CSCI E-170 Lecture 09:
Attacks, Exploits, and RFID

Simson L. Garfinkel
Center for Research on Computation and Society
Harvard University

November 28, 2005

Today's Agenda

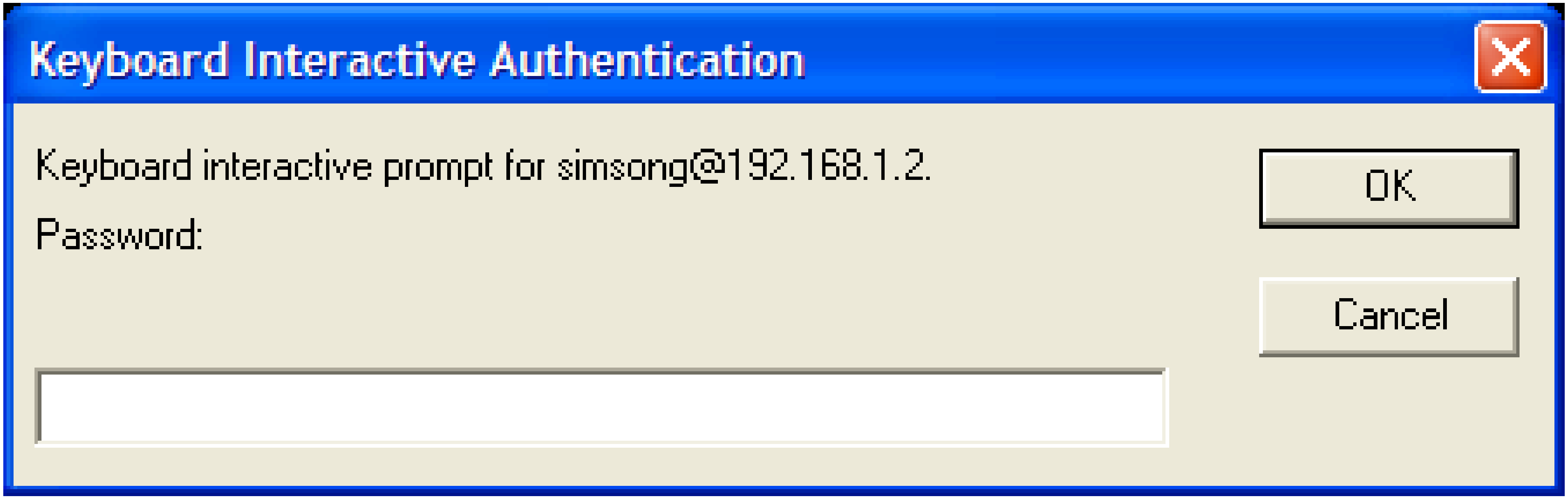
1. Administrivia
2. Hour 1: Hacking
3. Hour 2: RFID (Joe)

Administrivia

1. Midterm projects are still being graded
2. Final project groups will be assigned on November 30th
   (a) Propose projects and groups on LiveJournal!
   (b) Email us with groups to avoid assignment

An unanswered question.

One student reported that SSH didn't work properly after a client-side certificate was installed.



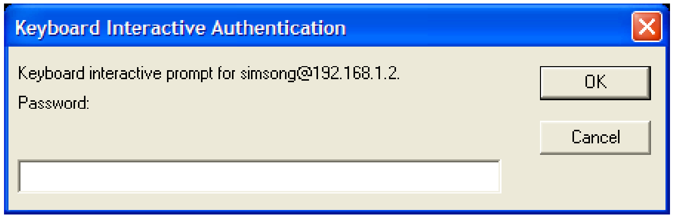
What was going on here?

Traditionally, users were instructed to “report anything out of the ordinary.”

Few attacks are perfect.

Users are more likely to encounter changes than security staff.

