6 Calling putchar() from an ELF Weird Machine.

by Rebecca .Bx Shapiro

Pastor's Exordium.[7] *Behold the daily miracle of the loader: it takes stored dumb bytes and makes them into a new process or splices them into a running one. The Pharisees may dismiss it as mere engineering, but verily I tell you, long after their textbooks are forgotten the loader and its Phrack exegesis will shine on, for there is more wisdom gathered in its metadata structures than can be found in a dozen OS textbooks.*

Yet there is more! The binary metadata structures consumed by the loader are actually a program *for the loader. A weird machine devotee will readily recognize that these data drive all the actions behind the loader's miracle; they can be thought of as executable bytecode for the loader, which can be thought of as a virtual machine. And just as assembly with all its glorious* `mov`*s,* `add`*s, and* `call`*s is encoded in opcodes and offsets, ABI metadata entries are encoded in types and addends, except that they are split into symbols and relocation structures, residing in different sections of the binary but cross-referenced by their entry numbers in the respective sections.*

In this follow-up to earlier work, Bx shares more nifty tricks of programming the ELF loader with relocation and symbol data as weird assembly. This work is as advanced as it is neighborly, so please read her articles from WOOT 2013 and POC‖GTFO 00:05 to learn how to build a Turing-complete virtual machine out of an ELF loader and how to extend that VM to call native code. In this sermon, Bx shows us how to make system calls from ELF relocation and symbol data; full shellcode is left as an exercise to the faithful! –PML

— — — —

Welcome back, friends. In the first edition of POC‖GTFO, I demonstrated how we can craft ELF relocation metadata to instruct the loader to make libc calls. The method I demonstrated was fairly limited and lacked the ability to do useful things such as control the arguments passed to the called function. Thus I ended the article with an unsolved challenge: *How can metadata control the arguments passed to the metadata-initiated function call?*

In this sermon, I will partially answer that challenge by demonstrating how to control a call to `putchar()` using relocation metadata.

```
PUTCHAR(3)           bx's Programmer's Manual            PUTCHAR(3)

SYNOPSIS
       #include <stdio.h>

       int putchar(int c);

DESCRIPTION
       putchar(c) writes the character c, cast to an unsigned char, to stdout.

RETURN VALUE
       putchar() returns  the character written as an unsigned char cast to
       an int or EOF on error.

       puts() and fputs() return a nonnegative number on success, or EOF on error.
```

One may ask "why focus on `putchar()`?" The answer is simple. Because `putchar()` is required in order to implement a full, honest-to-manul brainfuck-to-ELF metadata compiler. You may have noticed that `putchar()` requires only a single (byte-long) argument and have thought to yourself "I only have control over one argument!? How will that help me take over the world?" Don't worry your pretty little

[7] How is a sermon like a binary file? Both have prescribed parts that follow each other in a conventional order, but may be skipped or used creatively by an extra neighborly preacher. Convention is there to help, but it's the result that matters. So just think of *exordium* as the ELF/ABI header or vice versa and bear with the Preacher as you bear with your binary toolchain! *–PML*

nose off. I will provide insight on how you can control not one, not two, but three (ish) arguments to a function call!

Instead of asking how one can control the first argument to a function call, one should really be asking how can we be the last to set the RDI register (the first argument to a function as heralded by the System V amd64 ABI gospel 3:2:3, aka amd64 calling convention[8]) before our metadata-driven libc function is called.

It turns out that the loader generally processes each relocation entry within a single function, although there are a few exceptions to this rule. This means that, generally speaking, the arguments that are in place during any metadata-driven function call are the arguments that were passed to the currently executing function processing the relocation entries. An exception to this "rule" occurs when relocation entries of type R_X86_64_COPY are processed. These types of relocation entries cause the loader to make a call to memcpy(), thus changing the values of RDI, RSI, RDX, which by convention hold the first three arguments to a function call, and in the case of a call to memcpy(void *dest, const void *src, size_t n) hold dest, src, and size, respectively.

Now imagine that the dynamic loader has been processing our relocation entries and now the next dynamic symbol, pointed to by the next relocation entry r0 to be processed, looks like this:

```
s0 = {..., st_value = &putchar, st_size = 0x0}
```

(Note: We have already shown how to calculate the address of libc functions in past work and will not cover how to do that in this sermon. See our WOOT article and POC||GTFO 00:05 for a thorough explanation.)

