Linking Assignment

Due: Wednesday, October 25, 11:59pm

This assignment is about understanding how the ELF format supports run-time linking for shared libraries on x86_64, including the way that references to global variables and functions are implemented and represented. Your task is implement `redact`, which copies an ELF-format shared library and

- identifies all global functions that are exported by the shared library;
- determines the *secrecy level* of each function or accessed global variable, where the secrecy level is determined by digits at the end of the variable or function name; and

  > A name that doesn't end with digits counts as secrecy level 0.

- changes the code of each function to crash with a trace/breakpoint interrupt when (and only when) the function tries to access a global variable at a higher secrecy level or call a global function at a higher secrecy level.

  > You don't have to know how to implement a trace/breakpoint interrupt, since that's covered by a helper library that is included with the initial implementation.

For example, given a shared library that is the compiled form of

```
int a1[3] = { 1, 2, 3 };
int a2[3] = { 4, 5, 6 };

int f1(int v) {
  if (v > 0)
    return a1[0];
  else {
    a2[0] = 1;
    return 0;
  }
}

int f0(int u) {
  if (u == 1)
    return f1(u);
  else
    return 5;
}

int g(int u) {
  return a1[2];
}
```

then the shared library has the following initial behaviors:

- `f1(1)` will return 1 by acessing `a1`.
- `f1(-1)` will modify `a2` and return 0.
- `f0(0)` will return 5.
- `f0(1)` will return 1 by calling `f1`.
- `g(1)` will return 3 by accessing `a1`.

A copy of the shared library produced by `redact`, however, will have the following behaviors:

- `f1(1)` will return 1. Since both `f1` and `a1` have secrecy level 1, `f1` is allowed to access `a1` after redaction.
- `f1(-1)` will crash. Since `a2` has secrecy level 2, `f1` is *not* allowed to access it after redaction.
- `f0(0)` will return 5. No global variables or functions are accessed or called, so no access or call must be redacted.
- `f0(1)` will crash, since it `f0` has secrecy level 0 and is not allowed to call `f1` (which has secrecy level 1).
- `g(1)` will crash, since it `g` has secrecy level 0 and is not allowed to access `a1` (which has secrecy level 1).

Start with `linklab-handout.zip`.

Starting Implementation

The starting code `"redact.c"` handles copying a shared-library file to a destination file based on the command-line arguments. The `redact` function—which doesn't do anything in the starting code—receives a pointer to the in-memory copy of the destination file. Your job is to fill in the implementation of `redact` so that it changes the destination file.

To determine function and variable information, your `redact` function will read the destination ELF file as it exists in memory. To replace certain variable accesses and function calls with a crash, your `redact` function will modify the in-memory copy of the ELF file. Through the magic of of `mmap` (which we will cover later in the semester), those in-memory changes will be reflected in the destination file.

The initial `"redact.c"` includes a `get_secrecy_level` function that takes a string for a function or variable name and returns an integer for the secrecy level that is embedded in the name. A secrecy level is computed by converting a sequence of digits at the end of the name string. For example, `"f13"` has secrecy level 13, while `"g"` has secrecy level 0 (since it doesn't end in any digits). Although you're allowed to modify anything in `"redact.c"`, you shouldn't have to modify `get_secrecy_level`.

A key step in the implementation of `redact` is to decode the machine code that implements a function. Since decoding x86_64 machine code is tedious, the starting implementation includes `"decode.c"` and `"decode.h"`, where `"decode.c"` implements a `decode` function. The `decode` function is sufficient for functions in the shared libraries that we will use to test your implementation. In addition, `"decode.c"` provides `replace_with_crash`, which can replace an instruction with one to provoke a crash through a trace/breakpoint interrupt. See Decoding Machine Code for more information on using `decode` and `replace_with_crash`.

Your submitted solution will take the form of a single C file, `"redact.c"`, which must be implemented in ANSI C. You must not modify `"decode.c"` or `"decode.h"`, since you submit only your `"redact.c"` file. We'll compile and link `redact.c` in the same way as in `"Makefile"`, so it must be self-contained other than its uses of `"decode.h"`, `"decode.c"`, `"elf.h"`, limited Unix headers and libraries for `open` and `mmap`, and ANSI C headers and libraries.

Assumptions

To simplify the your task, you can make several assumptions:

- Your `redact` need only recognize and adjust functions that are registered in the dynamic-symbol table (i.e., the `.dynsym` section) as a function type with a non-zero size. Your `redact` can assume that functions are not inlined at call sites and that references to global variables are not optimized away.

