Dragon Star 2017 Course : Program Analysis and Software Security Project

Input Provenance Tracking Tool

Contents

1 Part 0 : Setup environment

First, you need to download and install Valgrind. Valgrind 3.12.0, the latest version, can support Mac OS X 10.10 but it is highly advised to use Linux. In order to create a new Valgrind tool you can follow the instructions in `http://valgrind.org/docs/manual/manual-writing-tools.html`. The project template already contains the necessary files.

- Download Valgrind 3.12.0

```
$ curl -O http://valgrind.org/downloads/valgrind-3.12.0.tar.bz2
$ tar xjvf valgrind-3.12.0.tar.bz2
```

- Download project template.

```
$ cd valgrind-3.12.0/
$ curl -O https://www.cs.purdue.edu/homes/kim1051/valgrind/project.tar.gz
$ tar xzvf project.tar.gz
```

- Configure and build Valgrind.

```
$ sh autogen.sh
$ ./configure --prefix=‘pwd‘/inst
$ make
$ make install
```

- Disable address space layout randomization.

  If your linux kernel supports address space layout randomization, you need to disable it to get similar results at different executions. You need to repeat this step every time you reboot your machine.

```
$ echo "0" > /proc/sys/kernel/randomize_va_space
```

2 Part 1. memtrace: Memory read/write tracking tool

2.1 Project Description

In this part, you will build a memory read/write tracking tool with Valgrind. Your tool will detect every memory operations at runtime and dump the address being accessed.

Output Format At every memory read/write operation, your tool should print the address in the following format:

```
[R/W] 0x####
```

- The first column indicates whether the operation is read or write.
- The second column shows the memory address of the operation.

2.2 Instructions

- The template code is in directory memtrace/
- Create helper functions mt_log_load() and mt_log_store() which will take the address of the accessed memory and print the address and the access type(R/W) at runtime.
- Read mt_instrument() function in the template and identify the locations where you should insert calls to helper functions.

  Memory read operation is represented by Load expression and memory write operation is represented by Store statements. For the basic understanding of the structure of the instrument function refer to slide 13 in [1] and Sec. 4.1.

- At load expression, insert a call to mt_log_load() by using unsafeIRDirty_0_N() function. Refer to the slide 14 and Sec. 4.2.

- At store expression, insert a call to mt_log_store().
- Execute the memtrace and check the results using the examples in the next section.

  ```
  $ valgrind --tool=memtrace ./ex1
  ```

2.3 Examples

Example 1 Code and the binaries are in examples/ directory.

```
int a, b;
int main(void) {
    a = 10;
    b = a + 20;
}
```

Expected output 1 Note: your output should include the following items but may also have other items.

```
W 804a020
R 804a020
W 804a01c
```

Example 2 Code and the binaries are in examples/ directory.

```
int a, b;
int array[] = {1,2,3,4};
int main(void) {
    for (a = 0; a < 3; a++)
        b += array[a];
}
```

Expected output 2 Note: your output should include the following items but may also have other items.

```
W 804a030
R 804a030
R 804a030
R 804a018
R 804a02c
W 804a02c
R 804a030
W 804a030
R 804a030
R 804a030
```

```
R 804a01c
R 804a02c
W 804a02c
R 804a030
W 804a030
R 804a030
R 804a030
R 804a020
R 804a02c
W 804a02c
R 804a030
W 804a030
R 804a030
```

3 Part 2. provenance: Input provenance tracking tool

3.1 Project Description

In this project, you are asked to build a provenance tracking tool for Valgrind. Provenance tracking is a process of marking and tracking input data in a program at runtime in order to identify all dependencies on those values at certain execution points. This type of dynamic analysis is becoming increasingly popular in the context of software testing, debugging and system security.

Provenance sources Provenance sources are initial entry points of information to your program. Depending on the application context, many different types of sources could be defined. It includes standard input (stdin), files , and network data, etc. In this project, we choose only the standard input (stdin) and files as provenance sources. You must devise a forward dynamic algorithm and maintain a proper data structure to keep track of data provenance in memory locations and registers. The goal is to trace back all provenance targets to the specified provenance sources.

