A heap, a stack, a bottle and a rack.

Johan Montelius and Amir H. Payberah

Introduction

In this assignment, you will investigate the layout of a process: where different areas are located and which data structures go where.

1 Where's da party?

So we will first take a look at the code segment, and we will do so using a `gcc` extension to the C language. Using this extension, you can store a *label* in a variable and then use it as any other value.

1.1 The memory map

Start by writing and compiling this small C program, `code.c`. As you see, we use two constructs that you might not have seen before, the label `foo:` and the conversion of the label to a value in `&&foo`. If everything works fine, you should see the address of the `foo` label printed in the terminal. The program will then hang, waiting for keyboard input (actually anything on the standard input stream).

```
#include <stdio.h>

int main() {
 foo:
  printf("the code: %p\n", &&foo);

  fgetc(stdin);
  return 0;

}
```

So we have the address of the `foo` label, it could be something that looks like `0x40052a` (but could also be `0x55d8ef4426ce` depending on which version of Linux you are running). Hit any key to allow the program to terminate, and then we try something else. We will now tell the shell to start the process in the background. That will allow us to use the shell while the program is suspended, waiting for input.

```
> ./code&
[1] 2708
the code: 0x55968c40f6ce
```

The number 2708 (or whatever you see) is the process identifier of the process. Now we take a look in the directory `/proc`, where we find a directory with the name of the process identifier. There are about fifty different files and directories in this directory, but the only one we are interested in is the file `maps`. Take a look at it using the `cat` command [1].

```
$ cat /proc/2708/maps
   :
```

What you see is the memory mapping of the process. Can you locate the code segment? Does it correspond to the address we found looking at the `foo` label? What is the protection of this segment? To bring the suspended process back to the foreground, you use the `fg` command. Then you can hit any key to allow the program to terminate.

```
 > fg
./code
```

1.2 Code and read only data

To make things a bit more convenient, we extend our program so that it will get its own process identifier and then print out the content of `proc/<pid>/maps`. We do this using a library procedure `system()` that will read a command from a string and then execute it in a sub-process.

[1] OSX users can try: `usr/bin/vmmap 2708`

```c
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

char global[] = "This is a global string";

int main() {

  int pid = getpid();
foo:
  printf("process id: %d\n", pid);
  printf("global string: %p\n", &global);
  printf("the code: %p\n", &&foo);

  printf("\n\n /proc/%d/maps \n\n", pid);
  char command[50];
  sprintf(command, "cat /proc/%d/maps", pid);
  system(command);

  return 0;
}
```

Look at the address of the global string. In which segment do you find it? What is the protection of this segment? Does it make sense? Now allocate a global constant data structure, print its address, and try to locate it - hmmm, is it where you thought it would be?

```c
  const int read_only = 123456;
    :

  printf("read only:  %p\n", &read_only);
    :
```

2 The stack

The stack in C/Linux is located almost at the top of the user space and grows downwards. You will find it in the process map, and the entry will probably look something like this:

```
7ffed89d4000-7ffed89f5000 rw-p 00000000 00:00 0          [stack]
```

Before we start to play around with the stack, we ponder the memory layout of the process and how it is related to the x86 architecture.

2.1 The address space of a process

The addresses on a 64-bit x86 machine are 8 bytes wide or 16 *nibbles* (4-bit unit corresponding to one hex digit). Look at the address `0x7ffed89f5000`.

How many bits are used? What is the 48th bit? In an x86-64 architecture, an address field is 64-bits, but only 48 are used. The uppermost bits (63 to 47) should all be the same; generally, if they are 0, it is the user space, and if they are 1, it is the kernel space. The user space thus ends in:

```
0x00 00 7f ff ff ff ff ff
```

The kernel space then starts in the logical address:

```
0xff ff 80 00 00 00 00 00
```

2.2 Back to the stack

So we have a stack, and it should be no mystery what data structures are allocated on the stack. We extend our program with a data structure that is local to the main procedure and tracks where it is allocated in memory.

```c
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main() {
  int pid = getpid();

  unsigned long p = 0x1;
  printf("p (0x%lx): %p \n", p, &p);

  printf("\n\n /proc/%d/maps \n\n", pid);
  char command[50];
  sprintf(command, "cat /proc/%d/maps", pid);
  system(command);

  return 0;
}
```

Execute the program and verify that the location of `p` is actually in the stack segment. If you run the programs several times, you will notice that the stack segment moves around a bit. This is by intention; it should be harder for a hacker to exploit a stack overflow and change things in the heap or vice versa. If you want the stack segment to stay put, you can try to give the following command in the shell:

```
$ setarch $(uname -m) -R bash
```

2.3 Pushing things on the stack

Now let's do some procedure calls and see if we can see how the stack grows and what is placed on it. We keep the address of `p`, make some calls, and then print the content from the location of another local variable and the address of `p`. The procedure `zot()` is the procedure that will do the printing, and it requires an address and will then print one line per item on the stack. We should maybe check that the given address is higher than the address of the local variable `r`, but it is more fun living on the edge.

```
void zot(unsigned long *stop) {
  unsigned long r = 0x3;
  unsigned long *i;

  for(i = &r; i <= stop; i++)
    printf("%p 0x%lx\n", i, *i);
}
```

We use an intermediate procedure called `foo()` that we only use to create another stack frame.

```
void foo(unsigned long *stop ) {
  unsigned long q = 0x2;

  zot(stop);
}
```

Now for the `main()` procedure that will call `foo()` and do some additional print out of the location of `p` and the return address `back`.

```
int main() {
  int pid = getpid();
  unsigned long p = 0x1;

  foo(&p);

 back:
  printf("  p: %p \n", &p);
  printf("  back: %p \n",  &&back);

  printf("\n\n /proc/%d/maps \n\n", pid);
  char command[50];
  sprintf(command, "cat /proc/%d/maps", pid);
  system(command);

  return 0;
}
```

If this works, you should have a nice stack trace. The interesting thing is now to figure out why the stack looks like it does. If you know the general structure of a stack frame, you should be able to identify the return address

after the call to `foo()` and `zot()`. You should also be able to identify the saved *base stack point*, i.e., the value of the stack pointer, before the local procedure starts to add things to the stack. You also see the local data structures: `p`, `q`, `r` and a copy of the address to `p`. If it was a vanilla stack, you could also see the argument for the procedures. However, `gcc` on an x86 architecture will place the first six arguments in registers, so those will not be on the stack. You can add ten dummy arguments to the procedures, and you will see them on the stack.