  For example, if `do_something` is implemented as

```
static int a100[1] = { 1 };

static int g100(int) {
  return 5;
}

int f0(int v) {
  return g100(a100[0]);
}
```

the your `redact` need not modify the access of the `static` array `a100` or the call of the static function `g100`, since neither of those will registered in the dynamic-symbol table of a shared library. Also, since `g100` is `static`, its implementation is likely to be inlined in place of the call in `f0`.

- Your `redact` can assume the usual strategies for implementing relocatable variables and functions in Linux. In particular, accessing a variable will go through a memory location that is listed in the `.rela.dyn` section, and jumping to an imported or exported function will go through the global offset table as described by a `rela.plt` section.
- Your `redact` can assume that all references to functions are function calls, and no references to variables are function calls.
- Your `redact` need only handle functions where the implementation is supported by `decode`. The `decode` function recognizes variants of `mov`, `ret`, `cmp`, `test`, `jmp`, and j*cc* instructions.
- Your `redact` need only handle *tail calls* to functions, which correspond to `jmp` instructions (as opposed to `call` instructions). A tail call is a function call whose result is immediately returned by the enclosing function, and GCC will compile such calls as `jmp`s.
- Your `redact` can assume that the functions in a shared library contain no loops, recursive function calls, or calls among mutually recursive functions. Your `redact` can also assume that a function contains a less than 20 conditional branches. These constraints simplify the traversal of a function's machine code.

Evaluation

To earn 100 points, your implementation must correctly replace each reference or call to a higher-secrecy variable or function with a trace/breakpoint interrupt.

You can earn up to 80 points by not handling function calls, but still replacing references to higher-secrecy variables with a trace/breakpoint interrupt.

When grading, we will infer a level of completion based on your program's behavior on the tests that are included with `linklab-handout.zip` (see Test Cases and Test Harness). We will then scale your grade based on the number of our test cases that pass.

Your `redact` can print out any information that you find useful. Our tests will look only at the shared library produced by `redact`.

Decoding Machine Code

The `decode` function provided by `"decode.c"` has the prototype

```
void decode(instruction_t *ins, code_t *code_ptr, Elf64_Addr code_addr);
```

where `code_ptr` refers to the machine code to decode, and `code_addr` is the address where that code will reside at run time. (The `code_addr` must be provided so that `decode` can tell the destination of relative references in the machine code.) The type `code_t` is an alias for `unsigned char`. Use `code_t*` for a pointer to machine code, so that pointer arithmetic works in terms of bytes, since `decode` reports an instruction length in bytes.

The `ins` argument receives the result of decoding one instruction, so pass the address of a locally declared `instruction_t` to `decode`. The `instruction_t` struct type has three fields: `op`, `length`, and `addr`. The `decode` function always fills in `ins->op` and `ins->length`, and it fills in `ins->addr` for some values of `ins->op`.

The possible values for `op` are (as defined in `"decode.h"`):

- `MOV_ADDR_TO_REG_OP` — The instruction moves a constant address into a register, possibly to access a global variable. The `ins->addr` field is set to the run-time address, and it's a reference to a global variable if that address corresponds to a variable that is registered as a dynamic symbol.
- `JMP_TO_ADDR_OP` — The instruction jumps to a constant address, possibly as a tail call to an global function. The `ins->addr` field is set to the destination of the jump as a run-time address, and it's a call to a global

function if that address corresponds to a function that is registered as a dynamic symbol. The instruction after the jump need not be considered, unless it is reached through some other jump.

- `MAYBE_JMP_TO_ADDR_OP` — The instruction conditionally jumps to a constant address, most likely due to an `if` in the original program. The `ins->addr` field is set to the destination of the jump as a run-time address. This jump is never a call to a function, but the code at the target of the jump may go on to call a function or access a variable. If the jump is not taken, the immediately following instructions might access a variable or call a function.

- `RET_OP` — The instruction returns from the current function, so the instruction after the return need not be considered, unless it is reached through an earlier jump. The `ins->addr` field is not set.

- `OTHER_OP` — The instruction is a move, arithmetic operation, or comparison operation that involves only registers and non-address constants. Your `redact` can assume that these operations do not somehow synthesize the address of a global variable or function, so it can effectively ignore them. The `ins->addr` field is not set.

For all operations, the `ins->length` field is set to the length of the machine-code instruction in bytes, so `code_ptr + ins->length` is a pointer to the next instruction, if any. The assumptions allowed for `redact` means that both branches of a conditional (i.e., the target of a `MAYBE_JMP_TO_ADDR_OP` instruction and the immediately following instruction) can be explored by using a recursive call to an instruction-traversal procedure for one or both of the branches.