Provenance Targets We consider program variables or function pointers as provenance targets. Therefore, we are interested in finding their dependency on the specified provenance sources.

Output Format The output trace-file must be a list of log records in the following format:

```
0x#### :  [0x####:value] ...
```

- The first column specifies the memory address of a provenance target. You have to repeat the log records to cover all provenance targets.

- The second column shows whether this memory location is dependent on a provenance source due to a data dependency DD or control dependency CD or both. This field is mandatory for those who work in teams.
- The remaining columns show pairs of memory addresses of provenance sources and their corresponding values. These are the memory addresses that keep the inputs from stdin or files.

3.2 Instructions

- The template for the part 2 is in provenance/ directory.
- Design your data structure to hold set of input provenances. You can either use set data structure provided in the Valgrind (Sec. 4.5) or your own set implementation.
- Implement 2-layer shadow memory as in slides 6-9. Also refer to slides 7 for shadow temp variables and shadow registers.

  You don't need to use shadow register function provided by Valgrind. Instead, you can create an array for shadow registers similar to shadow temp variables.
- Complete pv_post_call() which will be called after a system call is executed. Please refer to the slide 16.

  For a read system call, args[1] denotes the address of the buffer and args[2] denotes the size of the buffer. After a read system call is executed, your tool should mark each byte of the buffer as a provenance source.
- Create helper functions to copy the set of input provenances between memory, temp variable and registers.
- Insert calls to the helper function in pv_instrument().

  For example, in a store statement if the data is RdTmp expression you should copy the set of input provenances from the shadow temp variable to the shadow memory. Complete all switch/case blocks in pv_instrument().
- At the store statement, print the set of input provenances for the target address.

3.3 Examples

Example 3 Code and the binaries are in examples/ directory.

```
int a, b, x, y, z;
int main(void) {
    read(0, &a, sizeof(int));
    read(0, &b, sizeof(int));
    x = 10 + a;
```

```
    y = x - b;
    z = y - a;
}
```

Expected output 3 Note: your output should include the following items but may also have other items.

```
0x804a02c : 0x804a030:4
0x804a034 : 0x804a030:4 0x804a024:0
0x804a028 : 0x804a030:4 0x804a024:0
```

Example 4 Code and the binaries are in examples/ directory.

```
int a, b, x, y, z;
int main(void) {
    read(0, &a, sizeof(int));
    read(0, &b, sizeof(int));
    x = 10 + a;
    y = x - b;
    x = 20;
    z = x - b;
}
```

Expected output 4 Note: your output should include the following items but may also have other items.

```
0x804a02c : 0x804a030:4
0x804a034 : 0x804a030:4 0x804a024:0
0x804a02c :
0x804a028 : 0x804a024:4
```

4 Useful resources

4.1 VEX IR

Please refer to [1] for the basic structure of Valgrind. Valgrind translates machine instructions into VEX IR which has 14 types of statements and 14 types of expressions. During this project, you only need to handle 4 statements, IMark, Put, WrTmp and Store statements, and 8 expressions, Get, RdTmp, Load, Const, Unop, Binop, Triop and Qop expressions. In the following paragraph, the statements and expressions are briefly explained. You can see more details in valgrind-3.12.0/VEX/pub/libvex_ir.h.

- IMark statement specifies the beginning of a machine instruction. In order to start tracing from main function, refer to slides 13-14.
- Put statement stores a value into a register. `st->Ist.Put.offset` denotes the register number and `st->Ist.Put.data` is the value expression. The value expression should be either Const or RdTmp expression.
- Store statement writes a value to a memory location. `st->Ist.Store.addr` is the memory address expression and `st->Ist.Store.data` is the value expression. Both the address expression and the value expression should be either Const or RdTmp expression.
- WrTmp statement writes a value into a temp variable. All operands in VEX IR are either Const expression or a temp variable. VEX IR is SSA form and a temp variable is written only once inside a super block. `st->Ist.WrTmp.tmp` denotes the temp variable number and `st->Ist.WrTmp.data` is the value expression. The value expression can be any expression, but its operands should be either Const or RdTmp expressions.
- Get expression reads a value from a register. `expr->Iex.Get.offset` denotes the register number.
- RdTmp expression reads a value from a temp variable. `expr->Iex.RdTmp.tmp` denotes the temp variable number.
- Load expression reads a value from a memory location. `expr->Iex.Load.addr` is the memory address expression.
- Const expression denotes a constant.
- Unop, Binop, Triop and Qop denotes unary, binary, ternary and quaternary operations respectively.