If you do some experiments and encounter elements that can not be explained, it might be that the compiler just skips some bytes to keep the stack frames aligned to a 16-byte boundary. The *base stack pointer* will thus always end with a 0. Try locating the value of the variable `i` in the `zot()` procedure. It is, of course, a moving target, but what is the value of `i` when the location of `i` is printed? Can you find the value of the process identifier? Convert the decimal format to hex, and it should be there somewhere.

3 The Heap

Ok, so we have identified the code segment, a data segment for global data structures, a strange kernel segment, and the stack segment. It is time to take a look at the heap. The heap is used for data structures that are created dynamically during runtime. It is needed when we have data structures that have a size that is only known at runtime or should survive the return of a procedure call. Anything that should survive returning from a procedure can not be allocated on the stack. In C, there is no program construct to allocate data on the heap; instead, a library call is used, i,e., the `malloc()` procedure (and its siblings). When `malloc()` is called, it will allocate an area on the heap - let's see if we can spot it.

Create a new file `heap.c` and cut and paste the structure of our main procedure. Keep the tricky part the printing the memory map but now include the following section:

```
char *heap = malloc(20);

printf("the heap variable at: %p\n", &heap);
printf("pointing to: %p\n", heap);
```

Locate the location of the `heap` variable. It is probably where you would suspect it to be. Where is it pointing to, a location in the heap segment?

3.1 Free and reuse

The problem with using the heap is not how to allocate memory but to free it, only free it once and not use the freed memory. The compiler will detect

some apparent errors, but most errors will only appear at runtime and might be very hard to track down. If you allocate a data structure of 20 bytes every millisecond and forget to free it, you might run out of memory in a day or two, but if you only run small benchmarks, you will never notice.

Using a pointer to a data structure that has been free has an *undefined behavior* meaning that the compiler nor the execution needs to preserve any predictable behavior. This said, we could play around with it and see what might happen. Try the following code and reason about what is happening.

```
int main() {

  char *heap = malloc(20);
  *heap = 0x61;
  printf("heap pointing to: 0x%x\n", *heap);
  free(heap);

  char *foo = malloc(20);
  *foo = 0x62;
  printf("foo pointing to: 0x%x\n", *foo);

  /* danger ahead */
  *heap = 0x63;
  printf("or is it pointing to: 0x%x\n", *foo);

  return 0;
}
```

If you experiment with freeing a data structure twice, you will most certainly run into a segmentation fault and a core dump. The reason is that the underlying implementation of `malloc()` and `free()` assume things are structured in a certain way and when they are not, things break. Remember that by definition, the behavior is undefined, so you can not rely on that things will crash when you test your system. We can look at the location just before the allocated data structure to see what is going on (it will differ depending on what operating system you are using). We will here use `calloc()` that will not only allocate the data structure but also set all its elements to zero. Try the following, also print the memory map as before.

```
  long *heap = (unsigned long*)calloc(40, sizeof(unsigned long));

  printf("heap[2]:  0x%lx\n", heap[2]);
  printf("heap[1]:  0x%lx\n", heap[1]);
  printf("heap[0]:  0x%lx\n", heap[0]);
  printf("heap[-1]: 0x%lx\n", heap[-1]);
  printf("heap[-2]: 0x%lx\n", heap[-2]);
```

So we are cheating and access position `-1` and `-2`. As you see, there is some information there. Now change the size of the allocated data structure and see what is happening. You might wonder how `free()` knows the size of the object that it is about to free - any clues? Now look at this - we will

free the allocated space, but then we cheat and print the content. Add the following below the code we have above.

```
free(heap);

printf("heap[2]:  0x%lx\n", heap[2]);
printf("heap[1]:  0x%lx\n", heap[1]);
printf("heap[0]:  0x%lx\n", heap[0]);
printf("heap[-1]: 0x%lx\n", heap[-1]);
printf("heap[-2]: 0x%lx\n", heap[-2]);
```

Has something changed? Is it just garbage, or can you identify what it is? Take a look at the memory map. What is happening?

4 A shared library

When we first printed the memory map, there were lots of things that you had no clue of what they were. One after one, we have identified the segments for the code, data, strange kernel stuff, the stack, and the heap. There has also been a lot of junk in the middle, between the heap and the stack. Some of the segments are executable, some are writable, and some are described by something similar to:

```
/lib/x86_64-linux-gnu/ld-2.23.so
```

All of the segments are allocated for shared libraries, either the code or areas used for dynamic data structures. We have been using library procedures for printing messages, finding the process identifier, allocating memory, etc. All of those routines are located somewhere in these segments. The `malloc()` procedures keep information about the data structures that we have allocated and freed, and if we mess this up by freeing things twice, of course, things will break. If you do these experiments early in the course, you might not know what we are talking about, but it will all become clear.

5 Summary

You can learn about how things work by running very small experiments. You should test things for yourself and experience what you learn in the course. It is one thing reading about a segmentation fault on a slide, and another experience it by writing a small program. Remember the golden rule of engineering:

> If In Doubt, Try It Out!