Lab 7: Runtime stack

Lab sessions Tue Nov 14 to Thu Nov 16

Lab written by Julie Zelenski

Learning goals

This lab is designed to give you a chance to:

1. learn some new gdb tricks (just in time for bomb!)
2. observe and understand the correct operation of the runtime stack
3. diagnose symptoms of stack mismanagement

Find an open computer and somebody new to sit with. Introduce yourself and brag about how much sleep you plan to catch up on over Thanksgiving recess.

Get started

Clone the lab7 repo by using the command below.

```
git clone /afs/ir/class/cs107/repos/lab7/shared lab7
```

Open the lab checkoff form (https://web.stanford.edu/class/cs107/cgi-bin/lab7).

Lab exercises

1) Leveling up with gdb

Gdb is your go-to when defusing the infamous binary bomb. Share your most productive gdb tips with your labmates and let's try to get a few new tricks into everyone's repertoire!

Examining stack frames

A quick refresher of some gdb commands we previously introduced:

- Use `backtrace` to display all of the stack frames back to main. With an argument N, `backtrace N` shows only the N innermost frames and `backtrace –N` shows only the N outermost.
- The `up`, `down`, and `frame n` commands allow you to change the selected frame. These commands don't change the state of program execution (the execution remains where it stopped in the topmost frame), but they allow you to examine the runtime state from the perspective of another stack frame. For example, changing to the frame main allows to print the variables/parameters that are visible only in that scope.
- The `info frame` command tells you the story about this stack frame. The `info args` and `info locals` provide information about the parameters and local variables, respectively.
- You can view data on the stack using the `x` command on `$rsp`

  ```
  (gdb) x/4gx $rsp   // 4 quadwords, in hex, read from top of stack
  ```

Printing registers, auto-display

The `info reg` command displays current values for all integer registers. Use `p $rax` to print value from a single register. In gdb, remember that access to a register is via `$name` instead of the more familiar `%name`.

A register is treated as an untyped 8-byte value and when you ask gdb to print it, it shows a decimal integer or hex address. You can dictate how to interpret the value by adding a /fmt to the print command or by using a C typecast, e.g.

```
(gdb) p $rax
$1 = 4196128
(gdb) p/t $rax
$2 = 1000000000011100100000
(gdb) p (char *)$rax
$3 = 0x400720 "Hello, world!\n"
```

The handy gdb command `display` sets up an expression to be repeatedly evaluated and printed as you single-step. For example, try these display expressions:

```
(gdb) display/2gx $rsp         // 2 quadwords read from stack top
(gdb) display/3i $rip          // next 3 asm instructions to execute
```

Shrewd use of `display` can approximate a poor man's version of `tui` (minus the flakiness and garbled display).

Pro-tip: breakpoint commands

You can set up a sequence of commands for gdb to execute each time a particular breakpoint is reached. Any valid gdb command is fair game -- get a backtrace, enable/disable other breakpoints, evaluate a C expression, change the value in a register, and so on. You can even use breakpoint commands to patch buggy code from within the debugger!

Let's say a buggy line 192 allocates one fewer byte than needed:

```
 192   s = malloc(strlen(t));         // oops, supposed to be len+1
```

Set a breakpoint on line 192. Now add commands to that breakpoint to insert a correct allocation, jump over the buggy line, and continue from there.

```
(gdb) break 192
Breakpoint 1 at 0x400a8d: file program.c, line 192.
(gdb) commands
Type commands for breakpoint(s) 1, one per line.
End with a line saying just "end".
print s = malloc(strlen(t)+1)
jump 193
continue
end
```

Each time line 192 is about to be executed, the breakpoint commands intervene to automatically insert the correct allocation, skip over the buggy statement, and continue execution. Neat! (and perhaps quite useful if needing protection from a hair-trigger explosive device...)

2) Stack mechanics

Let's practice some of these new commands in gdb while examining the mechanics of the runtime stack. Open `stack.c` in your editor and peruse the C code for the various functions. Build the `stack` program and run gdb on it.

