Starting from a Clean Slate: Creating a Top-down Parseable Runtime

Randolph Langley
langley@cs.fsu.edu

1 Motivations

Going far off the beaten track of programming language development, I have become interested in creating programming language paradigms that preserve a well-defined runtime format.

Let me try to motivate this a bit. Try this experiment: create a simple x86_64 assembly language program that does nothing except exit and assemble and link it statically. (You can find a tar file here that includes some YASM code and a Makefile to make the various executables referenced here.)

From a debugger, execute this (the resulting binary is called `null` from the tar file), break on the `_start` symbol, and admire the very clean register set, with only the instruction pointer and the stack pointer among the general registers having any non-zero values. Among the math registers, we only see a sea of zeroes.

Now run a statically-linked assembly program that incorporates a system call to `getuid(2)` (this is called `systemcall` in the tar file); again, break right after the system call, and the registers are all nice and clean. Stepping through the system call, we see that while RAX has the correct userid value, RCX has acquired a value of -1 and R11 now has 770 (0x302). This isn't unexpected; the ABI explicitly states in A.2.1 that these registers are "destroyed." ([MHJM14])

Now run a statically-linked assembly program that incorporates a call to the GLIBC library function wrapper for the `getuid(2)` system call (the binary is called `wrapper-static` in the tar file.) The GLIBC function only returns one value in RAX. Again, break just after the start, and before the call to `getuid(3)`, and again, the registers look very clean.

Taking one step and calling `getuid(3)` (the GLIBC wrapper), like calling the system call directly, changes both RCX and R11, but differently, with RCX changing to 0x4000d7 and R11 to 0x202. Also, our stack now has a return address matching our call to `getuid(3)` in the "unused" area.

Now let's look at the registers when we run a dynamically linked version of the same program (the binary is called `wrapper-dynamic` in the tar file.) Even before starting anything in this binary, our dynamic loader has apparently left quite a bit of "digital debris" lying about. RAX has a 0x1c, RCX has 0x7ffff7ffe758, RDX has 0x7ffff7dea560, RSI has 0x1, RDI has 0x7ffff7ffe1c8, R8 has 0xb, R9 has 0x4, R10 has 0xd, R11 has 0x8, R12 has 0x400240, R13 has 0x7fffffffe040, and SSE3 registers YMM0 through YMM4 all have non-zero state.

If we take one step, RAX now has 1000 (0x3e8), our expected userid value, and RCX now has 0x7ffff7ad6e67, R10 has 0x7fffffffde00, and R11 has 0x202. While the R11 change is the same as with the statically-linked version, the others are not; indeed, R10 wasn't even changed in the static version.

So, even statically linked, GLIBC doesn't clean up after its use of registers; with `ld.so`'s dynamic linking, the register set is now positively untidy and strangely uneven.

2 Striving for a structured environment, where uncleanness is incorrectness

"Programs rely on a complex runtime environment that includes several libraries. It is often impossible to analyze the source code of these libraries. Frequently, only the binaries are available and, even when source code is available, some functions are hand-written in assembly. Yet, many attacks make use of libraries when exploiting vulnerabilities." [CCH06]

From a security perspective, this doesn't seem to me a good position to be in. I think it would be desirable from a security perspective to be able to uniformly **specify** what the exact state should be for all registers and all memory for a "knowable" process. While one could use data-flow integrity to at least detect a broad range of bad behaviors, data-flow integrity by itself does not guarantee that the data is well-formed, just not undesirably modified [CCH06].

The strawman that I propose is an entire programming language paradigm built around a credo of knowing, from top-down parsing of register state through all memory utilization, that

- That all process state is all well-formed;
- That all process state is consistent;
- That all process state is semantically coherent.

I will call process state that has all three of these properties "knowable."

To specify well-formedness, we start with actual hardware. This is different than formulations that work over an abstract machine; instead, we bite the bullet and eschew optimizations over speed or space, and instead try to optimize on the basis of having a precise hardware idea of what is supposed to be where.

In this instance, I will posit that we are working on x86_64 hardware with a Linux operating system. At any point in the computation, we can distinguish all three conditions given register state and working from there to memory-mappings (in this case, I am not using the stack given to the process by the kernel, and I do not create a heap with `brk(2)/sbrk(2)`.)

To do this, we optimize our programming language's use of registers and memory for knowability: that is, starting with the registers, we can verify that the register state is well-formed, and from those registers, we then derive that the memory mappings and their contents are all consistent and semantically coherent.

However, I don't believe that we want to pay the price of re-verifying this at every computational step in a practical setting; instead, we will try to use state changes that always preserve these properties, and provide frameworks for both internal and external verification.

So how do we do this?

There are two parts to this. The first is the runtime system of the programming language has to be able to 1) create new dynamic memory mappings, 2) structure these dynamic mappings so that they preserve knowability, and 3) mediate program requests for memory allocation and deallocation over these memory mappings. The runtime dedicates some number of registers to specify these memory-mapped areas.

The second is that the programming language uses a runtime model that uses a simple and regular system for all of its activities; to that end, I have been using a multiple stack model. There are four types of stacks: 1) a return stack; 2) an evaluation stack; 3) a local variable stack; and 4) an exception stack. All of these stacks are allocated from the runtime system's memory mappings, and dedicated registers point to the first three.

The semantics for every programming language construct must be explicable in terms of either modifying clearly defined internal state in the compiler. We also don't want a situation where, as documented in [WZKSL13], improper performance optimizations cause undefined behavior.

References

[CCH06] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 147–160, Berkeley, CA, USA, 2006. USENIX Association.

[MHJM14] Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. System V Application Binary Interface: AMD64 Architecture Processor Supplement, Draft Version 0.99.7. `http://www.x86-64.org/documentation/abi-0.99.7.pdf`, November 2014.

[WZKSL13] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 260–275, New York, NY, USA, 2013. ACM.