Example:

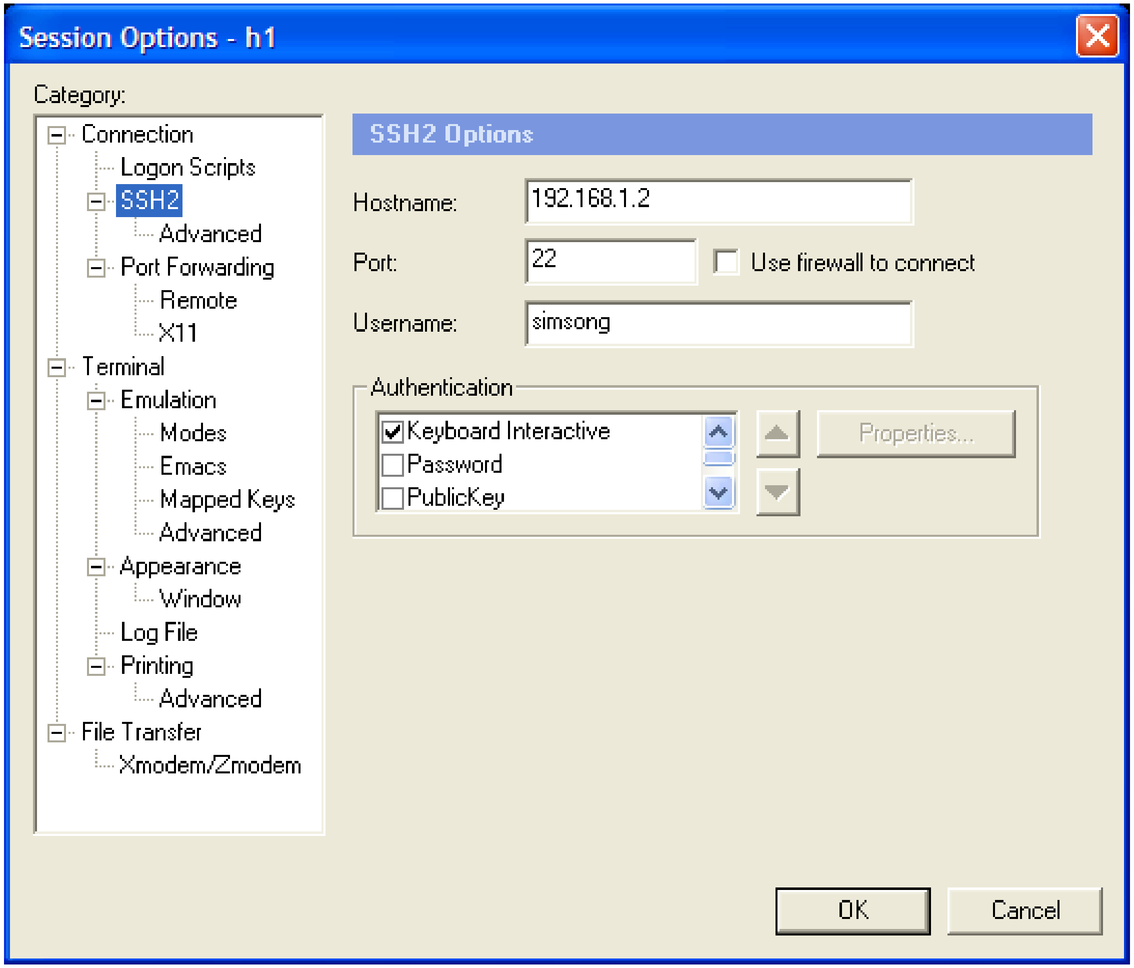


1. Appearance of new files

2. Additional authentication step

Strictly speaking, reporting “new keys” is a kind of “new behavior.”

SSH 2.0 allows two kinds of password authentication: "Keyboard Interactive" and "Password."



In this case, the new behavior was caused by an SSH configuration change.

In all likelihood, a server was upgraded and a new sshd_config file was installed:

