Dynamic Program Analysis using Dynamic Binary Instrumentation (with Pin)

In this lab session you will practice with the dynamic instrumentation of binaries. Specifically, you are going to make use of the Pin [1] environment to program an analysis application for certain purposes.

To carry out this practice, you will have to deploy the virtual machine of the Windows operating system that is provided in OVA format (link available at the workshop website, `https://webdiis.unizar.es/~ricardo/sbc-2022/advanced-malware-analysis/`). This virtual machine is running an evaluation version of Windows 7, provided by Microsoft for evaluation of its commercial product *Internet Explorer 8*. If you are more curious, you can visit the Microsoft page where you will find these virtual machines for download (`https://developer.microsoft.com/en-us/microsoft-edge/tools/vms/`).

You will need to deploy the virtual machine on the lab computer (or on your computer) with the virtualization software of your choice (such as VirtualBox, VMWare, or others). It is strongly recommended that you take a first *snapshot* of the virtual machine's running state.

The software that is already installed is the following:

- 7zip version 17.00;
- Notepad++ version 7.5.2;
- Web browser *Mozilla* version 62.0;
- .NET 4.5.1 environment, English language;
- Microsoft Visual Studio Express 2012 (compiler `cl`);
- MinGW environment with `gcc` version (the Windows path is set to find it); and
- Python 3.7.0 compiler (the Windows path is also set to find it).

Regarding the Pin environment, Pin 3.5 version 97503 has been installed. You can find it in the root directory of the virtual machine (**the Windows path has not been configured with this directory, beware!**). The Pin online manual can be found here: `https://software.intel.com/sites/landingpage/pintool/docs/98579/Pin/doc/html/index.html`.

Other Windows binary analysis tools have also been installed, such as the `CFF Explorer` tool, the `PEiD` tool, and the `OllyDBG` tool (the latter with some interesting plug-ins). They are available in the "Herramientas" folder in the root directory.

1 Discovering Pin...

We are going to see step by step how to build a dynamic binary instrumentation analysis tool with Pin, also called a *Pintool*. As explained in the lectures, the programming language of a Pintool is C/C++.

A Pintool will always have a fixed structure. On the one hand, in addition to the libraries that our analysis tool needs (depending on the C functions you are using, they will be some libraries or others), we also need to include the library `pin.h`, which is the one that contains all the definitions and dependencies necessary for our tool to compile and form a DLL perfectly.

As you already know, the code that is executed first in C is the one that is defined in the `main` function. In this case, since the Pintool will be executed as a parameter of the Pin tool (we will see it later), we have to declare the function header `main` indicating that there are execution parameters:

```
int main(int argc, char * argv[])
{
    ....
}
```

Now, we need to initialize the Pin analysis engine from our Pintool. This call, provided by the Pin library, is `PIN_Init(argc, argv)`.

After this call, the instrumentation routines we are considering should appear in our main function. Remember that Pin allows us to instrument at different levels of granularity:

Low level vision

- *At the instruction level* (INS): Pin allows us to instrument the assembler instructions that are going to be executed in the machine's CPU.
- *At the basic block level* (BBL): Similarly, we can also instrument the basic blocks to be executed on the CPU. Remember that a basic block is a linear sequence of instructions ending in a flow change instruction. Also remember that a basic block has a single input and a single output.
- *At the trace level* (TRACE): Pin defines one more structure, the trace or also called *superblock*. Traces typically start at the destination of a jump that has been caught and end with an unconditional flow change instruction (for example, unconditional jumps, a call, or a return). They have a single input but can have multiple outputs. These traces are what is then actually broken down into basic blocks, giving rise to the structures mentioned above.

Program vision

It is necessary to call the function `PIN_InitSymbols()`, which is responsible for initializing the symbol table, before the call to `PIN_Init(argc, argv)` when working at the program level /image.

- *At the routine level* (RTN): Pin allows us to implement the routines (functions) that an image of a program has declared. According to the Pin documentation, exit instrumentation is not very reliable when there are queued call sequences or when function return statements cannot be reliably detected.
- *At the section level* (SEC): Pin allows us to go through an image at the section level (note that this is not instrument). Each of these sections can then be traversed at the routine level or at the instruction level.

