Lab 2 Report **Abhishek Srivastava**
CS260-001: Computer Security Student Id: 861307778
Prof. Heng Yin 30/Jan/2017

Problem : Implementing a pintool for shadow stack

Objective for this lab assignment is to implement a shadow stack through dynamic binary instrumentation using Pin.

Solution

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 : For each call instruction, identify the return address, 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.

In Figure 1 you can see the Trace implementation of detecting the instruction to be of calling or returning type. We are going from block to block and for each tail instruction we are checking that whether it is calling or returning instruction. Iterating over blocks and checking only tail instructions makes this process fast by not going over each instructions. After checking we are executing based on the type of instruction, If it is a calling instruction we are passing it to the **do_call** analysis section or if it is returning instruction we are passing it to the **do_ret** analysis section.

In the Figure 2 you can see the implementation of both **do_call** and **do_ret** analysis section. In **do_call** section I am pushing the return address (**IARG_RETURN_IP**) parameter to the stack called *shadow*. In **do_ret** analysis section I am checking whether the top of *shadow* stack is same as the returning address(**IARG_BRANCH_TARGET_ADDR**). If they are the same value then it is okay to proceed and buffer overflow exploit has not been used. I am also maintaining a global parameter **isOkay** which can extended to stop the further execution. If the addresses are not same I am printing the error and setting **isOkay** to false because return address has been changed by buffer overflow exploit.

Figure 3, 4 and 5 are the executions of compiled code. In Figure 2 & 3, I tested the implementation by calling **/bin/ls** and **/bin/ps** whose output did not raised any error and produced the correct expected output. In Figure 5, I did 3 tests, First I checked whether code is giving Access denied message when the input is wrong password, then I checked if Access Granted is showed when inputed with correct password and at last I tested by overflowing the input buffer. Which as expected printing Error message that stack values does not match and segmentation fault is occurring which can be prevented using **isOkay** variable but I have not implemented it as it was not required.

Another thing which can be extended as well is maintaining shadow stacks for multi-threaded system. **IARG_THREAD_ID** can be used to maintain a global dictionary which can then be mapped to a particular shadow stack which will maintain the return address for that particular thread and rest of the implementation is same as for a single threaded system.

```
/* ===================================================================== */


VOID Trace(TRACE trace, VOID *v)
{
    //const BOOL print_args = KnobPrintArgs.Value();


    for (BBL bbl = TRACE_BblHead(trace); BBL_Valid(bbl); bbl = BBL_Next(bbl))
    {
        INS inst = BBL_InsTail(bbl);

        if(isOkay == true)
        {
            if( INS_IsCall(inst) )
            {
               INS_InsertCall(inst, IPOINT_TAKEN_BRANCH, AFUNPTR(do_call),
                              IARG_BRANCH_TARGET_ADDR, IARG_RETURN_IP, IARG_THREAD_ID, IARG_END);
            }
            else if( INS_IsRet(inst) )
            {
                INS_InsertCall(inst, IPOINT_BEFORE, AFUNPTR(do_ret), IARG_INST_PTR,
                              IARG_BRANCH_TARGET_ADDR, IARG_THREAD_ID, IARG_END);
            }
        }
    }
}

/* ===================================================================== */
```

Figure 1: TRACE section implementation.

```
/* ===================================================================== */

VOID  do_call(ADDRINT target, ADDRINT ret, THREADID tid)
{
    //const string *t = Target2String(target);
    //TraceFile << "Target: " << target << endl;
    //const string *r = Target2String(ret);
    TraceFile << "Return: " << ret << endl;
    shadow.push(ret);
    //Thread Id can be handled here
}

/* ===================================================================== */

VOID  do_ret(ADDRINT instaddr, ADDRINT target, THREADID tid)
{
    //const string *i = Target2String(instaddr);
    //TraceFile << "Instruction Address: " << instaddr << endl;
    //const string *tr = Target2String(target);
    TraceFile << "Return Adress: " << target << endl;
    if(shadow.top() == target)
    {
        shadow.pop();
    }
    else
    {
        isOkay = false;
        std::cerr << "Error!! Stack Values Does not match!!" << endl;
        TraceFile << "Error!! Stack Values Does not match!!" << endl;
    }

    //Thread Id can be handled here
}

/* ===================================================================== */
```

Figure 2: do_call and do_ret analysis section implementation.

```
$
$ ../../../pin -t obj-intel64/calltrace.so -- /bin/ls
bsr_bsf_app.cpp    calltrace.out  example01              ilenmix.cpp         malloctrace.cpp      oper-imm.cpp               shadowstack.cpp
bsr_bsf_asm.asm    catmix.cpp     extmix.cpp             inscount2_mt.cpp    objdump-routine.csh  oper-imm.ia32.reference    shadowstack.out
bsr_bsf_asm.s      coco.cpp       fence.cpp              inscount2_vregs.cpp obj-intel64          oper-imm.intel64.reference topopcode.cpp
bsr_bsf.cpp        dcache.cpp     flowgraph.py           jumpmix.cpp         opcodemix.cpp        pinatrace.cpp              toprtn.cpp
bsr_bsf.reference  dcache.H       get_source_app.cpp     ldstmix.cpp         oper_imm_app.cpp     regmix.cpp                 trace.cpp
callgraph.py       edgcnt.cpp     get_source_location.cpp makefile           oper_imm_asm.asm     regval_app.cpp             xed-print.cpp
calltrace.cpp      emuload.cpp    icount.cpp             makefile.rules      oper_imm_asm.s       regval.cpp                 xed-use.cpp
$
$
```

