Shuntaint: Emulation-based Security Testing for Formal Verification

Bruno Luiz

`ramosblc@gmail.com`

Abstract. *This paper describes an emulated approach to collect traces of program states, in order to verify formally that these traces belong to the algorithm accepted by the provided graph for the finite state machine (FSM) specification. Shuntaint can attack most types of erros which allow the execution of arbitrary code. Exploring program states for security testing in the server deamons. This approach allows to detect entry points that cause the memory corruptions to be reached during exploration of states at specific moments of execution using network requests.*

1. Introduction

Software-based vulnerabilities can have serious consequences. Currently, projects tha focus bug-finding tries automatically analyzes bugs behavior. These tools and techniques provides have different tradeoffs. Many of these tradeoffs differs from analysis techniques. Valgrind [9] is a *dynamic binary instrumentation* (DBI) framework that makes it easy to invoke your *dynamic binary analysis*.

The analysis is performed dynamically using a *dynamic binary translation* for emulation of the CPU. A sequence of super blocks[1] is being translated to an intermediate representation (IR), which can be managed using instrumentation code for execution flow modification.

Dynamic binary translation can detect memory regions with translated code that are modified and is a very useful technique for security testing and analysis.

This paper describe Shuntaint that is a proof of concept implementation of valgrind tool. It collection low-level information during simulated program execution to security testing. Instead of use undefinedness propagation techniques, which track taintedness, this tool allows prove models that lead to an error using *passive testing*. In addition, memory access checks performed by computer program during a certain time are caputured for observing of logical properties satisfy a formal specification based on Common Weakness Enumeration (CWE) [1].

Shuntaint uses *reverse tainting analysis* to trace back to the sources to determine all entry points that are related to the error. At various points during instrumentation are performed code tracing on the way of achieving execution flow modification in fact trigger an error. Recents work, such as ptrcheck [10], track accesses of pointers in memory

[1]Represents from 1 to perhaps 50 instructions.

to detect invalid accesses. However, has differences of scheme associates of approach. Shuntaint approach is to keeping track a set of program states reachable to trigger error.

State-of-the-art tools integrate formal verification engines with simulation in a seamless way. Using formal methods [5], especially model checking, to prove the correctness of the algorithms. Shuntaint use VEX that is an architecture-neutral IR for formula generation. Futhermore, formal verification made easy for a wide range of erros which uses VEX IR for automatically analyze behavioral properties.

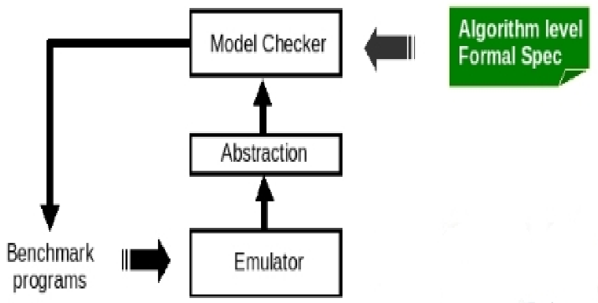


Figure 1. The emulated approach has 4 steps. First, several sets of benchmark programs being emulated to capture program states. Second, extract operations of IR that exhibits an error. Third, comparing critical program states for instrument the trigger. Finally, develop a model checking algorithm.

2. VEX IR

Valgrind is a memory debugging tool that allows monitors common problems in memory usage. Valgrind uses VEX to perform code analysis and transformation. Valgrind does emulation of a computer program using *just-in-time translation* to VEX IR which allow adds instrumentation code. The machine that is emulated (guest machine) instruments blocks of machine code (basic blocks) that is stored in blocks of memory (guest state).

Guest machine contains an sequence of basic blocks (IRSB). Each IRSB contains a list of statements (IRStmt) with side effects:

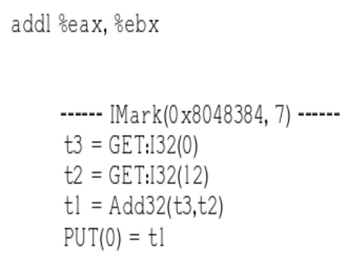
- storing a value to memory
- assigning to a temporary variable

and one IRStmt may have expressions (IRExpr) without side effects:

- arithmetic expressions
- loads from memory

In addition, a type environment represent a temporary value (IRTemp) present in the IRSB. A type (IRType) indicates the size of a value for each IRTemp. To operate on guest state, store (PUT) and load (GET) writing and reading respectively from guest state into a IRTemp to guest machine registers. These registers are characterised into the guest state by one offset that are array of bytes, which represent the memory. The representation of CPU state is accessed by one byte offset for an architecture.

The valgrind core hands the result back of instruments/transforms code blocks to machine code. The VEX coordinates instrumented code to a platform-independent. The IR statement (IMark) indicates translation at instrumentation-time.



```
addl %eax, %ebx

------ IMark(0x8048384, 7) ------
t3 = GET:I32(0)
t2 = GET:I32(12)
t1 = Add32(t3,t2)
PUT(0) = t1
```

Figure 2. Translation of x86 addl instruction at address 0x8048384 to VEX IR.

