CMSC430 Final Project Report

Foreign Function Interfaces (Lawbreaker)

Garrett Alessandrini & Nicholas Romano

1. Introduction

Our compiler project was the implementation of a foreign function interface. This feature would allow for calling functions that are written in another language. In this implementation, the foreign function interface allows C functions to be called in Lawbreaker, which is based on Loot. At the base level, the programmer can call any c function they wish with as many arguments as they want. The return value of the called c function is provided afterward for further processing by our program. While this feature not only allows the programmer to use features that have yet to be implemented in Lawbreaker, it also opens up major security vulnerabilities due to the lack of memory safety in C.

1. Implementation
   1. Parser

Augmenting the loot parser to identify calls to a C function was not too difficult. In our main parsing pattern match, a clause was added to handle whenever “ccall” was invoked with the required parameters. On top of that, a helper function was created that can traverse the entire resulting AST and produce a list containing every unique C function called using ccall, which comes in handy during the compilation stage to inform the assembler of our c functions so that it will link properly with the runtime.

* 1. Compiler

The compiler for Lawbreaker is similar to the Loot Compiler with the addition of the ccall.rkt file that holds the majority of the code for the Foreign Function Interface (FFI). In order for the FFI to function properly, the calling conventions for the System V Application Binary Interface must be followed. Therefore, the invocation of a C function using “ccall” requires for the first six arguments to be passed into specific registers in the order: rdi, rsi, rdx, rcx, r8, and r9. If more than six arguments are used in the function call, the remaining arguments must be pushed onto the stack.

The compile-ccall function is defined as:

(define (compile-ccall f es env))

Parameter ‘f’ is the C function id, parameter ‘es’ is the list of arguments for the C function, and ‘env’ is the current environment.

The function “compile-ccall” is broken down into six operations for successfully compiling the C function call and its arguments:

1. Compile up to six arguments and move each one into its designated register based on the System V ABI calling convention.
2. Pad the stack to be 16 byte aligned for the call to a C function.
3. Compile the remaining arguments and push each one on the stack.
4. Call the C function ‘f’ with the A86 Call instruction.
5. Clean the stack by moving the stack pointer by the length of the arguments pushed onto the stack.
6. Unpad the stack to no longer be 16 byte aligned.
   1. Runtime / Creation of callable C functions

The vast majority of our code for handling the calling of C functions is handled by the parser/compiler, so the only code that has to be added into the runtime is a header to define the functions as well as a C file that defines the body of each function. We also add our new C file/header to the makefile in order to have it linked with the final executable. Lastly, just for good measure, we include our header in the main.c file of our runtime to be sure our custom functions are included in the compiled runtime. Since the parameters come to us in the representation of our language, rather than the C representation, we must run val\_unwrap\_<type> to convert the value into a format that C understands. We must also use val\_wrap\_<type> on our return value so that any code from our language that utilizes the ccall return value will not experience undefined behavior.

1. Exploiting the stack using C functions

Now that we can execute all the C code that we want, we can take advantage of some of the conventions that C uses when calling functions to showcase some stack exploits. When a C function is called, the arguments for that function must reside in locations specified by the System V calling convention. The part that we care about is that the first six arguments are put in specific registers, and the remaining arguments are placed on the stack. Assuming that we have a function that is called with over 6 arguments, using the reference operator on the 7th argument (or greater) will get us the location of our stack in memory! Considering just how much we rely on the information on the stack being accurate for our programs, this blows a huge hole in the security of our programs. In our demonstration programs, we were able to overwrite entire defines and lambdas that existed in the stack, as well as modify the values of variables on the stack. Granted, an attacker would need to be able to sniff out the stack layout of the program at that specific chunk of code, so we had an easier time devising how to achieve the desired results, but with trial and error, it would definitely be possible.