Testing micro-kernel syscalls to discover vulnerabilities

Amaury Gauthier*, Clément Mazin*, Julien Iguchi-Cartigny* and Jean-Louis Lanet*
*Smart Secure Devices Team — Xlim Laboratories
University of Limoges
France
{amaury.gauthier, clement.mazin, julien.cartigny, jean-louis.lanet}@xlim.fr

***Abstract*—Virtual machine monitor is a hot topic in the embedded community. Most of the current hypervisors for embedded devices use the paravirtualization technique which runs well on small processor without virtualization instructions. This is the case of the OKL4 kernel which is based on the L4 micro-kernel and implements among other the Linux kernel as guest OS.**

We introduce our method based on fuzzing and constraints solving for testing the security of OKL4. We have chosen to focus on the lower level OKL4 interface usable from an external actor: the system calls API. Because all operating system components use directly or indirectly these system calls, a minor flaw at this level can impact in chain the entire system including a virtualized kernel.

I. Introduction

Virtualization is an attractive technology which brings a lot of flexibility whatever the application domain is. But this flexibility has implication on the system security. Embedded virtual machine monitors are still young, and currently, the security impact of a vulnerability in this software piece is not known. This fact motivated our work on the security analysis of these components.

We begin this paper with a section which explains the main arguments in favor of embedded virtualization in a mobile phone context. Then, we present a new grammar which enables the formalization of the OKL4 system calls. Finally, we talk about the use of these models to test the system calls.

II. Virtualization in mobile phone

The first argument used by Companies which develop mobile virtual machine monitors is the cost reduction. Indeed, we can use a hypervisor to make current operating systems run seamlessly on new hardware components with only modifications to the hypervisor architecture dependent code. Depending on the hypervisor architecture, we only need to write a hypervisor driver if we change or add a new device on the system. All operating systems running on top of the hypervisor will be able to use directly the new device functionalities. A hypervisor also enables the possibility to run a real time operating system dedicated to radio communication and a "user experience" operating system on the same system on chip. Nowadays, the majority of mobile phones use two systems on chip when the user experience part of the phone is slightly advanced.

There is also the software update management problem which appeared with the success of smart phone. It can be resolved at least in part by delegating update management to the hypervisor.

Finally, Companies are more and more concerned by the assets management. A hypervisor can be used to build a trusted chain at low level and enforces a Company policies when employees use the same mobile phone for unrelated context (e.g. business and personal context). Hence, a Company can use hypervisor capabilities to protect their assets and enforce a Company security policies.

III. Testing the OKL4 micro-kernel

There is little work focused on the hypervisor security. But this domain is very close to the more developed operating system security field. Several approaches exist to test the kernel of an operating system like statical analysis, fuzzing or formal verification[1]. We choose to base our work on the fuzzing technique.

A. System calls formalization for OKL4

In OKL4, a system call is a function taking at least one arguments and providing one or more results. There are two types of input and output arguments: standard parameters and virtual registers.

Standard parameters are like normal arguments of any function in a simple program and virtual registers are objects which are associated to each thread in the system. The user can interact with them with "getter" and "setter" functions. They can be mapped directly on processor registers and must be set before running a system call.

Some conditions must also be verified before running a system call to ensure its success. The system should be in a precise state described in the OKL4 Micro-kernel Reference Manual[2].

After a system call, some results are returned. There is always a *Result* parameter which indicates if the system call is successful or not. If the *Result* is set to false, the *ErrorCode* parameter provides some information about the error. The different errors are described in [2]. In case of a successful system call, the manual describes in which state OKL4 should be after this call.

B. A grammar for system calls testing

In order to enhance test generation, we have designed a grammar to model OKL4 system calls. This grammar allows to model precisely a system call: its execution context, its parameters, their constraints and the results of the system call.

Then, the data stored in these models can be used to perform computation. These models have two goals. Firstly, they will be used as inputs for the space test generation algorithm.

Secondly, they will be used to automatically configure the fuzzer.