- Disassemble the functions to see coordination between the caller and callee for a function call. Trace through the caller setup (parameters passing, call), callee prolog (push registers to save, make space on stack if needed), callee epilog (pop registers to restore, return value), and caller resume.
- Set up the display expressions from above (to show stack top and upcoming instructions). Then `stepi` through function call and return and note how `$rip` and `$rsp` are changing.

x86 has a mere handful of registers and they are in high demand; the compiler works hard to maximize their use. Parameters and return value are passed/returned using registers, local variables will be kept in registers whenever possible. The compiler will prefer use of scratch registers (e.g. callee-owned registers) where possible so as to avoid having to save/restore the caller's data (as is required if using the caller-owned registers)

- Examine the disassembly to determine where each local variable is being stored. Which are stored in registers? Which are on the stack?
- The prolog of the `calls` function pushes the values from the caller-owned registers `%rbp` and `%rbx` to the stack and the epilog pops them back off. This is consistent with the requirement that if a callee wants to use a caller-owned register it must preserve its value and restore when done. What does `calls` use those registers for? Why must it use caller-owned registers? Why would a callee-owned register not work in this case?
- The `locals` declares a fairly long list of local variables, but manages to juggle them in the callee-owned registers, without needing to write to the stack or dip into the caller-owned registers. Why is it able to do this?
- Compare the C source for the `calls` function to its disassembly. Where did the call to `atoi` go? Open the header file `/usr/include/stdlib.h` in your editor and search for the definition of the `atoi` function to see how this transformation was arranged. `atoi` is defined as a `inline` function, which instructs the compiler to paste the body of the function in at the calling site.

3) Pass params by "channeling"

The `channel` program was inspired by a bug I once helped a novice student resolve. Their code was "working fine" until they added a print statement which caused a incommensurate amount of grief. Removing the print statement "fixed" the problem, very mysterious!

1. Review the code in `channel.c`. The `init_array` and `sum_array` function each declare a stack array. The init function sets the array values, the sum function adds the array values. Neither init nor sum takes any arguments. There is a warning from the compiler about `sum_array`, what is it?
2. The program calls `init_array()` then `sum_array()`. Despite no explicit pass/return between the two calls, the array seems to be magically passed as intended. How are these functions communicating?
3. The program's second attempt to channel the array from init to sum fails. What is different about this case?

Tell your partner a story about a previous situation where your program's behavior was reproducibly yet inexplicably altered by adding/removing/changing some innocent and seemingly unrelated code -- does this exercise shed light on a possible explanation?

4) Recursion

The `fact` program runs a classic recursive factorial function.

1. Run the `fact` program on a few small numbers: 2, 5, 9 and then on successively larger numbers. What is the smallest argument for which the computed result becomes erroneous? Why does that happen?
2. Let's push things a bit further. Look at the code and trace the behavior of the call `factorial(-1)`. Do you and your partner agree on the expected result? Run the program on the argument -1 and see its result. Didn't I once tell you that a segmentation fault can only come from memory mismanagement and use of bad numbers isn't usually catastrophic? How then is an invalid number causing a memory problem? Run `fact` under gdb and use `backtrace` to find out where the program was executing at the time of the crash. Does this shed some light on what has gone wrong?
3. Get into `gdb` and figure out how big each `factorial` stack frame is. Here are two different strategies; try both!
    - Disassemble `factorial` and scan its instructions. You are looking for those instructions that change the `%rsp`. Each `pushq` and `callq` instruction copies an 8-byte value to the stack and decrements `%rsp`. The `%rsp` can also be explicitly moved to open up space for local variables or scratch results. (Factorial has no such manual adjustment of `%rsp`. The only data stored in its frame are saved registers placed there by push/call instructions.) Find those instructions in `factorial` and sum the total number of bytes stored in its stack frame.
    - Set a breakpoint on `factorial`, let a few calls through, and then use the gdb commands `info frame 1` and `info frame 2` to examine the top two stack frames. Look for the frame addresses labeled `Stack frame at` and subtract the two to compute the size of a single frame.
    - These two approaches should give you the same answer. Do they?
4. In the OS on `myth`, the stack segment is configured at program start to a particular fixed limit and cannot grow beyond that. The bash shell built-in `ulimit -a` (or `limit` for csh shells) displays various process limits. What does it report is the process limit for stack size?
5. Divide the stack limit by the size of a `factorial` frame. This estimates the maximum stack depth for factorial. Try factorial of -1 under gdb again. When it crashes, use `backtrace 10` and `backtrace -10` to see stack top and bottom and subtract frame numbers to determine the stack depth. How close is that depth to the maximum you estimated?
6. Edit the Makefile to change the optimization level used to compile the program. Find the line that reads `fact: CFLAGS += -Og`. The current setting is `-Og`, a relatively modest level of optimization. Change `-Og` to `-O2` to apply more aggressive optimizations. Make clean and make, then run fact -1 again. Woah! What happened? Did the compiler "fix" the infinite recursion? Disassemble the optimized `factorial` to see what the compiler did with the code. This fancy optimization is called "tail-recursion elimination (http://en.wikipedia.org/wiki/Tail_recursion)".