/etc/ssh/sshd_config

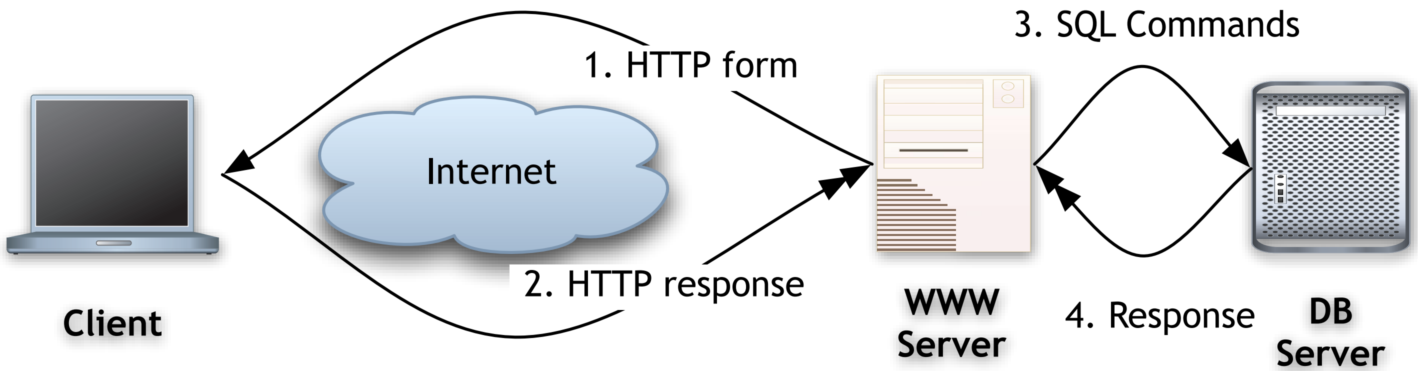
```
# Change to yes to enable built-in password authentication.
#PasswordAuthentication yes
#PermitEmptyPasswords no
```

Example of “Bystander Effect” or “false correlation.”

In the basic WWW system, programs running on the server formulate SQL queries with data provided by the user.

```
<form method='post' action='login.cgi'>
<input type='text' name='username'>
<input type='password' name='password'>
<input type='submit'>
</form>
```

```
select password
from users
where username='smith'
```




```
POST /login.cgi HTTP/1.1
Accept: */*
Accept-Language: en
Accept-Encoding: gzip, deflate
Referer: http://www.simson.net/login.cgi
User-Agent: Mozilla/5.0 (Macintosh; U; PPC Mac OS X; en)
Content-Type: application/x-www-form-urlencoded
Content-Length: 30
Connection: keep-alive
Host: www.simson.net
username=smith&password=MyPassword9
```

A login attempt with username *smith* is validated at the WWW server.

```
my $username = "smith"; my $password ='MyPass9'; # for testing

my $dbi = 'dbi:mysql:webapp:localhost';
my $dbuser = 'web';
my $dbpass = 'mypass';

my $cmd  = "select password from users where username='$username'";
my $dbh = DBI->connect($dbi,$dbuser,$dbpass);
my $sth = $dbh->prepare("select password from users " .
                        "where username='" . $user . "'");
if (!$sth->execute){
    die "SQL failure" . $sth->errstr ;
}
my @vals = $sth->fetchrow_array;
my $pass = $vals[0];

print "The password is ",$pass,"\n";
print "the passwords match\n" if($pass eq $password);
```

The username ’smith’ is used to create an SQL statement.

username = smith

```
select password from users where username='smith';
```

Inject a bit of SQL into the username...

username = xxx’ or password=’joker’;

```
select password from users
       where username='x32356xx' or password='joker';
```

This will return the password of user *x32356xx* or any user that has ’joker’ as their password.

There are two defenses against SQL injection attacks.

1. Sanitize information provided by the user.

2. Use prepared statements.

Sanitize the user's input, rather than trusting it.

Option 1: Remove objectionable characters.

Option 2: Pass allowable characters.

Option 2 is more safe, but a little more work.

User prepared statements

Instead of this:

```
my $cmd = "select password from users ".
          "where username='$username'";
my $sth = $dbh->prepare($cmd);
if (!$sth->execute()){
...
```

Do this:

```
my $cmd = "select password from users where username=?";
my $sth = $dbh->prepare($cmd);
if (!$sth->execute($username)){
...
```

Prepared statements are more efficient *and* more secure.

Minimize the damage with restricted privileges.

Instead of this:

```
grant ALL PRIVILEGES on *.* to 'web'@'localhost';
```

Use this:

```
grant SELECT on webapp.* to 'web'@'localhost';
```

Minimize damage by doing more work in the database.

Instead of doing this:

```
j = select password from database
    where username=provided_username
if j==provided_password {allow access}
```

Try this:

```
j = select username from database
    where username=provided_username and
    password=provided_password
if j!=0 {allow access}
```

This makes attacks less likely, but still possible

Sources for information on SQL injection attacks.

