gets() returns to the call made from getbuf().

We cannot alter the execution of getbuf() prior to the point that getbuf() returns.

What we can do is this:

- **cause getbuf() to return to somewhere else, like the beginning of our exploit**
- **cause our exploit to do whatever we want as long as it fits within the available space**
- **cause our exploit to "return" to wherever we want, specifically bang()**

The student needs to:

- **replace the return-to address of getbuf() with the address of the buffer, in order to execute the exploit; therefore the exploit needs to be exactly the correct length (hence padded with something sensible) and to end with the address of buf, overwriting the old return address from getbuf() into test()**
- **during the execution of the exploit, overwrite the value at the address of global_value with the cookie**
- **during the execution of the exploit, replace the return-to address of getbuf() with the address of bang()**
- **execute a ret from the exploit, so as to trigger the execution of bang()**

This all needs to be described clearly and completely, so that it's clear what must be placed where, and what the order of execution will be.

2. (10 points) In the following code

```
...
static int f();
static int g();

int z() {

   int a = f();
   int b = g();
   return ( a + b + f() );
}
```

Why cannot a C compiler necessarily safely optimize `z()` by making the following change:

```
return ( 2*a + b );
```

Answer: because the compiler cannot guarantee that successive calls to the function f() return the same result. For example, f() might use a static variable or f() might refer to a global variable that's modified by g().

II. Processes and Signals (60 points)

3. (10 points) Mark the appropriate box next to each statement to indicate whether that statement is true or false.

Statement	True	False
A context switch happens on every system call.		X
A context switch implies that a mode switch has also occurred.	X	
A mode switch causes a different process to execute.		X
Some system calls cause a mode switch but do not necessarily cause a context switch.	X	
Some system calls cause a context switch but do not necessarily cause a mode switch.		X
Sending a signal blocks the sender if the receiving process has the signal blocked.		X
The signal handler for a process executes in user mode and not in kernel mode.	X	
File descriptors of a process remain unchanged across a fork system call but not across an exec system call.		X
Using the signal or sigaction system call to install a signal handler blocks until a signal arrives.		X
Read/write operations on a pipe do not require any synchronization to be correct.	X	

Answer: shown in table above.
Grading: 1 point per correct answer.

4. (15 points) When execution has reached the comment “HERE” in the following code say how many processes are running and what code they are executing. The files progA, progB, and progC are user-defined executable programs in the current path.

```
main() {
int index = 0;
…
int p = fork();
if (p==0 && index ==0)
   { index++; exec(“progA”…);}
if (p!=0 && index==0)
   { index++; p = fork(); exec(“progB”…);}
if (p==0 && index>0)
   { index++; exec(“progC”…); }
// HERE
}
```

Answer: three process are executing, one is executing progA and two are executing progB

Grading: 6 points for answering there are 3 processes
3 points each for giving the correct code that is executed by each process

5. (15 points) In a typical shell (like esh in project 2) the command to be executed is done by a child process while the parent process continues executing the shell code. However, the parent and child processes are essentially identical. Could you design the shell so that the parent process performs the command to be executed while the child process continues executing the shell code? Explain.

Answer: This could not be done because there would be no way for the shell (the child process) to properly wait for the completion of the command (the parent process).

Grading: 15 points for a correct answer and reason.
10 points for a correct answer and a reason that is incorrect though plausibly related (e.g., terminal control, processes groups)
7 points for a incorrect answer and a reason that is incorrect though plausibly related (e.g., terminal control, processes groups)
5 points for making correct statements about process concepts that have plausible connection to the question

6. (10 points) Write the code for a parent process that will create and wait for a child process. The created child process should execute the program in the file “progX”, read its standard input from the file “myinput” and write its standard output to “myoutput”. For the purposes of this question the otherwise unacceptable practice of ignoring error conditions is allowed.

Answer:

```
int child = fork();
if (child==0) {
   int fd1 = open(myinput, O_RDONLY);
   int fd2 = open(myoutput, O_WRONLY);
   dup2(fd1,0);
   dup2(fd2, 1);
   close(fd1);
   close(fd2);
   exec(“progX”,…);
}
wait(child);
```

Grading:

2 points for correct fork and test for child process
2 points for correct wait by parent
1 point for opening both files
2 points for correct use of dup2 system call
1 point for closing file descriptors fd1 and fd2
2 points for correct exec call (ignore flavor of exec call used and other parameters to exec.

III. Threads and Synchronization (24 points)

7. (6 points) Consider the following declarations and code.

```
int x[20];
int y[20];

void addTo(int* vec, int start, int stop) {
  for(int i = start; i<stop; i++)
    vec[i] = vec[i] + 1;
}
```

The table below shows three cases of calls made by two threads. For each case determine if there is a possible race condition and write Yes or No in the right column.

Thread 1	Thread 2	Race Condition (yes/no)
`addTo(x,5,10);`	`addTo(y,7,10);`	NO
`addTo(x,5,10);`	`addTo(x,7,10);`	YES
`addTo(y,5,10);`	`addTo(y,12,20);`	NO

Grading: 2 points for each correct answer

8. (6 points) Show the pthread synchronization needed in the code below to insure that the file is not used until after the file is opened. Be sure to show the declaration and initialization of any additional variables needed.

```
void fun(int i) {
  ....
  int fd = open("testdata",...);
  ....
}

void games(int i) {
   ....
   read(fd,...);
   ....
}
```

Answer:

```
pthread_mutex_t  file_opened = PTHREAD_MUTEX_INITIALIZER;

main() {
   pthread_mutex_lock(&file_opened);
   …
}

void fun(int i) {
  ....
  int fd = open("testdata",...);
  pthread_mutex_unlock(&file_opened);
}

void games(int i) {

   pthread_mutex_lock(&file_opened);
   read(fd,...);
   ....
}
```

Grading:
- 1 point for declaration of mutex variable
- 1 point for initialization of mutex variable
- 2 point for indicating that the variable must be locked initially (any way they have of showing this is fine)
- 1 point for use of unlock after file opened
- 1 point for use of lock operation before file accessed

9. (12 points) Show the pthread synchronization needed so that the function below can be executed safely by multiple threads. Answers will be graded on whether the correct synchronization has been added and whether the synchronization is added in the correct places. Be sure to show the declaration and initialization of any variables needed for the synchronization.

```
struct {            // global shared variable
  int cumulative;
  int values;
} val;

pthread_mutex_t  mutex = PTHREAD_MUTEX_INITIALIZER;

int func(int x, int y) {
{
    int increment, myvalues;
    increment = (x+y)**2 + (x-y)**2;
    myvalues = x+y;

    if (increment > 100) {
       pthread_mutex_lock(&mutex);
       val.cumulative = val.cumulative + increment;
       val.values = x+y;
       pthread_mutex_unlock(&mutex);
    }
    else {
       pthread_mutex_lock(&mutex);
       val.cumulative = x;
       val.number = y;
       pthread_mutex_unlock(&mutex);
    }

    int result = x*y;

    return result;
}
```

Answer: shown above in bold.

Grading: 2 points for declaration and initialization of global mutex variable
5 points for lock/unlock in then clause
5 points for lock/unlock in else clause

deduct 2 points if entire if-then-else is treated as one critical section
5 points if initial assignments to increment and myvalues are in the critical section