1 Stack smashing

(5 points) Draw a picture of the stack during the execution of the following function. You may use the simplified stack layout we have used in class.

```c
char * lookup_password(char *username)
{
    char buf[32];
    int i;

    strcpy(buf, username);

    for (i = 0; i < nusers; i++)
        if (strcmp(buf, user[i].name) == 0)
            return user[i].password;
    return NULL;
}
```

**Solution:**

(5 points) Provide a “username” that will cause this function to execute arbitrary code of your choice. You do not need to specify your shell code in full detail – you can just say something like “<shell code (95 bytes)>”. Your input string must correspond to the diagram given above, though.

**Solution:**

\[
< \text{ShellCode(32bytes)} > < 0 \times abcdef0104 > \tag{1}
\]
2 SQL injection and prepared statements

(5 points) Prepared statements allow programmers to write secure code, but do not force them to write secure code. For example, consider the following simplified prepared statement API:

```java
void executePreparedStmt(String template, ...);
```

A prepared statement template may have placeholders $1, $2, etc., and a programmer can execute a prepared statement by doing, e.g.

```java
executePreparedStmt("SELECT * FROM users WHERE name = $1 AND passwd = $2", username, pwd);
```

Write a simple function (at most 5 lines of code) that uses the above prepared statement API but is still vulnerable to SQL injection attacks.

Solution:

```java
executePreparedStatement("SELECT * FROM users WHERE name =" + name "AND passwd =" + pwd);
```

(5 points) Give a simple and easy-to-check rule that would prevent this kind of programming error.

Solution: The template must always be a compile time constant.

Most Common Incorrect Answer: Perform checks on the inputs to verify the existence of SQL identifiers, etc.
3 Model checking

(1) Programmers must disable signals before entering privileged mode. Assume the following functions: `enable_signals()`, `disable_signals()`, `setuid()`. Solution:

(2) Programmers should not call `access(fname)` and `open(fname)` on the same filename. Solution:

(3) Whenever a web browser loads a new web page, it must update the location bar to match. Assume the browser has functions `beginTxn()`, `endTxn()`, `loadPage(String url)` and `setLocation(String url)`. Solution:
4 Run-time taint tracking

(5 points) CQual allowed programmers to cast away taintedness of a variable. Why is it not necessary to give programmers the ability to mark variables as untainted with run-time taint-tracking?

**Solution:** A policy can be written to consider the presence of tainted bytes. This makes it unnecessary to give the programmer control of marking variables as untainted.

(5 points) Specify, in a single English sentence or as a taint-annotated regular expression, a run-time taint tracking policy for format string bugs.

**Solution:** No percent character may be marked tainted unless it is immediately followed by another tainted percent character.
(10 points) The baggy bounds checking paper does not specify if or how it might prevent format string bugs. Assuming that both your program and the printf functions are transformed by your transformation, show how you can prevent format string bugs using baggy bounds checking. You do not need to explain how to transform the printf function, just show how you would transform the following code:

```c
void foo(int x, int *p, char *s)
{
    printf("%d\n", x, p + x);
    printf(s);
}
```

Solution:

```c
void foo(int x, int *p, char *s)
{
    boundscheck(p + x, p);
    printf("%d\n", 2, 0, x, 1, p + x);
    \*2 is the number of optional arguments,
    0 indicates the following argument is NOT a pointer,
    1 indicates the following argument is a pointer.*
    printf(s, 0);
}
```