CSE 509 Assignment 3: Binary Instrumentation

1 Background

Binary instrumentation provides a versatile mechanism for a number of security-related applications, including software hardening and exploit mitigation, sandboxing, malware analysis, and vulnerability detection. In this assignment, you will use Pin, a binary instrumentation system that we discussed in class. Pin is developed and supported by Intel. The projects seems to be quite active, as shown by recent software releases. The primary web page is

https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

The page includes a number of resources, including a tutorial presented at CGO 2013, a leading conference on compiler backends.

Please go over the tutorial carefully. Plan on spending at least a couple of hours on it. The tutorial includes examples that are very similar (if not identical) to some of the simpler exercises below.

In addition to the examples in the tutorial, also pay attention to "analysis routines" and "instrumentation routines." The former get executed all the time, while the latter get executed just the first time a basic block is executed. So you want to make the analysis code as fast as possible by doing as much as possible in the instrumentation code. This is similar to a compiler performing more work at compile time in order to reduce the computational effort at runtime.

2 Warmup (35%)

Most of these are examples from the Pin web page or the tutorial mentioned above.

- Implement a Pintool for counting the number of basic blocks executed by a program.
- Implement a Pintool for wrapping calls to `malloc`. In the wrapper, implement code to keep track of the number of calls made to malloc and the total amount of memory allocated.
- Implement a Pintool to count (a) the number of direct control flow transfer instructions executed by a program, and (b) the number of indirect control flow transfers executed by a program. Ensure that you take care of all types of branches, including returns.

All of the information maintained by your instrumentation should be printed when the instrumented program exits.

You should start by testing your program on a simple `hello_world` program, and then move on to more complex programs, e.g., `/bin/ls`, `gedit` and `Firefox`. Ideally, your instrumentation will work on arbitrary programs, but if it does not, please include documentation on the programs it is known to work with, and justify why it fails to work on others.

3 Security Applications (65%)

3.1 System Call Interception

We have discussed two approaches for system call interception: ptrace is secure but incurs a heavy performance overhead, while library interception is fast but insecure *and* incomplete. An alternative we explore in this project is system call interception using dynamic binary instrumentation. Such an approach can be secure, and can be fast if the underlying instrumentation platform itself is fast.

To intercept system calls, you need to examine every basic block at the time of instrumentation to identify any `int 0x80` instructions. You can then insert instrumentation right before and after this instruction. The instrumentation preceding the instruction can obtain information about the system call and arguments, and then invoke appropriate actions, such as logging the system call, or enforcing a policy. Note that, as in the case of

ptrace, the system call number will be in the `eax` register, and the arguments in `ebx`, `ecx`, etc. In addition, some arguments will be pointers, which need to be dereferenced in order to obtain their value. This step is also similar to ptrace, except that your instrumentation can directly dereference these pointers. Instrumentation can be added after `int 0x80` instruction to obtain the return value of system calls, or to modify the return value.

Your instrumentation should implement one of the following functions:

- *btrace:* Implement something similar to strace using binary instrumentation. Like strace, btrace will print the system call name, argument values (including file names) and return values/parameters. Make sure that you cover the most important and common system calls. In terms of arguments, make sure that you capture, at a minimum, all integer and string-valued arguments. The invocation of btrace should be similar to strace, i.e., it will be followed by an arbitrary command name and arguments.
- *bbox:* Implement a sandbox for all file-name based system calls. The sandbox policy is specified in a file that has four columns:
  - *match type* is one of `exact`, `prefix` or `suffix`.
  - *name* is a string to be used in the match operation.
  - *action* is one of `allow`, `deny`, or `redirect`.
  - *newdir* is present only when the *action* is `redirect`. In this case, a copy of the original file is made into `newdir` if it does not already exist there. The file name argument is modified to operate on this copy.

  For any file-name based operation, your instrumentation should match it against the rules in the policy file. The first matching rule should be used, and the corresponding action taken. The invocation of *bbox* is of the form `bbox <policyFileNme> <cmd>`, where `<cmd>` is an arbitrary command (i.e., a command and its command-line arguments.

For both applications, watch out for the possibility that the instrumentation code itself uses system calls. If this happens, your program may go into an infinite loop: interception of the original system call results in the invocation of another system call, which in turn is intercepted, causing yet another system call and so on. One way to avoid this is to ensure that the interception code only invokes system calls that are themselves not intercepted. For instance, it is possible to limit the interception code to use just `mmap` and `sbrk`[1]. For `btrace`, you can store the results in a memory buffer, and write them to standard out when the program exits. You can set a flag during this step so that your interception code can disable itself.