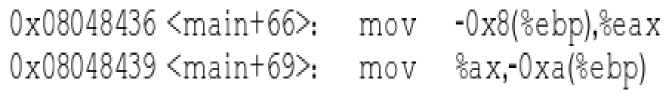
2.1. Changing memory states

Memcheck [11] is a valgrind tool to detect memory problems that implement *dynamic taint analysis*. Each value in memory is associated their own *shadow bits* marked as accessible (valid-value). The valid-value bits are checked, if a validity of location (valid-address) not may be accessed, an error is emitted. Ptrcheck is different of mencheck. The mechanism detect invalid uses of pointers of heap, stack and global arrays. The error is reported when the address is accessed out of bound of an associated block.

Shuntaint affect the behaviour of the program which basically allows to write an arbitrary value to an arbitrary address, which can lead to system security compromise. VEX compile these values to machine code executing in memory of real CPU forcing occurence of error.

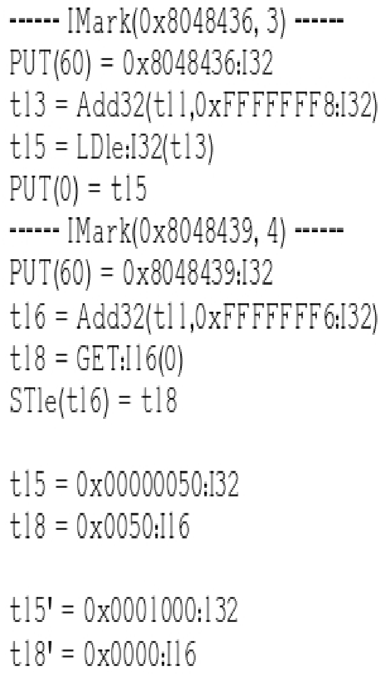
Shuntaint approach checks states of a given program to generate instrumentation code into a guest machine at particular points of the guest state for modeling of bit-vector array that eventually lead to the error. Note that if the code inserted or data manipulated by the guest machine is not visible to the program being tested.

A disassembly of binary shows where to write *taintedness bits* for catching logic error in an intermediate form.



```
0x08048436 <main+66>:    mov    -0x8(%ebp),%eax
0x08048439 <main+69>:    mov    %ax,-0xa(%ebp)
```

The emulator has support for tracking memory access to detect when the register change. In Figure 3, the amount of instrumentation code submitted back to the real CPU exceeds the range of the variable that cause an forced integer overflow, which is useful for automatically analyze the functional behaviour of the code for specific user data or functionality can be bypassed.



```
------ IMark(0x8048436, 3) ------
PUT(60) = 0x8048436:I32
t13 = Add32(t11,0xFFFFFFF8:I32)
t15 = LDle:I32(t13)
PUT(0) = t15
------ IMark(0x8048439, 4) ------
PUT(60) = 0x8048439:I32
t16 = Add32(t11,0xFFFFFFF6:I32)
t18 = GET:I16(0)
STle(t16) = t18

t15 = 0x00000050:I32
t18 = 0x0050:I16

t15' = 0x0001000:132
t18' = 0x0000:I16
```

Figure 3. a PUT operation write a 32-bit value to memory in guest offset 0 that corresponds to the guest register eax. Read a 16-bit value from guest offset 0 and load from memory into IRTemp t18.

3. Dynamic Memory Graph

The translation of block of code can catch a range of logic and intersect possible behavior with invalid behavior through VEX IR. The intrumentation need for each *memory referencing instruction* that hold at a point of executiont create a *model memory as a graph* [7] in which each memory location is associated with a property to indicate whether the data in this location are derived from user input.

To build a memory graph $G = (V, E, root)$ selected areas of the guest state are a set V of vertices selected, for each value in memory becomes a set of vextex $\upsilon \in V$ has the form $\upsilon = (val, tp, addr)$, standing for a value *val* of type *tp* at memory address *addr* and references between values become a set E of edges between these vertices, and $\epsilon \in E$ has the form $\epsilon = (\upsilon_1, \upsilon_2, op)$, where υ_1, υ_2 e V are the related vertice. Each operation *op* is a function λx.B that takes the expression of υ_1 to construct the expression of υ_2.

A memory graph encompasses the entire program state, it can be used to determining references whether a property violation can occur.

3.1. Stack Tracking

A new stack frame consists of the return, the frame pointer, and the local stack variables. The area from stack pointer up to the base of the stack is marked as accessible. When the user input data access the memory location are perform memory operations. These memory operations are extractor of program states within a context that holds all the variables events.

Let *extract*(*parent*, *op*, *G*) be a procedure the takes the name of a parent and an operation *op* and extract the element *op*(*parent*), adding new edges and vertices to the memory graph *G*.