Figure 3: Output of /bin/ls test with compiled code.

```
$
$ ../../../pin -t obj-intel64/calltrace.so -- /bin/ps
  PID TTY          TIME CMD
 5109 pts/0    00:00:00 bash
 8053 pts/0    00:00:01 ps
$
$
```

Figure 4: Output of /bin/ps test with compiled code.

```
$
$ ../../../pin -t obj-intel64/calltrace.so -- ./example01
Enter password:
abhishek
Access denied
$ ../../../pin -t obj-intel64/calltrace.so -- ./example01
Enter password:
goodpass
Access granted
$
$
$ ../../../pin -t obj-intel64/calltrace.so -- ./example01
Enter password:
asdfgaskdasdahfjkfhjdjfjgdasgdas
Error!! Stack Values Does not match!!
Segmentation fault (core dumped)
$
$
```

Figure 5: Outputs of normal and buffer overflow exploit testing.

Code Implementation of shadow stack

```
#include ``pin.H''
#include <iostream>
#include <fstream>
#include <stack>
/* ================================================================== */
/* Global Variables                                                   */
/* ================================================================== */
std::ofstream TraceFile;
std::stack<ADDRINT> shadow;
static bool isOkay = true;
/* ================================================================== */
/* Commandline Switches                                               */
/* ================================================================== */
KNOB<string> KnobOutputFile(KNOB_MODE_WRITEONCE, ``pintool'', ``o'',
                            ``shadowstack.out'', ``specify trace file name'');
/* ================================================================== */
/* Print Help Message                                                 */
/* ================================================================== */
INT32 Usage()
{
    cerr << ``This tool produces a call trace.'' << endl << endl;
    cerr << KNOB_BASE::StringKnobSummary() << endl;
    return -1;
}
string invalid = ``invalid_rtn'';
/* ================================================================== */
/* Handling if instruction is of INS_IsCall type.                     */
/* ================================================================== */
VOID  do_call(ADDRINT target, ADDRINT ret, THREADID tid)
{
    //Writing to the TraceFile
    TraceFile << ``Return: '' << ret << endl;
    //Pushing the ret(IARG_RETURN_IP) to the stack
    shadow.push(ret);
}
/* ================================================================== */
/* Handling if instruction is of INS_Ret type.                        */
/* ================================================================== */
VOID  do_ret(ADDRINT instaddr, ADDRINT target, THREADID tid)
{
    //Writing to the TraceFile
    TraceFile << ``Return Adress: '' << target << endl;
    //Checking if the top of stack is same as target(IARG_BRANCH_TARGET_ADDR).
    if(shadow.top() == target)
    {
        //If they are the same pop the return address
        shadow.pop();
    }
    else
    {
        //If they are not same set isOkay as false and print the error and
        //write it to the TraceFile
        isOkay = false;
        std::cerr << ``Error!! Stack Values Does not match!!'' << endl;
        TraceFile << ``Error!! Stack Values Does not match!!'' << endl;
    }
}
```

```
/* ===================================================================== */
/* Trace Instrumentation Implementation.                                 */
/* ===================================================================== */
VOID Trace(TRACE trace, VOID *v)
{
    //Iterating over blocks
    for(BBL bbl = TRACE_BblHead(trace); BBL_Valid(bbl); bbl = BBL_Next(bbl))
    {
        //Getting the tail instruction
        INS inst = BBL_InsTail(bbl);
        //If isOkay is true
        if(isOkay == true)
        {
            //Checking if the instruction is INS_InCall type or INS_Ret type
            if( INS_IsCall(inst) )
            {
                //calling do_call analysis section
                INS_InsertCall(inst,IPOINT_TAKEN_BRANCH,AFUNPTR(do_call),
                  IARG_BRANCH_TARGET_ADDR,IARG_RETURN_IP,IARG_THREAD_ID,IARG_END);
            }
            else if( INS_IsRet(inst) )
            {
                //calling do_ret analysis section
                INS_InsertCall(inst,IPOINT_BEFORE,AFUNPTR(do_ret),IARG_INST_PTR,
                        IARG_BRANCH_TARGET_ADDR,IARG_THREAD_ID,IARG_END);
            }
        }
    }
}
/* ===================================================================== */
VOID Fini(INT32 code, VOID *v)
{
    TraceFile << ``# eof'' << endl;
    TraceFile.close();
}
/* ===================================================================== */
/* Main                                                                  */
/* ===================================================================== */
int  main(int argc, char *argv[])
{

    PIN_InitSymbols();

    if( PIN_Init(argc,argv) )
    {
        return Usage();
    }
    TraceFile.open(KnobOutputFile.Value().c_str());
    TraceFile << hex;
    TraceFile.setf(ios::showbase);
    string trace_header = string(``#\n''
                                 ``# Call Trace Generated By Pin\n''
                                 ``#\n'');
    TraceFile.write(trace_header.c_str(),trace_header.size());
    TRACE_AddInstrumentFunction(Trace, 0);
    PIN_AddFiniFunction(Fini, 0);
    // Never returns
    PIN_StartProgram();
    return 0;
}
```