The following three relocation entries (represented here as C structs, but of course encoded in a .rel section) will make a call to putchar(), to print the character of our choice:

```
r0 = {r_offset=<&r2->r_addend>, r_symbol=0, r_type=R_X86_64_64,
      r_addend=0x0}
r1 = {r_offset=<char to print>, r_symbol=0, r_type=R_X86_64_COPY,
      r_addend=0x0}
r2 = {r_offset=&r2, r_symbol=0, r_type=R_X86_64_IRELATIVE,
      r_addend=<&putchar (filled in by r0)>}
```

The purpose of r0 is to write the address of putchar() into r2's addend. The purpose of r1 is to setup RDI (the first argument) for r2's function call. When it is processed, memcpy() is called with the following arguments: memcpy(<char to print>, &putchar, 0). More generally, the call to memcpy() looks like: memcpy(r1->r_offset, s0->st_value, s0->st_size).

After r1 is processed, 0 byes are copied from &putchar to <char to print>[9], and RDI=<char to print>, RSI=&putchar, and RDX=0. r2, of type R_X86_64_IRELATIVE, instructs the loader to treat its addend as a function pointer, making a call to it(!). How's that for a relocation-based weird assembly instruction? But, one problem: relocation entries of type IRELATIVE do not support functions that require arguments (meaning that there is no conventional way to pass them). Still, the actual function doesn't care and will happily reach for its arguments in RDI etc.—and, luckily, we were able to set up the arguments via our relocation-entry crafted call to memcpy() via r1! Hence r2 will cause the loader to call putchar(), which will consult RDI to determine what character to print to stdout.

You may see the potential downfalls of manufacturing a call to memcpy() in order to put arguments in place for the following library call. For example, if the third argument is not zero, you need to start worrying about your first two arguments pointing to read/writable memory. However, it may be comforting to know that the value returned by the function call is written into a spot of your choosing (in r2->r_offset).

If you would like to further your studies of metadata-driven library calls, please refer to the **elf-bf-tools** repository on github.[10] May the Great Manul keep and protect you from the Weird Machine. And let us say, amen.

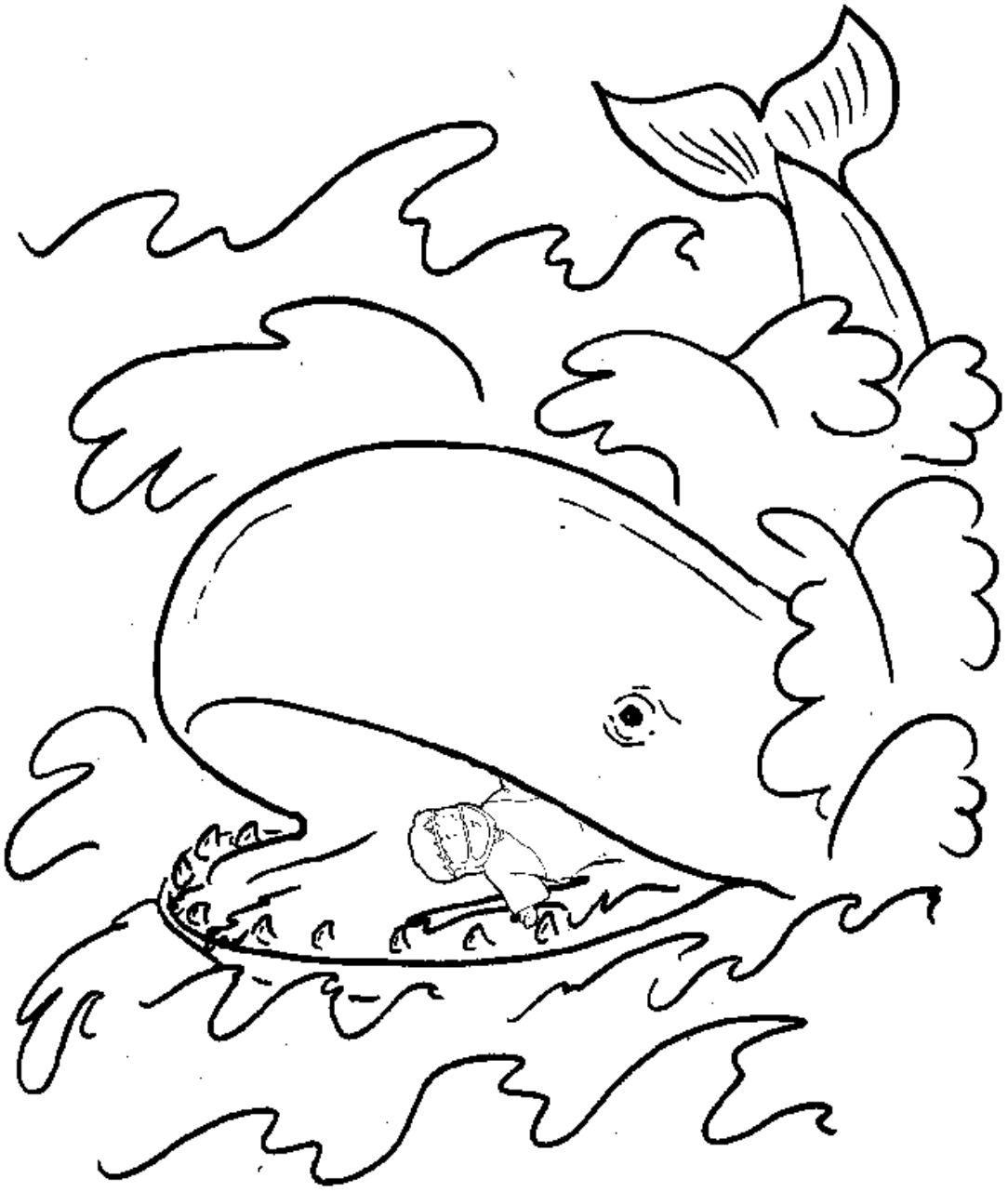
[8] http://www.x86-64.org/documentation/abi.pdf, pages 17-21, Fig. 3.4—and don't ask us in what world RDI, RSI, RDX might stand for A, B, C or suchlike. This program may be brought to you by the register RDI anyhow, but let's just say if the Manul meets the amd64 Big Bird there might be feathers flying.