LAMP:

Steve Friedl, "SQL Injection Attacks by Example,"

`http://www.unixwiz.net/techtips/sql-injection.html`

Windows:

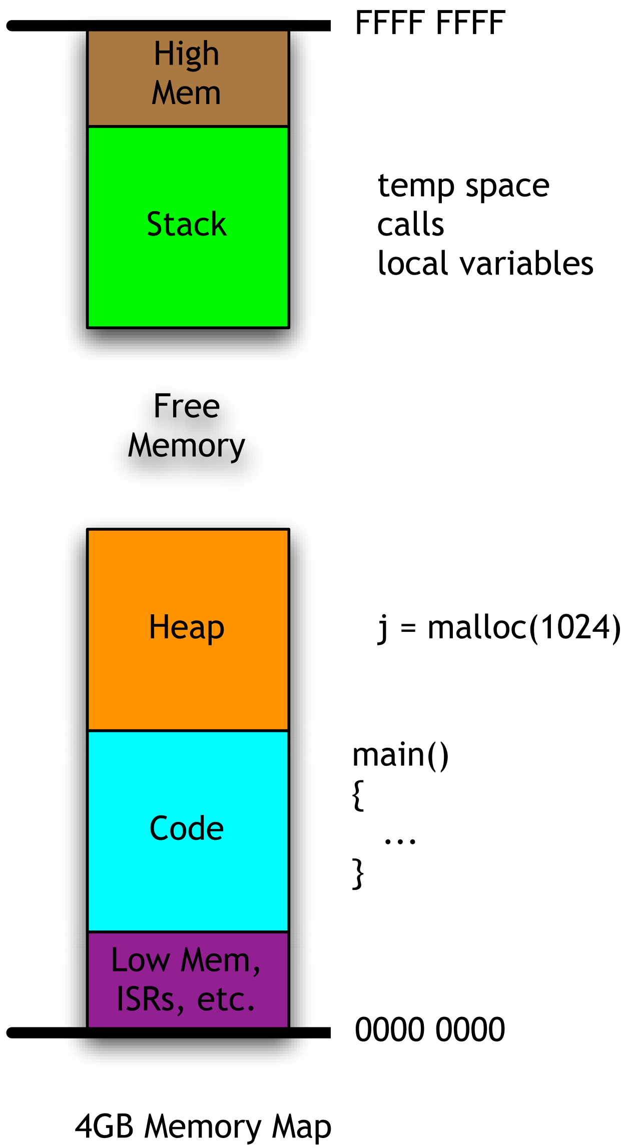
Mitchell Harper, "SQL Injection Attacks — Are You Safe?" July 17, 2002.
`http://www.sitepoint.com/article/sql-injection-attacks-safe`

Paul Litwin, "Stop SQL Injection Attacks Before They Stop You," *MSDN Magazine*, September 2004
`http://msdn.microsoft.com/msdnmag/issues/04/09/SQLInjection/`

Ross Overstreet, "Protecting Yourself from SQL Injection Attacks,"

`http://www.4guysfromrolla.com/webtech/061902-1.shtml`

Understanding buffer overflows



What happens when this code is run?

```
#include <stdio.h>
#include <unistd.h>

main(int argc,char **argv)
{
    char a;
    char buf[80];
    int  c;
    int  d;

    printf("a is at %x (%d)\n",&a,&a);
    printf("b is at %x (%d)\n",buf,buf);
    printf("c is at %x (%d)\n",&c,&c);
    printf("d is at %x (%d)\n",&d,&d);
    return 0;
}
```

Which number will be bigger, *a* or *b*?

Run the program:

```
% make
cc    -c -o stack_demo.o stack_demo.c
cc -o stack_demo stack_demo.o
%

% ./stack_demo
a is at bffff268 (-1073745304)
b is at bffff269 (-1073745303)
c is at bffff2bc (-1073745220)
d is at bffff2c0 (-1073745216)
%
```

Why are the numbers negative?