- *At the image level* (IMG): Pin also allows us to instrument at the image level. Thus, we can execute code the first time a certain image is loaded. For example, we can inspect the sections that the image contains, what functions it declares, and even implement a particular routine.

System vision

- *At the process level*: Pin allows us to instrument all the processes that are generated from the beginning with the same current instrumentation tool.
- *At the thread level*: Pin also allows us to instrument each time a thread is created or terminated, among other options.
- *At the exception level*: Pin even allows us to control the context switches that happen in the instrumented program. Thus, we can detect if the context change is due, for example, to the execution of an exception. You can also define a function to be executed as the analysis tool's exception handler (note that it will not catch the exceptions of the application that is being analyzed, but of the analysis tool itself).
- *At the syscall level*: Pin gives us the necessary power to implement the application at the syscall level directly. At `https://software.intel.com/sites/landingpage/pintool/docs/98484/Pin/html/group__SYSCALL.html` you can find the list of all functions provided by Pin for this purpose.

Instrumentation routines are registered via callback, executed and instrumented the first time they are executed. Some of these routines are: `INS_AddInstrumentFunction`, `TRACE_AddInstrumentFunction`, `RTN_AddInstrumentFunction`, and `IMG_AddInstrumentFunction`. Depending on our analysis needs, we will have to choose one routine or another.

In addition to instrumentation routines, there are other callbacks, such as `PIN_AddFiniFunction`, `PIN_AddDetachFunction`. The `PIN_AddFiniFunction` function allows you to define a function that will be executed when the entire instrumentation process is finished. Note that this is not an instrumentation routine, since it does not insert new code. Pin allows for more than one completion function. For example, if you have used a text file in your analysis tool (or other dynamic memory objects), this function can be in charge of closing the file and freeing all this occupied dynamic memory. The `PIN_AddDetachFunction` function is equivalent, but is used when a tool has been made to instrument a currently running process.

After having inserted the instrumentation routine or routines that we want (and, optionally, having registered a last callback at the end of the analysis process), it will be necessary to tell Pin that we have already finished the configuration and can start the execution and analysis of the application.

Remember that Pin provides two methods of analysis:

JIT analysis. In this type of analysis, Pin creates a modified copy of the application directly on the fly, during the execution of the application itself. This modified code is what contains the instrumentation. So what happens is that the original code is never executed. Despite the fact that this analysis method provides a lot of flexibility to the analyst (we can do *almost* anything), as you can guess, it induces an overload in the execution time that sometimes becomes really excessive [2, 3].

Probe mode analysis. Here, what happens is that Pin modifies the code of the original application. Specifically, Pin will directly insert jump instructions that lead directly to execution of the instrumentation code at those defined instrumentation sites. This technique is known

as inserting *trampolines*. This analysis method introduces much less overhead; on the other hand, it is much less flexible: some of the instrumentation functions discussed here cannot be applied when running this analysis mode. If you use this mode at some point, you will have to consult the Pin manual if all the functions that you are going to use are compatible with this analysis mode.

The difference between one analysis or another lies also in the start function that is called: `PIN_StartProgram()` (JIT analysis) or `PIN_StartProgramProbe()` (Probe mode analysis). After this call, the program will never return: that is, all code defined below this call is unreachable code.

Pin will then start the analysis of the application to be instrumented. It will process the binary code of the application and upload parts of the code to a code cache. If instrumentation has been defined at the instruction level, it will evaluate the instructions to see if they need to be instrumented. If necessary, it will modify the code to be executed by inserting new code that allows the defined instrumentation function to be invoked. If not needed, the loaded instructions will be executed. For optimization reasons, the modified code is saved in a code cache with the aim of reusing it later (and thus avoiding the overload induced by a new instrumentation).

Pintool Compilation

The compilation of a Pintool can be done from the Visual Studio environment, if you have it installed, after configuring it correctly. The necessary steps for this configuration can be found explained here: http://blog.piotrbania. com/ (some of the build flags have changed with the latest versions of Pin, on the first build you will see a warning message which will help you debug this configuration).

