Lab 8: Format String Vulnerabilities

Reading:

• Jon Erickson, *Hacking: The Art of Exploitation*, Section 0x350: Format Strings

• scut/team teso, “Exploiting Format String Vulnerabilities” (can be found at http://cs.wellesley.edu/~cs342/papers/formatstring/formatstring-1.2.pdf).

1 Fun with printf

Here we will learn how to use printf not only to inspect the stack but also, 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 of format specifiers, which are treated as holes in the plain text. The remaining arguments are expected to be values whose printed representations, as determined by the corresponding specifiers, will fill corresponding holes. Here are some of the format specifiers:

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

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 1. This program expects 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@lark format-vulnerabilities] test-printf "a=%i; b=%u; c=%x; d=%s;"
  With values: a=42; b=4294967254; c=bffff2ec; d=xyz;
  Printing: 10, 20, 80485bd, 80485ba
  Without values: a=10; b=20; c=80485bd; d=40;
```

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

#include <stdio.h> // Headers that include types of printf and scanf

int test (char* fmt, int a, int b, int* c, char* d) {
    printf(" With values: ");
    printf(fmt, a, b, c, d);
    printf("\n Printing: %i, %i, %x, %x", 10, 20, "30", "40");
    printf("\nWithout values: ");
    printf(fmt);
    printf("\n");
}

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

Figure 1: The contents of test-printf.c.

displayed as expected. In the second line, the integers 10 and 20 are displayed, followed by the
hex addresses of the strings “30” and “40”. In the third line, no explicit values are provided for
the four arguments, so these are taken from the stack, and happen to be (with this compiler and
invocation, your mileage may vary) the same four values supplied in the second call to printf.

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

\begin{verbatim}
[cs342@lark format-vulnerabilities] test-printf "a=%10i; b=%12u; c=%08x; d=%5s;"
    With values: a= 42; b= 4294967254; c=\texttt{bffff2ec}; d= \texttt{xyz};
    Printing: 10, 20, 80485cb, 80485c8
Without values: a= 10; b= 20; c=080485cb; d= 40;
\end{verbatim}

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 2.

Normally, a format specifier refers to the “next” argument in the argument sequence. But
starting a specifier with \%\textit{j}$ refers to the \textit{j}th argument (1-indexed) in the argument sequence.
This notation can be combined with the field-width notation:

\begin{verbatim}
[cs342@lark format-vulnerabilities] test-printf "a=%3\$15i; b=%1\$12u; c=%2\$08x; d=%4\$5s;"
    With values: a= -1073745188; b= 42; c=\texttt{fffffffdd}; d= \texttt{xyz};
    Printing: 10, 20, 80485cb, 80485c8
Without values: a= 134514123; b= 10; c=000000014; d= 40;
\end{verbatim}

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:

\footnote{\textit{n} is a lower bound on the number of characters.}
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`:

```c
int main () {
    int x, y, z;
    printf("a=%i; %nb=%5i; %nc=%10i;%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`:

```bash
gcc -o test-nspec test-nspec.c
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.

## 2 Stack Hacking Revisited

Figure 2 presents a program `hackme2.c` that is similar to the `hackme` program from Lab 7 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`. 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. We will see both of these exploits in the next section.

---

2 `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`.

3 All examples in this section should be executed on micro-focus machine `lark`, on which both stack randomization and Exec Shield have been turned off (i.e., as root, setting both `/proc/sys/kernel/randomize_va_space` and `/proc/sys/kernel/exec-shield` to 0).
/* A program that hints at issues involving software exploits */
/* Compile this as: gcc -o hackme2 hackme2.c */

#include <stdio.h> // Headers that include types of printf and scanf

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 *****
", process(a));
}

Figure 2: The contents of hackme2.c.

Supplying an index outside the bounds of the array results in squaring the value in stack that happens to follow the array.

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

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

```
perl -e 'print "%08x %08x %08x %08x\n"x10 . "">"
```

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 7 words later. So entering the index 7 squares 2.

**Problem 1** Show how to use hackme2 to display the square of any positive number $n$. Demonstrate this for $n = 1000$ in two ways:

1. without using the %n specifier;
2. with using the %n specifier.

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 pointer to the return address. 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*.

**Answer to Problems from Section 2**

Now that we know argc is at an offset of 7 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:

The Perl dot (.) operator concatenates two strings.
Here, Perl creates 999 space-separated copies of the command-line argument > (but only displays the first one). 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 bffff254 because it appears right below the frame boundary for getelt (bffff238 0804846b) and right below the frame boundary for process (bffff268 0804846b). These occurrences of bfffe4f0 are at offsets 12 and 20, 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:

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.

Since the address of a[0] is at both offsets 12 and 20 from the top of the stack, the following work as well:

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.
***** ANS = 1000000 *****

[cs3420|lark format-vulnerabilities] hackme2 "%20%0100x%12$n>"

***** ANS = 1000000 *****