Let's fix the program and re-run it.

```
    printf("a is at %x (%lu)\n",&a,&a);
    printf("b is at %x (%lu)\n",buf,buf);
    printf("c is at %x (%lu)\n",&c,&c);
    printf("d is at %x (%lu)\n",&d,&d);
```

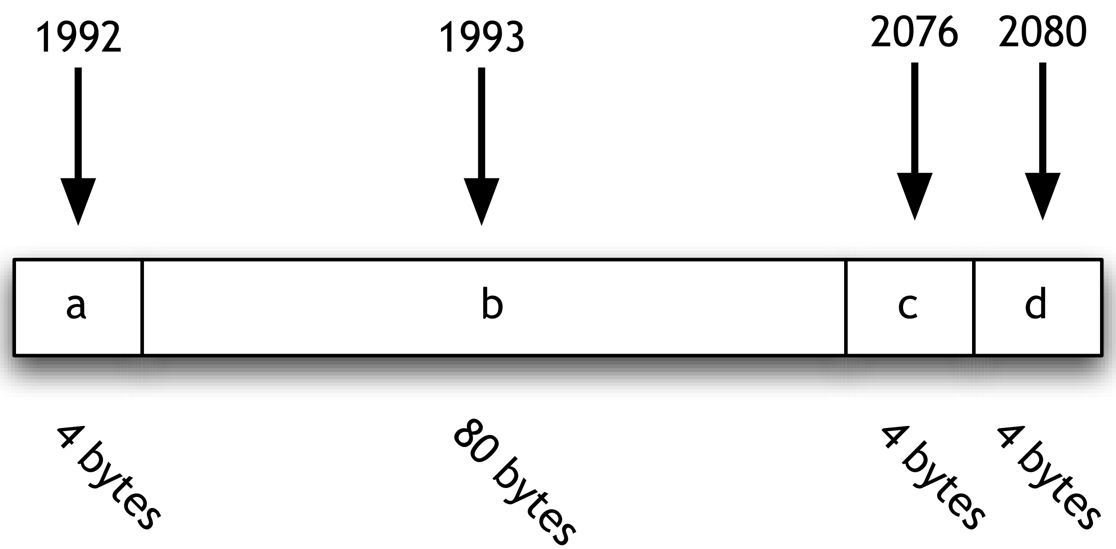
Results are:

```
a is at bffff268 (3221221992)
b is at bffff269 (3221221993)
c is at bffff2bc (3221222076)
d is at bffff2c0 (3221222080)
```

These numbers are locations in memory—the stack!

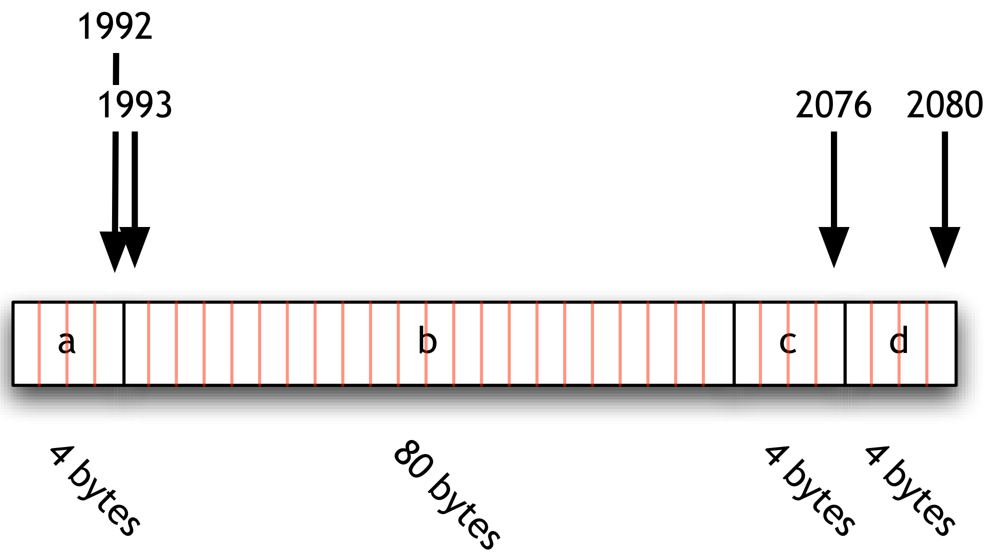
Draw a map of what we think this looks like:

```
a is at bffff268 (3221221992)
b is at bffff269 (3221221993)
c is at bffff2bc (3221222076)
d is at bffff2c0 (3221222080)
```