If `decode` does not recognize the content at `code_ptr` as a machine-code instruction, it reports an error and exits. We will use test inputs where public functions conform to the constraints of `decode`, so you do not need to handle machine code that `decode` does not recognize.

The `"decode.c"` library also provides `replace_with_crash`:

```
void replace_with_crash(code_t *code_ptr, instruction_t *ins);
```

Pass `code_ptr` for an instruction that you want to replace with a trace/breakpoint interrupt. Also pass `ins` as filled with information about the existing instruction, so that `replace_with_crash` knows how many bytes to replace (filling with “no-op” instruction bytes as necessary to fully replace the original instruction).

Tips

The information about ELF that you need to complete this assignment is mostly covered by Videos: ELF. The map slides should be helpful to find your way through a file; see also a video explaining the map. More generally, you can use `"/usr/include/elf.h"` as a reference to find relevant structures, fields, and macros. You might also consult any number of other ELF references on the web.

Use `readelf` to get a human-readable form of the content of an ELF file to get an idea of what your program should find. Use `objdump -d` to disassemble functions to get an idea of what your program will recognize using `decode` and to compare your output shared library to the input shared library.

All of the information that you need from the ELF file can be found via section headers and section content, so you will not need to use program headers. You’ll need to consult the `.dynsym`, `.dynstr`, `.rela.dyn`, `.plt`, and `.rela.plt` sections, at least.

When working with ELF content, you have to keep track of which references are in terms of file offsets and which are in terms of (tentative) run-time addresses where the library will eventually run. When working with ELF content that is `mmap`ped into memory (as in the starting `"redact.c"`), then you have one more way of referencing things: a pointer in memory at present. Be careful to keep in mind which kind of reference you have at any time. Again, the maps should help.

Don’t confuse “symbol” with “string,” and keep in mind that they are referenced in different ways. Symbols are referenced by an index that corresponds to the symbol’s position in an array of symbols. Strings are referenced by an offset in bytes within the relevant string section. Every symbol has a string for its name.

Take small steps. Start out by printing the symbol index for every function that is provided by the shared library. Then, print the name instead of the symbol index. Then, print the address where each function’s implementation

is found, and so on. Make redaction of variables work before attempting to implement redaction of function calls.

Avoid parsing ELF information into your own set of data structures. An ELF file in memory can be viewed as a data structure already, based on the types that `"elf.h"` defines. Following different kinds of references is not always trivial, but it's not difficult, and you can write little functions to make access more convenient and readable. The task and examples are small enough that there's no need for extra tables or data structures to speed up searches.

As a lower bound, a complete solution can fit in about 250 lines of C code, including the 120 lines in the provided in the starting `"redact.c"`. Depending on whitespace (fine), comments (good), error checking (commendable), and duplicated code instead abstracting into a function (bad), many solutions will be the range of 300-400 lines.

Test Cases and Test Harness

The archive `linklab-handout.zip` provides a `"Makefile"` where the default target builds

- your `redact` program from `"redact.c"`;
- a `call` program to try a function from a shared library; and
- shared libraries `"ex1.so"` and `"ex2.so"` from the sources `"ex1.c"` and `"ex2.c"`.

For example, after running `make` with the initial files, you can check that calling `"ex1.so"`'s `f0` with the argument `0` produces the result `5` and with the argument `1` produces the result `1`:

```
$ call ex1.so f0 0
5
$ call ex1.so f0 1
1
```

After you have completed `redact` for redacting variable references, then

```
$ redact ex1.so dest.so
$ call dest.so f0 0
5
$ call dest.so f0 1
Trace/breakpoint trap (core dumped)
```

Alternatively, you can use `objdump -d dest.so` and compare to `objdump -d ex1.so` to check whether your `redact` has replaced an access of a variable address with an `int $0x3` instruction that triggers a trace/breakpoint interrupt.

The `test` makefile target runs your `redact` on the `"ex`*n*`.so"` shared libraries (overwriting `"dest.so"` for each attempt) and checks that the result behaves as predicted at the top of the `"ex`*n*`.c"` source.

You are not required to use the test files, but for grading purposes, we expect your `redact` to work correctly on `"ex1.c"` for a chance at more than 80 points (because `"ex1.c"` checks only variable references). For grading, we will run your program on additional shared-library files. We may also compile your program with different optimization or debugging options; as always, your program must build with `gcc` on CADE machines with no language-adjusting command-line flags.