As material for the practice, you have been given a Windows console command file[1], named «`compile32.bat`», which allows you to directly compile and link a source code file and generate a Pintool for a 32-bit system using the Microsoft Visual C++ compiler. As a parameter, you need the filename of the source code to compile and link (**without extension**). It will generate the Pintool as output in the same directory where the source code is located.

NOTE: you need to run the build script from an MS Visual C++ Console for the command console to recognize the Visual C++ tools (Windows Start button » All Programs » Microsoft Visual Studio 2012 » Visual Studio Tools » VS2012 x86 Native Tools Command Prompt).

1.1 Example: Counting instructions

Let's consider a simple example to get started. Listing 1 shows the typical code for a "Hello, world!" in C. As you can see, it only displays the string `"Hello, world!"` by calling `printf` and ends its execution. We are going to use dynamic instrumentation in order to see how many assembler instructions are needed to carry out this action (in pure assembler, a "Hello, world!" is 5 assembler instructions using software interrupts).

[1] File created and provided by Miguel Martín-Pérez, PhD student at the University of Zaragoza (November 2017).

Listing 1: File «hello_world.c».

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("Hello, world!\n");
6     return 0;
7 }
```

Compile Listing 1 with both `gcc` and `cl` (Microsoft Visual C++ compiler) to get two executable files that are semantically identical, but *slightly different* in its binary code. Name the generated binaries, for example, «`hello_gcc.exe`» and «`hello_cl.exe`». Since we are going to count how many assembly instructions are actually executed, we can also check which compiler *generates a more efficient binary*, understanding the measure of efficiency as *performing the intended action with fewer instructions*.

Let's create our Pintool. Create a source code file and name it «`inscount.cpp`». The code of the main function, considering what was explained at the beginning of this section, should be similar to:

```
#include <iostream>
#include "pin.H"

int main(int argc, char * argv[])
{
    // Initialize pin
    if (PIN_Init(argc, argv))
        return 1; // you can print here usage error

    // Register Instruction to be called to instrument instructions
    INS_AddInstrumentFunction(Instruction, 0);

    // Register Fini to be called when the application exits
    PIN_AddFiniFunction(Fini, 0);

    // Start the program, never returns
    PIN_StartProgram();

    return 0;
}
```

The instrumentation function, `Instruction`, must be defined before the `main` for visibility reasons (typical of structured languages). Its implementation is as follows:

```
VOID Instruction(INS ins, VOID *v)
{
    // Insert a call to docount before every instruction, no arguments passed
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)docount, IARG_END);
}
```

As you can see, the function `Instruction` defines that **before each instruction to be executed, the `docount` function is called**. The header of this instrumentation function, in the case of instrumenting at the instruction level, is always the same. The parameters received by the function must be placed between the name of the instrumentation function and `IARG_END`. As you can see, in this case the instrumentation function does not receive any parameters.

Regarding the analysis function `docount`, the code can be as follows:

```
// The running count of instructions is kept here
// make it static to help the compiler optimize docount
static UINT64 icount = 0;

VOID docount() {
    icount++;
}
```

As you can see, the only thing it does is increment a counter variable by one unit, which is initialized to zero and declared as global and static, and its data type is a 64-bit integer.

Finally, the completion function that we have previously defined as `Fini` would be in charge of displaying the value of the counter directly on the screen (it will be through the console where the Pintool is executed).