This is what the stack really looks like:

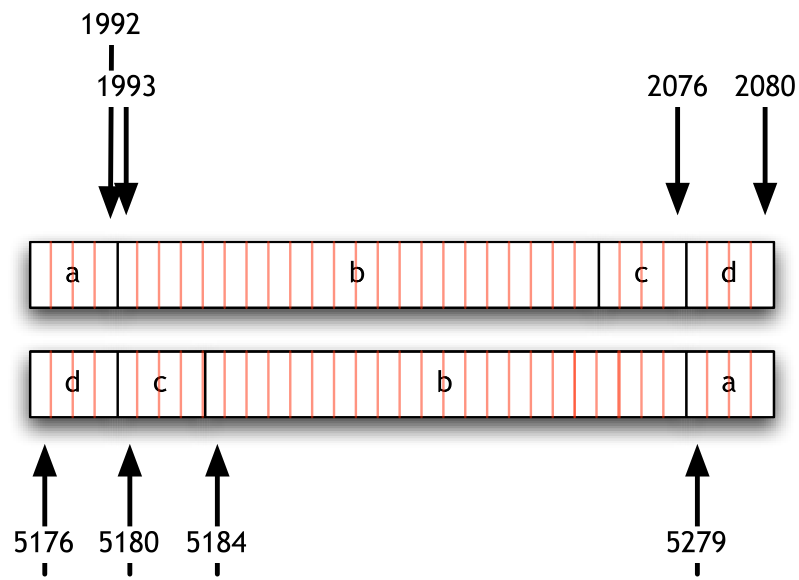
```
a is at bffff268 (3221221992)
b is at bffff269 (3221221993)
c is at bffff2bc (3221222076)
d is at bffff2c0 (3221222080)
```



... on a PowerPC

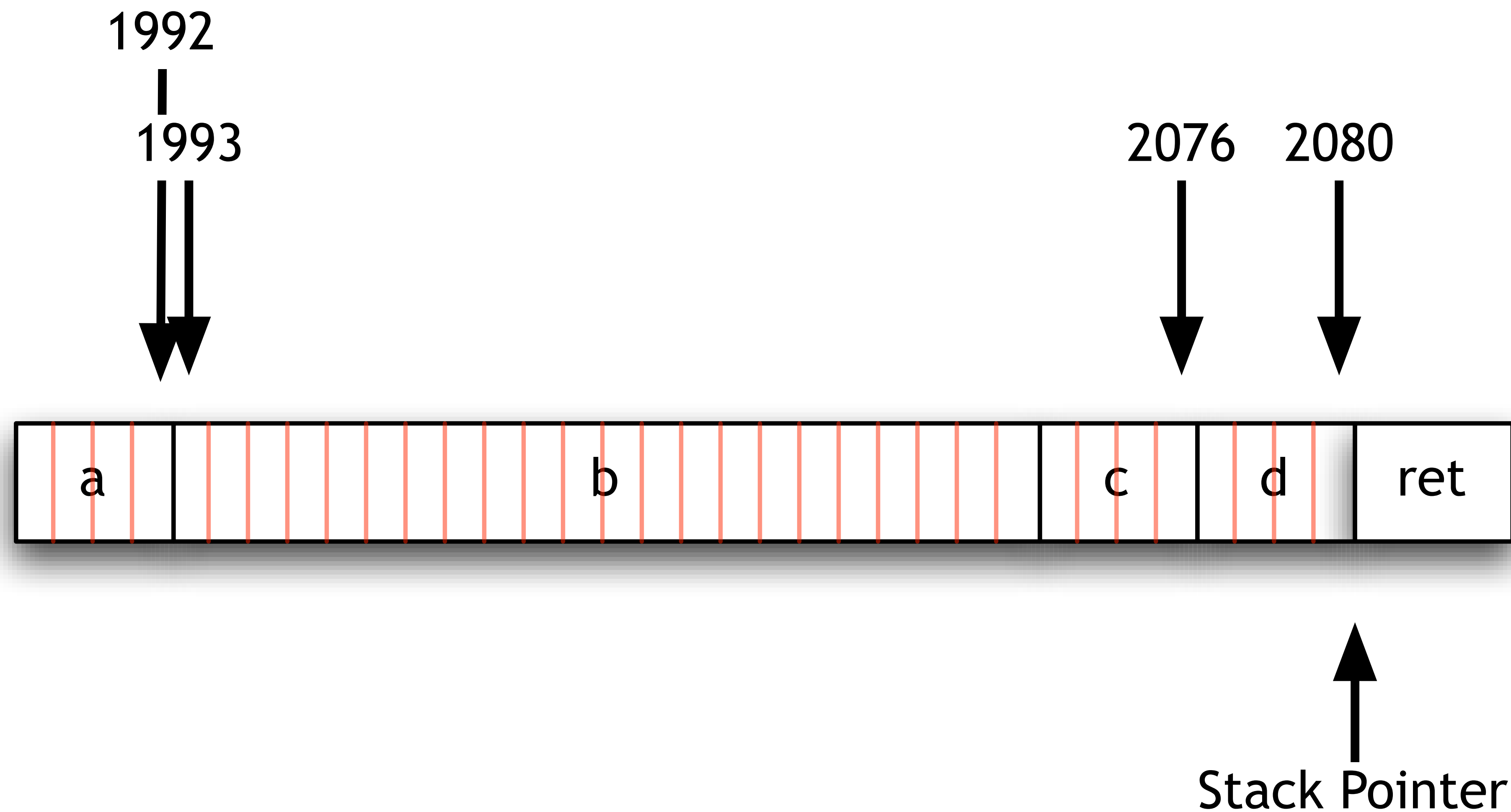
Different architectures will give different answers:

```
a is at bffff268 (3221221992)
b is at bffff269 (3221221993)
c is at bffff2bc (3221222076)
d is at bffff2c0 (3221222080)
```




```
a is at bfbfe8ff (3217025279)
b is at bfbfe8a0 (3217025184)
c is at bfbfe89c (3217025180)
d is at bfbfe898 (3217025176)
```