As an example, we have modeled the OKL4 *IPC* system call. This system call is used when two processes want to exchange data.

```
Name Ipc
Args
  Name 'TargetIN'
  Type 'ThreadID'
  VirtualRegister 'Parameter0'
  Constraint '=NilThread OR
    in(SysState.ExistingThread)'
  Name 'SourceIN'
  Type 'ThreadID'
  VirtualRegister 'Parameter1'
  Constraint '=WaitNotify OR =AnyThread OR
              =ThreadID OR =NilThread'
  Name 'AcceptorIN'
  Type 'Acceptor'
  VirtualRegister 'Acceptor'
  Constraint '=True OR =False'
  Name 'TagIN'
  Type 'MessageTag'
  VirtualRegister 'MessageDate0'
  Constraint ''
InputRegisters
  Name'Data1IN'
  Type 'Word'
  VirtualRegister 'MessageData1'
  Constraint ''
Output
  Name 'ReplyCapOUT'
  Type 'ThreadID'
  VirtualRegister 'Result0'
  Constraint 'TargetIN=NilThread => undefined'
  Name 'SenderSpaceOUT'
  Type 'SpaceID'
  VirtualRegister 'SenderSpace'
  Constraint 'TargetIN=NilThread => undefined'
  Name 'ErrorCodeOUT'
  Type 'ErrorCode'
  VirtualRegister 'ErrorCode'
  Constraint '=NoPartner OR = InvalidPartner OR
              =MessageOverflow OR =IpcRejected OR
              =IpcConcelled OR =IpcAborted,
              TargetIN=NilThread => undefined'
  Name 'TagOUT'
  Type 'MessageTag'
  VirtualRegister 'MessageData0'
  Constraint 'TargetIN=NilThread
   => TagOUT.Untyped=0'
  Name 'DataOUT'
  Type 'Word'
  VirtualRegister 'MessageData0'
  Constraint ''
Success
  'TagOUT.E=False AND ErrorCode=undefined,
   TargetIN=NilThread AND SourceIN=NilThread
    => null_syscall,
   SourceIN in(WaitNotify,AnyThread,ThreadId)
    => ReplyCapOutSysState.ExistingThread'
Failure
  'TargetIn.NotWaiting => ErrorCode=NoPartner,
   TargetIn notIn(SysState.ExistingThread)
    =>  ErroCode=InvalidPartnerr,
   TagIN.Untyped>MaxMessageData'
```

A constraint like *TargetIN=NilThread => undefined* means that if *TargetIN* value is *NilThread*, then the value of *ReplyCapOUT* will be undefined.

IV. How to use our system calls model to generate useful fuzzer outputs

From the above model, we can compute some properties. The set of properties allows the generation of each valid system calls. But this set can not be used as fuzzing input directly because the input domain is too large to be evaluated in a reasonable time.

But, we can use the *Constraint* property of the model in a manner to remove some testing values from this first set and therefore reduce it. Indeed, this property allows the elimination of all values which are not valid. We just need to keep some values which are out of these bounds to effectively test verifications operated by the kernel at system call invocation.

The main algorithm of our method is defined in the next pseudo-code fragment. It takes as parameter a system call formalized with the grammar and the system state. This last argument enables the computation of constraints which use the operating system state to restrict the set of values. The function returns a set of values to be tested on the system call in addition to the expected values.

```
computeValues(Grammar syscall, SystemState state)
              : output{
  foreach syscall.Args,
          syscall.InputRegisters as arg{
    space <- computeConstraint(arg.Constraint,
                               state)
    foreach space as value{
      output[arg] <- value
      output[arg] <- lowerBound(value)
      output[arg] <- upperBound(value)
    }
  }
  output[Success] <-
      computeConstraint(syscall.Success, state)
  output[Failure] <-
      computeConstraint(syscall.Failure, state)
  foreach syscall.Output as out {
    output[Success] <-
        computeConstraint(out.Constraint, state)
    output[Failure] <-
        computeConstraint(out.Constraint, state)
  }
}
```

V. Future works

This is an on going work, so it remains a lot of work to make this method fully functional and automatic.

When this goal will be reached, we will work on the portability of this method to make it usable on other micro-kernels and then we will try to apply it to more traditional virtual machine monitors like Xen[3].

References

[1] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: Formal verification of an OS kernel," *Communications of the ACM*, vol. 53, no. 6, pp. 107–115, Jun 2010.

[2] O. K. Labs, "Okl4 microkernel reference manual."

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*. New York, NY, USA: ACM, 2003, pp. 164–177.

A. Gauthier is a second year PhD student, in the Xlim laboratories at University of Limoges, working in the Smart Secure Devices team on the security of hypervisor for embedded devices. His research interests include operating system security and hypervisors for embedded systems.