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 movs, adds, and calls 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-manual 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

———

7How 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

14
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:

\[
s0 = \{\ldots, \text{st\_value} = \&\text{putchar}, \text{st\_size} = 0x0\}\]

(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:

\[
\begin{align*}
r0 &= \{\text{r\_offset}=<&r2->r\_addend>, \text{r\_symbol}=0, \text{r\_type}=\text{R\_X86\_64\_64}, \\
    &\quad \text{r\_addend}=0x0\} \\
r1 &= \{\text{r\_offset}=<\text{char to print}>, \text{r\_symbol}=0, \text{r\_type}=\text{R\_X86\_64\_COPY}, \\
    &\quad \text{r\_addend}=0x0\} \\
r2 &= \{\text{r\_offset}=\&r2, \text{r\_symbol}=0, \text{r\_type}=\text{R\_X86\_64\_IRELATIVE}, \\
    &\quad \text{r\_addend}=\&\text{putchar} (\text{filled in by } r0)\} \\
\end{align*}
\]

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(<\text{char to print}>, \&\text{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 bytes are copied from `\&\text{putchar}` to `<\text{char to print}>`, and RDI—`<\text{char to print}>`, RSI—`\&\text{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 found. */
434 break;
435 memcpy (reloc_addr_arg, (void *) value,
436 MIN (sym->st_size, refsym->st_size));
437 if (__builtin_expect (sym->st_size > refsym->st_size, 0)
438 || __builtin_expect (sym->st_size < refsym->st_size, 0)
439 & & GLRO(dl_verbose))
440 
441 fmt = '\"%s: Symbol \"%s\" has different size in shared object, consider re-linking\n\"';
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) print/x reloc_addr_arg
$9 = 0x601241
(gdb) x/gx reloc_addr_arg 0x601241:0x00000000060103800
(gdb) x/gx value

16
```
(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
```

---

**P.C. cards made simple—with COPYDAT!**

1. Prepare the 1X artwork, using an opaque layout aid such as Chartpak, Bishop Graphics, or other similar product.
2. Make a negative: Place the artwork face down, cover with the negative material colored film side up (we recommend Scotchplast products), and expose with the Copydat. Typical exposure time is 1.5 minutes.
3. Develop the negative in developer provided with negative material.
4. Attach negative to pre-sensitized face of copper board. Place board and negative face down on Copydat. Expose. Typical exposure time: 30 seconds.
5. Save the negative for reuse, and develop the board in the developer provided.
6. Etch the board.
7. As a finishing touch, tin the board to avoid oxidation of the copper and to improve solderability.

Result: a custom, high quality, single-sided P.C. board.

With careful alignment, you can make doublesided boards too!

Alternatively, buy high-quality hardware assemblies from us — and these are predrilled as well (and feature plated-through holes):

P.S. The Copydat does a lot more than make high-quality P.C. boards. It makes superior blueinline, blackline, sepia, and other diazo process copies, and you can make pressure-sensitive labels with it and even instrument front panels from pre-sensitized metal plates!!

---

from $149.95 (B size prints)  CELDAT Design Assoc.
P.O. Box 752
Amherst, N.H. 03031

17
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.