main() is a function.
The return address is also on the stack.



When the function is finished executing,

`PC←mem[SP]`

x86: `printf("d is at %x (%lu)\n",&d,&d);`

```
        .section        .rodata
.LC3:
        .string "d is at %x (%lu)\n"
...
        .text
...
        addl    $16, %esp
        subl    $4, %esp
        leal    -112(%ebp), %eax
        pushl   %eax
        leal    -112(%ebp), %eax
        pushl   %eax
        pushl   $.LC3
        call    printf

        addl    $16, %esp
        movl    $0, %eax
        leave
        ret
        .size   main, .-main
        .ident  "GCC: (GNU) 3.4.4 [FreeBSD] 20050518"
```

PowerPC: `printf("c is at %x (%lu)\n",&c,&c);`

```
        .cstring
LC3:
        .ascii "d is at %x (%lu)\12\0"
        .align 2
...
        .text
        .align 2
        .globl _main
_main:
...
        addi r0,r30,144
        addi r9,r30,144
        addis r2,r31,ha16(LC3-"L00000000001$pb")
        la r3,lo16(LC3-"L00000000001$pb")(r2)
        mr r4,r0
        mr r5,r9
        bl "L_printf$LDBLStub$stub"    ; $
```

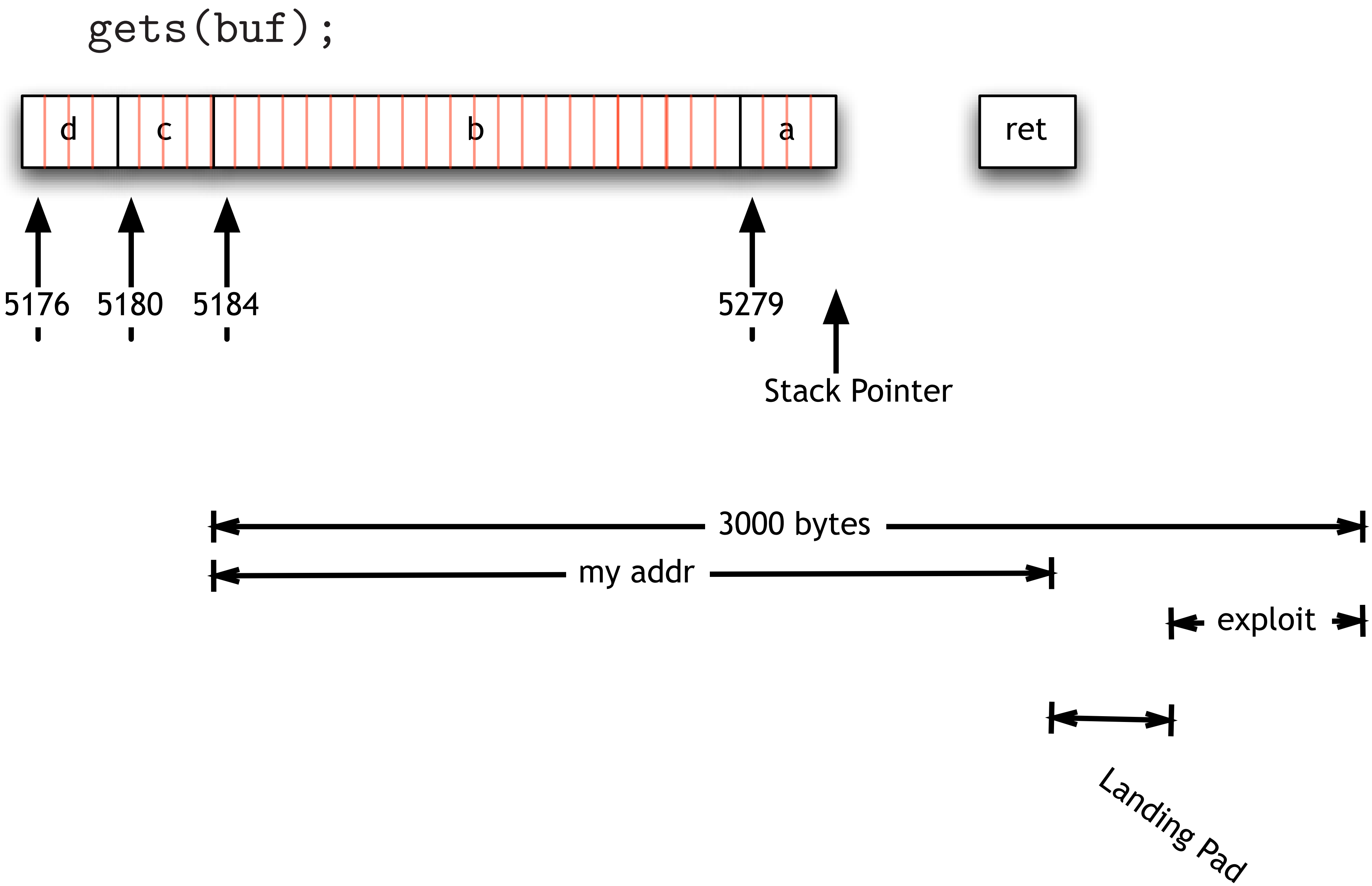