5) Debugging stack misuses

What happens when a function creates chaos within its stack frame? What are the observed symptoms? How do we debug these kinds of errors?

Love thy neighbor. The `clear_array` function in `buggy.c` has the wrong bounds of on its loop. It's a common-enough bug, especially to programmers unaccustomed to C's zero-based array indexing, but it has a surprising consequence in this context.

1. Before you try executing the program, first discuss with your partner: what do you expect to be the consequence?
2. Run the program at the shell. It never seems to complete. Hmmm.
3. Run it under gdb. Interrupt the program and see where it is executing. Continue the program, then interrupt again. Where is executing now?
4. Interrupt again and then step from here. Why is the loop taking so darn long? Just what is happening with the value of `i`?

The value of `i` is changing as a result of code that doesn't directly assign to it. These kind of bugs are calling "stomping" or "clobbering" memory. A variable can be stomped on due to an under/overrun of a neighboring region, using a pointer into a deallocated stack frame, referring to freed heap memory, or using an uninitialized pointer. These errors are difficult to debug since the variable is not named in the code that affects it and the stomping may go unnoticed until much later, making it difficult to connect cause and effect. Whereas Valgrind is awesome about reporting when you write to an invalid address, it cannot detect that you wrote to a valid address for the wrong reason. We need another strategy/tool to find this kind of error.

Let's learn how to use a gdb *watchpoint*. The gdb `watch` command is given an expression or a memory location to watch. gdb sets up a special kind of breakpoint that stops your program whenever there is a change in the value of that expression or a write to that memory location.

```
(gdb) watch i               // report when i changes
(gdb) watch *0x608502       // report if write to memory location
```

1. Run under gdb, set a breakpoint on `clear_array` and once hit, set up a watchpoint to monitor changes to `i`.
2. Continue from here and gdb will stop on each change to i. The watchpoint will alert you to each increment of the counter in the loop: from 0 to 1, 1 to 2 and so on, but it also will report when `i` gets stomped on.
3. What code is responsible for clobbering `i`?

Watchpoints can be a useful tool for tracking down those bugs that make mysterious changes to memory (and may be a useful tool in reverse engineering a bomb, too. Hmm...)

gets, the sequel. `code.c` is a reprise of a program you explored as a code study exercise in assign3 (/class/cs107/assign3/). It uses the unsafe `gets` function to read the user's input into a buffer of size N. (Note the program is compiled without stack protector, so we have disabled protection against stack smashing). Any input longer than N will overflow the buffer for which the consequence ranges from silent to catastrophic depending on how much excess. In assign3, you experimented to empirically determine out what length resulted in a crash. This time, you will work from the disassembly to compute the triggering length.

1. Disassemble the `main` function and sketch a diagram of its stack frame, including the location of the buffer and its relationship to neighboring data.
2. For any input longer then N, the characters read by `gets` will overflow the buffer. What is the immediate neighbor of the buffer? What is the consequence of overwriting it?
3. What is the critical data that when overwritten, will cause the program to crash? How long must the input be to reach this critical data? Verify your calculation with empirical experimentation.
4. The crash does not occur at the time when the data is overwritten; it is later when the corrupted data is used that is the problem. Identify the assembly instruction in the `main` function at which the crash occurs and explain what is happening in that instruction.
5. Set a breakpoint on the crashing assembly instruction and add these breakpoint commands:

   ```
   (gdb) b *0xADDR       // replace ADDR with instr addr in main
   (gdb) commands
   Type commands for breakpoint(s) 1, one per line.
   End with a line saying just "end".
   p *(void **)$rsp = pluto
   continue
   end
   (gdb)
   ```

Run the program. With these commands in place, it no longer crashes on a long input. What does it do instead? Talk it over with your partner and work out how the above "fix" operates.

Check off with TA

Before you leave, be sure to submit your checkoff sheet and have your lab TA come by and confirm so you will be properly credited. If you don't finish everything before lab is over, we strongly encourage you to finish the remainder on your own. Double-check your progress with self check (/class/cs107/selfcheck.html#lab7).