```
// This function is called when the application exits
VOID Fini(INT32 code, VOID *v)
{
    cout << "Count " << icount << endl;
    // This is equivalent to:
    // fprintf(stdout, "Count %d\n", icount);
}
```

The necessary code could have been implemented so that the writing was, for example, in a data file instead of on the screen. To compile the tool, you can run the following from a Visual C++ command console (explained above),:

```
compile32 inscount
```

This command should create a DLL file named ≪`inscount.dll`≫ which is the instrumentation tool you just developed. Let's try it! To run the analysis, you have to open a command console to run Pin, indicating both the Pintool and the program you want to instrument. For instance, to instrument ≪`hello_gcc.exe`≫:

```
C:\pin-2.14-71313-msvc11-windows\pin.exe -t inscount.dll -- hello_gcc.exe
```

After running, the program should show us on the screen (in addition to the string "Hello, world!") the following:

```
Count: 556146
```

As you can see, 556146 instructions of that simple program that we have instrumented have been executed. Now try to do the same, but this time instrumenting the program ≪`hello_cl.exe`≫. What is the result obtained? Contrary to expectations, it turns out that the Visual C++ compiler is creating a program that is more efficient, in terms of the number of instructions that are executed (the count for the binary created with VC++ is 317666). This number of instructions is taking into account not only the instructions of our program but also all the instructions of the library functions of the system that have been called. Also, this number can vary between runs for a variety of reasons, such as environment variables, results of system function calls, and so on.

However, this result is surprising if low-level programming is known. As additional files, in this lab session you have been provided with several versions of "Hello, world!" in assembler so

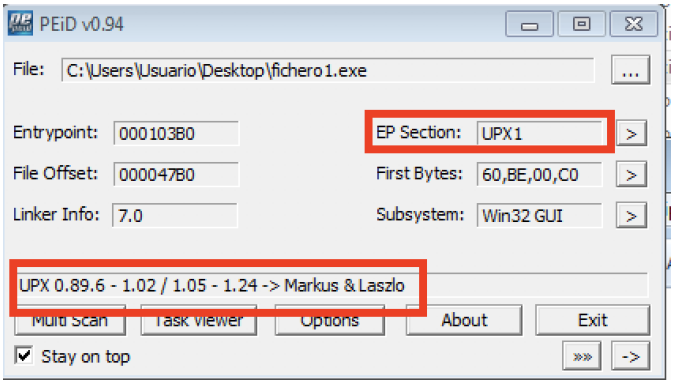

Figure 1: Analysis with `PEiD` of the program ≪`fichero1.exe`≫.

that you can see different possibilities (more optimal than the one that the compiler seems to generate for us). As you can see, the instructions do not reach ten in the worst case. That is, the overhead introduced by the compiler, in both cases, is (quite) considerable.

1.2 Analysis of Packed Binaries

We are now going to implement another slightly more complex analysis tool. Consider the program ≪`file1.exe`≫ provided in this lab session. This Windows program is a file that is compressed with `UPX`. This free and portable executable compressor supports different executable formats, including Windows. Its official website is `https://upx.github.io/`.

You can use the `PEiD` tool (or `RDG Packer Detector`, among others) to check how effectively this binary is detected as being protected with `UPX`. Remember that this tool is available in the "Herramientas" folder in the root directory. If you analyze the provided program, you will see something similar to what is shown in Figure 1. Notice that the binary is detected to be UPX-compressed by its signature (the detection of `PEiD` is based on signatures) as well as having a section name `UPX1`.

As you may remember from lectures, a binary compressor is going to decrypt (once or several times) the program during execution. To do this, certain memory areas will be overwritten to be later executed. Based on this reasoning, a Pintool can be developed to do the following:

1. Monitor all memory accesses.
2. Save a record of all memory accesses that correspond to a write.
3. Keep a record of all those memory addresses that have been executed after being written.

To develop this tool, instrumentation at the instructional level is needed. Given the extension of the Intel x86-64 ISA, it is unfeasible to think of developing a tool where each and every one of the possible instructions is analyzed in a concrete way. Therefore, you will surely be interested in knowing the following generic analysis functions given by PIN:

- `INS_MemoryOperandCount(INS ins)` **function**: given an `ins` instruction, it returns the number of memory operands it contains.
- `INS_MemoryOperandIsRead(INS ins, UINT32 memopIdx)` **function**: given a `ins` instruction and a `memopIdx` position, it returns true if the memory operand `memopIdx` is read.

- `INS_MemoryOperandIsWritten(INS ins, UINT32 memopIdx)` **function**: given a `ins` instruction and a position `memopIdx`, it returns true if the memory operand `memopIdx` is written.

You can find more information about these functions, as well as other generic analysis functions, on the web page `https://software.intel.com/sites/landingpage/pintool/docs/98484/Pin/html/group__INS__INSPECTION.html` provided by Pin.

Regarding the parsing insertion function commented above, it is convenient that you know that there is also a version that allows the analysis function to be executed or not, depending on whether the instruction being instrumented has a predicate and this is true (it runs) or not (it does not run). This function is `INS_InsertPredicatedCall`. You can find out more about it and the other instruction-level instrumentation features at `https://software.intel.com/sites/landingpage/pintool/docs/98484/Pin/html/group__INS__INSTRUMENTATION.html`.

Finally, you should be aware of the `IARG_MEMORYOP_EA` parameter, which allows you to access the effective address of the operand at the position you specify. For example, the code:

```
INS_InsertPredicatedCall(
                ins, IPOINT_BEFORE, (AFUNPTR) instrumentation_func,
                IARG_INST_PTR,
                IARG_MEMORYOP_EA, memOp,
                IARG_END);
```

inserts the `instrumentation_func` instrumentation function before the `ins` instruction, passing two parameters to it: the address of the instruction itself (`IARG_INST_PTR`) and the effective address of the memory operand at `memOp` location. In this case, the header of the called instrumentation function would be:

```
VOID instrumentation_func(VOID *ip, VOID *addr)
```

Remember that both parameters would be memory addresses. That is, if we wanted to know what values both parameters have, the following code could be incorporated into the body of the `instrumentation_func` instrumentation function:

```
fprintf(stdout,"Instruction at %p is making some stuff at %p\n", ip, addr);
```

To know the understanding of what you have just read, in this part of the session you are asked for the following assignment.

Assignment 1.

Given all of the above, you must develop a Pintool to monitor the memory addresses that are first overwritten and then executed during the execution of the program «`file1.exe`» and the program «`file2.exe`» (folder "programs' " of the auxiliary files of the lab session). Remember that you will need to instrument at the instruction level all memory writes, saving them, and then all the instructions that are executed, checking if you had them stored before. You can use whatever data structure you find most convenient to store the overridden instructions (simple lists, hash tables, etc.).
Knowing that the original entry point (OEP) of both binaries before they were compressed was the address `0x01006EB3`, *what is the number of false positives?* (In this context, consider a false positive to be a memory address that is first overwritten and then executed, but is NOT the OEP.) Remember to **focus only on instructions that are in the memory space corresponding to the section of code of the binary** (i.e., we are not interested in the instructions first overwritten then executed within the Windows libraries, if any). Also remember that in general you will be aware of the OEP in other programs that you will analyze, such as malware samples. Therefore, **you should not use the OEP value in your instrumentation program: you should only use it later (in the analysis of results) to calculate the number of false positives** (the false positives will be calculated by then as the number of addresses you have registered that first overwrite and then execute before the OEP is registered).

2 Analyzing More Complex Software

Let's now see how to instrument at a higher level. Specifically, we are going to take advantage of the power that Pin gives us to work with executable images (dynamic link libraries, DLLs) to be able to analyze the functions that an application executes.

Remember that on Windows, executables declare what functions they use (presumably) in a certain section, visible with tools like `CFF Explorer` (installed on the provided Windows virtual machine). The presence of certain functions in this section does not mean that the program actually executes them, but rather that the source code that generated the pogram had references to these calls. That is, imagine an unreachable piece of code in the source code that would make use of certain functions exclusively. These functions, when compiled, will be inside the import section. However, they will never be executed since the code is unreachable.

Similarly, remember that the absence of other functions does not mean that the binary does not use them: in Windows there are two methods of importing functions, either statically or dynamically. Static import is the one just described: the compiler looks at references to certain functions in the source code, and incorporates these references into the import table. When a dynamic import is used, it is the program itself that, when executing, will resolve the function addresses before executing them. This dynamic method is commonly used by malware to prevent a static analysis of the import table from revealing clues about its behavior.

Finally, one very important thing when working with Windows APIs is to respect the type nomenclature they use. In Section 2.2 you will learn how to replace Windows functions and see how it is necessary to define a C++ namespace relative to the Windows library, `windows.h`, and how it is necessary to prepend each data type with the appropriate namespace (for instance, `WINDOWS::DWORD`).

2.1 Routine Level Instrumentation

Consider Listing 2, which performs dynamic memory allocation and subsequent freeing. However, the code has a vulnerability because there is a pointer that has been freed twice. This vulnerability causes a bug in the heap manager that can lead to arbitrary code execution.

We are now going to develop an analysis tool that is capable of detecting this type of problem. When a DLL is loaded with the program being analyzed, the Pintool will look for possible references to the function `malloc` or `free`, and if found, will instrument them to execute certain code.

Specifically, the Pintool is going to save a list of memory areas that have been allocated (function call `malloc`). Then, every time a memory zone is freed (call to `free` function), it will check if such a memory zone is allocated.

A first possible solution (and useful to show the power of Pin during dynamic analysis) is to instrument the `malloc` function after its execution, since the memory zone will have already been allocated and we will know where the malloc zone is located; while the `free` function will be instrumented before its execution. In the first case we are interested in the value that `malloc` will return (remember the header of the function `malloc`, `http://www.cplusplus.com/reference/cstdlib/malloc/`), while in the second we are interested in the parameter that the `free` function receives (remember its header, `http://www.cplusplus.com/reference/cstdlib/free/`) .

Listing 2: File «`double_free.c`».