To understand this, you really need to understand the PowerPC calling sequence. See `http://www.linuxbase.org/spec/ELF/ppc64/spec/x280.html`

This program has an exploitable buffer overflow.

```
main(int argc,char **argv)
{
    int a;
    char buf[80];
    int c,d;

    puts("What is your name?");
    gets(buf);
    printf("I'm glad to meet you, %s\n",buf);
}
```

Overflow the buffer with an exploit.



Buffer overflow references

Aleph One, Smashing the Stack for Fun and Profit, in Phrack issue 49, November 9, 1996.
`http://www.simson.net/ref/1996/smashstack.txt`

"64-bit PowerPC ELF Application Binary Interface, Function Calling Sequence," Free Standards Group, 2004,
`http://www.linuxbase.org/spec/ELF/ppc64/spec/x280.html`

"Calling convention," Wikipedia,
`http://en.wikipedia.org/wiki/Calling_convention`

The SQL injection and buffer overflow exploits show the importance validating program input.

These are some of the most common errors today.

Buffer overflows aren’t possible in Java, Python or Perl, but other errors are.

Homework 4: Three Attacks

Attack #1: Buffer-overflow attack

Attack #2: SQL-injection attack

Attack #3: Hidden URL attack

Grading will be by web page; no partial credit.
Should be up in 1 week.