A memory *load* operation moves data from memory to processor register, and a *store* operation moves data from processor register to memory. The memory access is instrumented for collect details for each *memory referencing instruction* holds one stack blocks acessible in order to implement the memory model.

Let *expr* = *op*(*parent*) be the expression to extract, let *tp* be the type of *expr*, and let *addr* be its address. if the structure of *expr* is a pointer with address value *val*, add a vertex υ = (*val*, *tp*, *addr)* to *V* and edge (*parent*, υ, *op*) to *E*. If *expr* is an array containing *n* members *m*[0], *m*[1],....,*m*[n-1], add a vertex υ = ([...], *tp*, *addr*) to *V*, and an edge (*parent*, υ, *op*) to *E*. For each $i \in \{0,1,..,n\}$, invoke *extract*(*expr*, λx."x[i]",*G*). extracting the array elements.

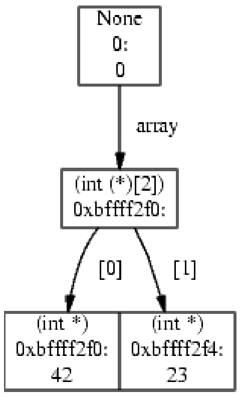


Figure 4. Show a memory graph for an array with two elements

4. Reverse Tainting Analysis

Reverse tainting analysis is necessary to trace back to the origin to determine all *objects*[2] that are related to the error. This technique is also used to identify how the *objects* can be used for dynamically instrument the trigger. This technique includes understand the origin of the series of unexpected events.

The set of *objects* resulting of *reverse tainting analysis* helps compare program states and identify what can be used for instrument the trigger. An inportant stage for reverse tainting analysis is generating error traces for determines IR instructions. The *code cache* contain IRStmt and IRExpr that are modified by running a set of bechmark programs written for each of the programming erros. The IR is used for determines the algorithm. This section shows some programming erros observed.

Unchecked array indexing. When an array indices are within the valid range the value can be influenced by data originating from untrusted sources. The instrumentation code can detect rule for violations to dereference the pointer value. The attack is launched inputting the pointer value enbedded in loop index variables as buffer indexes. The pointer

[2]Abstractions for construction detail of memory model and the algorithm.

to the array is an unchecked value outside of array index allowing the attacker to overwrite the memory address.

Integer overflow or wraparound. An integer overflow can be triggered through misinterpretations of signed, unsigned, long and short intergers. Invalid conversions of integers between types cause unexpected value bypass the bounds check resulting in an undefined behaviour.

Off-by-one error. One byte outside the range of the array can cause off-by-one error. The analysis trying caracterize a sets of intervals for states comparison after each memory access to visualize security asptects. Examining data structure to look for the presence or absence of the null character terminator.

5. Future improvements

Reproduce more experiments and refinements for other programming erros. Use formal language for description of model checking algorithm for each of the programming erros. Provides experimental results of the regression test in *shuntaint/tests*. Add suport for extract dynamic arrays for heap-based erros. Show false negative and positive scenarios generated during execution of experiments.

6. Conclusion

In order to combat the impact of programming erros, it is necessary to have automatic attack mechanism. This paper proposes a runtime attack mechanism that allow a successful memory corruption. The Shuntaint tool provides mechanisms to obtain a memory graph of the program to systematise the understanding of programming error. This force specific user data bypass the bounds checks present in the system with instrumentation necessary to trigger erros that leads to other erros.

References

[1] Common Weakness Enumeration. http://cwe.mitre.org/index.html.

[2] CWE/SANS TOP 25 Most Dangerous Programming Errors. http://www.sans.org/top25errors/.

[3] W. Drewry and T. Ormandy. Flayer: Exposing application internals. In Boston, editor, *First USENIX Workshop on Offensive Technologies* $WOOT'$07, Massachussetts USA, August 2007.

[4] T. Durden. Automated vulnerability auditing in machine code. In *Phrack Magazine*, volume 64. www.phrack.com, May 2007. Issue 8.

[5] Formal Methods Europe. www.fmeurope.org/.

[6] Gnu gdb. http://www.gnu.org/software/gdb/.

[7] Memory Graphs. http://www.st.cs.uni-saarland.de/memgraphs/.

[8] D. Molnar and D. Wagner. Catchconv: Symbolic execution and run-time t.ype inference for integer conversion erros. Technical report, UCB/EECS-2007-23 EECS Department University of California Berkeley, February 2007.

[9] N. Nethercote and J.Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In S. Diego, editor, *PLDI*, California USA, June 2007.

[10] Ptrcheck. http://valgrind.org/docs/manual/pc-manual.html.

[11] J. Seward and N. Nethercote. Using valgrind to detect undefined value erros with bit-precision. In Anaheim, editor, *USENIX'05 Annual Technical Conference*, California USA, April 2005.

[12] Silvio Cesare. Security Applications for Emulation. http://www.ruxcon.org.au/files/2008.