Lab 2: Implementing a pintool for shadow stack

Preparation

Download pin from https://software.intel.com/en-us/articles/pintool-downloads. Unzip this tarball, you will find the executable "pin" already appears in the folder. You should also go into the sub folder "source/tools" and run "make" to build all the existing pintools.

You are highly recommended to read the user guide at https://software.intel.com/sites/landingpage/pintool/docs/76991/Pin/html/ carefully, to understand how to run an existing pintool, how an existing pintool is implemented, and how to write your own pintool.

Objective

Our objective for this lab assignment is to implement a shadow stack through dynamic binary instrumentation using Pin. Shadow stack is a well-known and effective defense mechanism to defeat control-flow hijacking attacks that aim to overwrite a return address on the stack. The general algorithm works like below: for each "call" instruction, identify the return address (or the next instruction after the call instruction), and push it onto the shadow stack; for each "ret" instruction, identify the return target and see if it matches with the value on the top of the shadow stack. If so, pop up the value from the shadow stack; otherwise, report an attack.

Work Flow

First of all, you need initialize Pin, and then register an instrumentation function with Pin, and start the program. Normally, you will do it in the main function, like below:

```
int  main(int argc, char *argv[])
{
   PIN_InitSymbols();
   if( PIN_Init(argc,argv) ) {
      return Usage();
   }
   TRACE_AddInstrumentFunction(Trace, 0);
   PIN_AddFiniFunction(Fini, 0);
   PIN_StartProgram();
   return 0;
}
```

In this example, we use Trace_AddInstrumentFunction to instrument a trace at a time. You can also use INS_AddInstrumentFunction to instrument an instruction at a time.

Then in your instrumentation function, you will enumerate each instruction to identify call and ret instructions. In particular, INS_IsCall and INS_IsRet can be used to determine if the specified instruction is a call or ret.

Once a call or ret instruction is identified, use INS_InsertCall to instrument that instruction. Please read the documentation for INS_InsertCall carefully (https://software.intel.com/sites/landingpage/pintool/docs/76991/Pin/html/group__INS__INST__API.html#g82d2ecc73dcd5e2af26fef5bd6ff7190).

```
VOID LEVEL_PINCLIENT::INS_InsertCall( INS        ins,
                                      IPOINT     action,
                                      AFUNPTR funptr,
                                                 ...
                                    )
```

Especially after the first three arguments, you can specify a list of arguments to be passed to “funptr”. For example, IARG_INST_PTR will pass the address of the instrumented instruction, and IARG_BRANCH_TARGET_ADDR will pass the target address of this branch instruction.

A possible way to instrument a call instruction is like below:

```
INS_InsertCall(ins, IPOINT_TAKEN_BRANCH, AFUNPTR(do_call),
                        IARG_BRANCH_TARGET_ADDR, IARG_RETURN_IP,
                        IARG_THREAD_ID, IARG_END);
```

In this example, Pin will pass the call target, the return address, and the thread ID to a function called “do_call” specified by the developer. So in the “do_call” function, you would push the return address on the shadow stack.

Similarly, you can instrument a ret instruction like below:

```
INS_InsertCall(insn, IPOINT_BEFORE, AFUNPTR(do_ret), IARG_INST_PTR,
        IARG_BRANCH_TARGET_ADDR, IARG_THREAD_ID, IARG_END);
```

In this example, you register a “do_ret” function for each ret instruction, and pass the address of the instrumented ret instruction, the branch target (which is the return address), and the thread ID. Since in the “do_ret” function, you would check the top of the shadow stack and see if it matches with the branch target. If so, you pop up from the shadow stack.

Then you are pretty much done with a basic version. Of course, a more complete implementation would need to handle multiple threads. It means that one shadow stack must be associated with each thread. More specifically, you need to create shadow stack in the thread local storage. In this lab assignment, you are not required to deal with multiple threads.

Submission

Please submit your report through iLearn, preferably in PDF format. In the report, please list your complete source code with sufficient explanation and output messages.

Run some normal programs like ls and ps, to show that your pintool won’t raise any false alarm.

Compile example01.c as a 64-bit binary, as below (note that “–m32” option is removed) :

```
gcc –g –fno-stack-protector –z execstack –o example01 example01.c
```

Run this binary with your pintool with a very long input and show that your pintool can detect the stack overflow before the program crashes.