3.2 Stack Use Analysis and Stack Pivoting Detection

This application has two parts: in the first part, you develop a tool to compute the mazimum stack size reached during execution. The second part is concerned with the detection of stack pivoting, a key step in ROP. Note that ROP attack requires an attacker to control the contents of the stack. The easiest way to achieve this is to execute an instruction that changes the SP to point to a data buffer that holds attacker-provided data. This requires loading the SP with the value of another register, or the contents of memory. Such an operation is called stack-pivoting.

Implement a Pintool that measures the maximum stack used by a program. One way to do this is to instrument all call instructions, as well as instructions that load the stack pointer, say, `leave` or an instruction that moves the contents of another register (or memory) into ESP. You can omit instructions that change the SP by a constant amount — this means that your answer may be off by a small amount, but that is acceptable if you are interested in understanding approximate stack usage of your program.

Your instrumentation can keep track of the minimum ESP value, from which the total stack use can be computed. (It is sufficient if your instrumentation works for single-threaded programs.)

To debug your instrumentation, write a simple recursive C program that you can use for testing, e.g., a program that takes a number x as a command-line argument, and computes the sum $1 + 2 + 3 + \cdots + x$ recursively. Once debugged, you should use the instrumentation on arbitrary binaries, and check that it works. (There is no fool-proof way to check that it worked on arbitrary programs, so you should test it on smaller programs that you have some understanding of.)

[1] These arise due to memory allocation and hence can be hard to avoid.

Extend your instrumentation for stack use analysis to detect stack pivoting. Note that functions, when they return, will load ESP from EBP. There may be other benign use cases as well, e.g., during signal delivery or thread context switching. You do not have handle the last two cases, but make sure that you think through other possible benign use cases, and handle them correctly. "Handling correctly" does not mean that you simply accept arbitrary changes to ESP before return; instead, you should implement some means to check that the value loaded to ESP is legitimate, e.g., the new ESP value is greater than the old ESP, and less than the base of the stack.

Test your instrumentation using a test program that loads ESP from other registers and memory. Also ensure that it does not produce false alarms on legitimate programs. (If there are exceptions, describe them and justify them.)

3.3 Shadow Stack

Implement a shadow stack using Pin. The following paper contains a discussion of shadow stack instrumentation in Section 4.4, together with instrumentation-level details.

http://seclab.cs.sunysb.edu/seclab/pubs/vee14.pdf

Alternatively, you can see the following paper which uses Pin for shadow stack instrumentation. The entire paper is concerned with this, which means that it will contain an in-depth discussion. (This may be a feature or a bug depending on how much background you already have, and how much time you want to spend n reading the paper.)

https://download.hrz.tu-darmstadt.de/media/FB20/Dekanat/Publikationen/TRUST/ropdefender.pdf

4 Deadlines, Teams and Submission

This assignment can be completed as individuals or by groups of two students. If you plan to pair up, you need to identify your partners as soon as possible.

Individuals need to submit solutions to 2 of the 3 warm-up problems, and 1 of the 3 security applications. Individuals solving all three warmups are eligible to receive 7 bonus points. In addition, they can receive 25 bonus points if their security application works without significant limitations, such as the inability to support multi-threaded applications, failing on large applications, failure to support all relevant system calls etc. Finally, individuals implementing a second security application can receive 25 bonus points.

Team submissions should solve all of the warmups and one security application. Teams may receive up to 15 bonus points for implementing a second security application. They can also receive 15 bonus points if their application implementations work without significant limitations, such as the inability to support multi-threaded applications, failing on large applications, failure to support all relevant system calls, etc.

Do not copy code from anywhere except the Pin tutorials mentioned above. All other forms of copying will be considered cheating, and can result in an F-grade. We will actively use plagiarism detectors to detect instances of cheating. In addition, we may ask you to meet us to explain the code.

Your submission for this part will be a tgz file that we should be able to untar on Linux. It should include appropriate Makefiles and READMEs so that we can build and run your code without having to think about it. Your code should build with "make all." It should work within the 32-bit VM that was given to you for the first programming assignment. Your README must specify the location and version of Pin you used.

Please make sure that your code is documented. We plan to review your code. The points you receive will be based on the quality of your code, as well as how well it works.