[9] Note, memcpy would treat it as a destination pointer, but luckily nothing gets copied here, and memcpy implementation isn't paranoid about checking its arguments, since a bad pointer would trap anyway.

[10] See syscall/putchar in https://github.com/bx/elf-bf-tools .

```
446 case R_X86_64_IRELATIVE:
447  value = map->l_addr + reloc->r_addend;
448  value = ((Elf64_Addr (*) (void)) value) ();
449  *reloc_addr = value;
450  break;


429case R_X86_64_COPY:
430  if (sym == NULL)
431    /* This can happen in trace mode if an object could not be (gdb)
432       found.  */
433    break;
434  memcpy (reloc_addr_arg, (void *) value,
435  MIN (sym->st_size, refsym->st_size));
436  if (__builtin_expect (sym->st_size > refsym->st_size, 0)
437      || (__builtin_expect (sym->st_size < refsym->st_size, 0)
438  && GLRO(dl_verbose)))
439    {
  440      fmt = ``\
  441%s: Symbol `%s' has different size in shared object, consider re-linking\n'';
(gdb)
  442      goto print_err;
  443    }
444  break;
445#  endif


--------------
Breakpoint 6, elf_machine_rela (sym=0x601030, reloc_addr_arg=0x601241, version=<optimized out>,
    reloc=0x601318, map=0x555555773228) at ../sysdeps/x86_64/dl-machine.h:434
    434  memcpy (reloc_addr_arg, (void *) value,
(gdb) print/x *reloc
$6 = {r_offset = 0x601241, r_info = 0x5, r_addend = 0x0}
(gdb) print refsym->st_size
$7 = 0
(gdb) print sym->st_size
$8 = 0
(gdb)
(gdb) print/x reloc_addr_arg
$9 = 0x601241
(gdb) x/gx reloc_addr_arg
    0x601241:0x0000000060103800
(gdb) x/gx value
```

```
    0x7ffff7ce1184:0x011d8b48f8894153
(gdb) print/x $rsi
$5 = 0x7ffff7ce1184
(gdb) print $rdx
$10 = 0

(after memcpy)
(gdb) x/gx 0x601241
    0x601241:0x0000000060103800
(gdb) print/x $rdi
$14 = 0x601241
(gdb) c
Continuing.

Breakpoint 5, elf_machine_rela (sym=0x601030, reloc_addr_arg=0x6012e8, version=<optimized out>,
    reloc=0x601330, map=0x555555773228) at ../sysdeps/x86_64/dl-machine.h:448
    448  value = ((Elf64_Addr (*) (void)) value) ();
(gdb) print/x $rdi
$15 = 0x601241
(gdb) print/x value
$16 = 0x7ffff7ce1184
(gdb) x/10i value
    0x7ffff7ce1184:push   %rbx
    0x7ffff7ce1185:mov    %edi,%r8d
    0x7ffff7ce1188:mov    0x313c01(%rip),%rbx        # 0x7ffff7ff4d90
    0x7ffff7ce118f:mov    (%rbx),%eax
    0x7ffff7ce1191:test   $0x80,%ah
    0x7ffff7ce1194:jne    0x7ffff7ce11ea
    0x7ffff7ce1196:mov    %fs:0x10,%r9
    0x7ffff7ce119f:mov    0x88(%rbx),%rdx
    0x7ffff7ce11a6:cmp    0x8(%rdx),%r9
    0x7ffff7ce11aa:je     0x7ffff7ce11df
(gdb) print/x $rsi
$4 = 0x7ffff7ce1184
```



Just as Jonah was told to preach in Nineveh,
Pastor Laphroaig was once called to preach to the harlots and tax collectors at RSA.
Asked about the experience, he said that, like Jonah,
he'd rather be thrown overboard than go back.