```c
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>

char* reserveMemory(int size)
{
    char *temp = (char *) malloc(size);
    return temp;
}

int main(void)
{
    /* Create an array for storing dummy data */
    char *c = reserveMemory(10);
    printf("(malloc) %p\n", c);
    c[0] = 5;

    char *c2 = reserveMemory(10);
    printf("(malloc) %p\n", c2);
    free(c);
    free(c2);
    free(c2); // double free
    return 0;
}
```

The complete code for this analysis tool is shown in Listing 3. **Look at the provided code and understand it**. Then compile and test it with the vulnerable code provided.

Listing 3: File «DoubleFreeDBA.cpp».

```
#include "pin.H"
#include <iostream>
#include <iomanip>
#include <algorithm>
#include <list>
#include <string.h>
#include <stdio.h>
#include <ostream>

list<ADDRINT> MallocAddrs;

VOID FreeBefore(ADDRINT target, ADDRINT inst)
{
   list<ADDRINT>::iterator p;
   p = find(MallocAddrs.begin(), MallocAddrs.end(), target);
   if ( (!(MallocAddrs.empty())) && (MallocAddrs.end() != p) ){
       p = MallocAddrs.erase(p); // Delete this from the allocated address list
   }else{   // We caught a Free of an un-allocated address
       cerr << "DOUBLE-FREE DETECTED: " << hex << target << " @" << inst << endl;
   }   // Using cerr is not a good practice, I do it oly for the sake of the example
}

VOID MallocAfter(ADDRINT ret, ADDRINT inst)
{
   // Save the address returned by malloc in our list
   if (ret != 0){
      list<ADDRINT>::iterator p;
      p = find(MallocAddrs.begin(), MallocAddrs.end(), ret);

      if (MallocAddrs.end() == p){ //not found
         MallocAddrs.push_back(ret);
         cerr << "Saving " << hex << ret << " in the address list @" << inst << endl;
      }else{
         // malloc address already in the list?!
         cerr << "already saved" << hex << " @" << inst << endl;
      }
   }else{
     cerr << "Malloc fail" << endl;
   }
}

// Instrument the malloc() and free() functions.
// note that there are malloc and free in the os loader and in libc
VOID Image(IMG img, VOID *v)
{
    cerr << "Hooking img: " << IMG_Name(img) << endl;