4.2 Insert a function call statement

To insert a function call into a super block, `unsafeIRDirty_0_N` function is used. The following is an example that inserts a call to helper_function().

```
void VG_REGPARM(0) helper_function() {
    /* do something at runtime */
}

IRSB* mc_instrument(...)
{
    ...
    IRExpr** argv = mkIRExprVec_0();
    IRDirty* di =
        unsafeIRDirty_0_N(0, "helper_function",
            VG_(fnptr_to_fnentry)(&helper_function),
```

```
                argv);
    addStmtToIRSB(sbOut, IRStmt_Dirty(di));
}
```

The first argument is the number of register parameter and it should match VG_REGPARM(n) of helper_function. It is safe to use '0' for both unsafeIRDirty_0_N and VG_REGPARM. The second argument is the name of the called function and third argument is the address of the called function. The forth argument is the argument list that will be passed to the helper_function. You can use functions from mkIRExprVec_0() to generate a zero-length argument list to mkIRExprVec_9() to generate an argument list of length 9.

In order to provide an constant value instead of VEX IR expression, you can use mkIRExpr_HWord(). The function takes a word as an parameter and returns a VEX IR expression that will be evaluated to the value of the parameter at runtime. Please refer to slide 14.

4.3 Standard C functions

Memory allocation To allocate and free memory dynamically, use VG_(malloc) and VG_(free) instead of malloc and free.

```
void* buffer = VG_(malloc)("buffer", size);
VG_(free)(buffer);
```

The first argument of VG_(malloc) is a string that can identify allocation point when debugging. The second argument is the allocation size.

Printing To print information to screen, VG_(printf) function can be used. The function is similar to printf except it uses standard error.

To use a file, you can use VG_(fopen), VG_(fclose), VG_(fprintf) functions. Refer to the project template for an usage example of VG_(fopen).

4.4 Sample Valgrind tool

memcount in the project file is a small Valgrind tool that counts number of memory read/write operations executed during runtime. In mc_instrument(), tool inserts a call to mc_load() and mc_store at load expressions and store statements respectively. mc_load() function is called every load expressions and counts the number of executed memory read operations at runtime. mc_store() function similarly counts the number of executed memory write operations.

4.5 Valgrind OSet datastructure

Set data structure is provided by Valgrind. The following code shows basic usage of the set and you can find more details from include/pub_tool_oset.h

```
/* Create a set whose elements are word */
OSet* set = VG_(OSetWord_Create)(VG_(malloc), "set", VG_(free));

/* Insert a word to the set */
/*   if the value is already in the set, it will cause a failure */
VG_(OSetWord_Insert)(set, (UWord)value);

/* Check if a word exists in the set */
if (VG_(OSetWord_Contains)(set, (UWord)value))
    VG_(printf)("exists");

/* Remove a word from the set */
if (VG_(OSetWord_Remove)(set, (UWord)value))
    VG_(printf)("removed");

/* Iterates over the set */
UWord value;
VG_(OSetWord_ResetIter)(set);
while ( VG_(OSetWord_Next)(set, &value) ) {
  /* do something with value */
}

/* Destroy the created set */
VG_(OSetWord_Destroy)(set);
```

Note: The OSetWord can only holds words (32/64bit integers) as elements. During the part 2, you can create an array that can hold provenance sources including their address, value and other information and you can maintain only the index to the array in the set.

References

[1] Implementing information flow system on Valgrind. `http://www.cs.purdue.edu/homes/xyzhang/spring17/5-slicing-IFS.updated.pdf`.