    // Find the malloc() function and add our function after it
```

```
49      RTN mallocRtn = RTN_FindByName(img, "malloc");
50      if (RTN_Valid(mallocRtn)){
51          // print function name
52          cerr << "Function name: " << RTN_Name(mallocRtn) << endl;
53          RTN_Open(mallocRtn);
54          RTN_InsertCall(mallocRtn, IPOINT_AFTER, (AFUNPTR)MallocAfter,
55                         IARG_FUNCRET_EXITPOINT_VALUE, IARG_INST_PTR,
56                         IARG_END);
57          // IARG_FUNCRET_EXITPOINT_VALUE function result, valid only at
                return instruction
58          RTN_Close(mallocRtn);
59      }
60
61      // Find the free() function and add our function before it
62      RTN freeRtn = RTN_FindByName(img, "free");
63      if (RTN_Valid(freeRtn)) {
64          // print function name
65          cerr << "Function name: " << RTN_Name(freeRtn) << endl;
66          RTN_Open(freeRtn);
67          RTN_InsertCall(freeRtn, IPOINT_BEFORE, (AFUNPTR)FreeBefore,
68                         IARG_FUNCARG_ENTRYPOINT_VALUE, 0,
                           IARG_INST_PTR,
69                         IARG_END);
70          // IARG_FUNCARG_ENTRYPOINT_VALUE int argument, valid only at the
                entry point of a routine
71          RTN_Close(freeRtn);
72      }
73  }
74
75  int main(int argc, char *argv[]){
76      // Initialize pin & symbol manager
77      PIN_InitSymbols();
78      PIN_Init(argc,argv);
79
80      // Register Image to be called to instrument functions.
81      IMG_AddInstrumentFunction(Image, 0);
82
83      PIN_StartProgram(); // Never returns
84
85      return 0;
86  }
```

2.2 Replacing Program Functions

Another possible solution is to *hook* the functions. That is, instrument the binary in such a way that when the program wants to execute the `malloc` or `free` function, our functions are actually executed. In them, you can call (if you want) the original functions of `malloc` and `free`, in order to effectively perform dynamic memory allocation and release so that the correct execution of the program is not broken.

To perform this *hooking* process, the Pintool code changes slightly. Let's do a simple example. Suppose you are interested in hooking the Windows function `Sleep`, which is responsible for sleeping the process that invokes it for a certain time indicated by parameter (in millisec-

onds). You can find the description of this API here: `https://msdn.microsoft.com/en-us/library/windows/desktop/ms686298(v=vs.85).aspx`. According to the MSDN documentation, the `Sleep` function header is:

```
void Sleep( DWORD dwMilliseconds );
```

Now let's see how the following instrumentation function is responsible for performing a replacement of the `Sleep` function to the *mySleep* function:

```
namespace WINDOWS{
    #include <windows.h>
}

void ReplaceWinAPI(IMG img, void *v)
{
    RTN rtn = RTN_FindByName(img, "Sleep");
    if(RTN_Valid(rtn)){
        PROTO proto = PROTO_Allocate(PIN_PARG(void),
                CALLINGSTD_STDCALL, "Sleep",
                PIN_PARG(WINDOWS::DWORD), //argument type of Sleep
                PIN_PARG_END()
                );
        RTN_ReplaceSignature(rtn, (AFUNPTR) mySleep,
                IARG_PROTOTYPE, proto,
                IARG_ORIG_FUNCPTR, // original function
                IARG_FUNCARG_ENTRYPOINT_VALUE, 0, // First parameter
                IARG_CONTEXT, // execution context
                IARG_END
                );
         PROTO_Free(proto);
    }
}
```

There are some important things you should know about the code above. First, in the function `PROTO_Allocate` it is defined how the function is: note that it follows the stdcall standard (you can see it in the function specification in the MSDN, the link has been given above) and that receives a parameter of type DWORD. Again, you can know this by reading the function header on the MSDN.

Then, the `RTN_ReplaceSignature` function of Pin is executed, which is responsible for replacing the original function `Sleep` with the one indicated, in this case *mySleep*. Pin is then told the argument list of this new function. Notice that it takes three arguments: (1) a function pointer that is the original function (i.e., the address of `Sleep` in this case); (2) the first parameter with which `Sleep` is called; and (3) the execution context. This execution context is useful to be able to consult or modify the current execution state (values of the CPU registers) in the analysis function.

Finally, the new function *mySleep* that replaces `Sleep` could look something like:


```
static VOID mySleep(AFUNPTR orig,
                        WINDOWS::DWORD dwMilliseconds, CONTEXT *ctx)
{
    CALL_APPLICATION_FUNCTION_PARAM param;
    memset((void*)&param, 0x00, sizeof(param));

    // Get from where Sleep is invoked
    ADDRINT *esp = (ADDRINT *)PIN_GetContextReg(ctx, REG_STACK_PTR);
    ADDRINT rtnAddr = *(esp);

    fprintf(stdout, "[Sleep @0x%08X] (0x%08x)", rtnAddr, dwMilliseconds);

    // Call the original Sleep function
    PIN_CallApplicationFunction(ctx, PIN_ThreadId(),
                CALLINGSTD_STDCALL, orig,
                &param,
                PIN_PARG(void), // this is the return value, void in this case
                // XXX Example: PIN_PARG(WINDOWS::HMODULE), &retVal,
                // XXX if would return an HMODULE
                PIN_PARG(WINDOWS::DWORD), dwMilliseconds,
                PIN_PARG_END()
                );
}
```

Notice that the header of this function coincides with what was expressed previously: 3 parameters, the first a function pointer, the second a DWORD, and the third a context.

Note that this context is exploited to retrieve the current value of the stack pointer register through the Pin functions. The goal is to know the address from which the `Sleep` function was called. For information purposes, it simply prints on the screen the address from where it was called and the parameter that was passed to the function.

Finally, the call `PIN_CallApplicationFunction` is used to call the original function `Sleep`, whose address is in the parameter *orig*. Note that in the event that there is an output parameter of the function, it should be indicated in this call. After the output parameter, both the type and value of each input parameter are indicated. In this case, remember that the `Sleep` function only receives a parameter of type DWORD.

Build yourself a Pintool that hooks the `Sleep` function as shown here and try it with Listing 4. Does it work?

Listing 4: File «`example_sleep.c`».

```
1  #include <stdio.h>
2  #include <windows.h>
3
4  int main(void)
5  {
6      // Naif example using Sleep
7      printf("I'm gonna sleep for a while...\n");
8      Sleep(10);
9      printf("Nice nap time, but still tired :(. See you later!\n");
10     Sleep(0xBABE);
11     printf("All right, ready to work now!\n");
12
13     return 0;
14 }
```


Assignment 2.

Following the new hooking scheme presented, modify the previously developed Pintool to replace the `malloc` and `free` functions with your own.

2.3 Password Capture

One last challenge is proposed to you to finish this lab session. You have been provided with a program file named «`licenseme.exe`» (MD5 hash: `bc1e2b9a1c104d114421130526693a43`). This program requires a string parameter, which it checks against a string encoded inside it. On success, it writes the string `"YOU WIN!"` to the screen. If not, it doesn't write anything.

Your goal is to find out the correct parameter **using dynamic binary analysis techniques, and especially dynamic instrumentation**. It may be helpful to take a look at the functions that the program declares to use. (Let me remind you that functions from the `glibc` library [like `strcmp`, `strcat`, `strcpy`] slightly change their name in MSVC-compiled binaries [e.g., `lstrcmp` , `lstrcat`, `lstrcpy`].)

Assignment 3.

Build two Pintools that allow you to find out the string parameter required to get the message `"YOU WIN!"` written on the screen, using the two methods explained above.

References

[1] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[2] Ricardo J. Rodríguez, Juan Antonio Artal, and José Merseguer. Performance Evaluation of Dynamic Binary Instrumentation Frameworks. *IEEE Latin America Transactions (Revista IEEE America Latina)*, 12(8):1572–1580, December 2014.

[3] A. Skaletsky, T. Devor, N. Chachmon, R. Cohn, K. Hazelwood, V. Vladimirov, and M. Bach. Dynamic Program Analysis of Microsoft Windows Applications. In *IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, pages 2–12, March 2010.

