COMP9242 - Paper 1

z

November 9, 2014

1 Summary

This paper addresses the issue of automated management of virtual machines from the host VM, without administrative access to the running guests. Some common use cases are administrative actions such as virus scans, updating a firewall rule, reconfiguring the system IP, etc, all of which we might want to manage from the host VM.

To do this, two aspects are required - firstly administrative tools that allow us to perform the intended action, and secondly a way to control the guest VM using those tools. This paper reuses existing administrative tools via alterations to dynamically linked libraries, resulting in intercepting system calls on the host made by those administrative libraries, and executing selected system calls on the guest OS instead.

To control the guest VM - namely, to be able to execute the selected system calls on the guest - firstly they detail a method of safely hijacking the guest VM to create a new process, namely via intercepting execution as soon as execution switches to the kernel (either system calls or interrupt traps), calling getpid, and forking once we find the init process to ensure a sufficiently privileged process. This process then constantly loops calling int3, an interrupt instruction commonly used via debuggers, to constantly trap into the kernel when scheduled.

To execute an actual system call on the guest, we then intercept one of the int3 traps, and execute the system call with that helper process for execution context such as open files on the host, using a memory mapped shared memory buffer to transfer the arguments and results between guest and host.

In choosing system calls to execute on the host and on the guest, some are annotated with their execution context, chosen primarily for their typical usage in a management context. For example, fork and exec will typically be executed on the host, while data related system calls - such as open and read - may be executed on either, and the choice is determined via other methods such as annotating file names and adding offsets to file descriptors.

2 Pros

The authors have successfully built a system that can be used to perform syscalls within the guest OS that are called from the host OS, and demonstrated this in practice to perform system management. In particular, the ability to reuse existing binaries is notable.

The paper has a reasonable overview of related work which gives an idea of several alternative approaches to the same and related issues, such as other methods to obtain and update information required to manage the guest, and alternate methods of process injection.

3 Cons

In transferring data between host and guest OS they map - and lock - the entire amount of memory required in both host and guest.

Their trusted computing base includes the guest OS kernel, the host OS and the hypervisor code, plus HYPERSHELL. This is a massive trusted computing base!

It may also be potentially exploitable - suppose that a malicious guest OS syscall was modified to return "/outside/shadow" as one of the files in the current working directory, and that output was later used as input to cp? Their method of distinguishing between syscalls that should be executed in the guest OS vs externally using the file prefix would cause that file to be opened and modified on the host, rather than guest. (They do note that it cannot defend against guest kernel attacks, however arguably this is a requirement of virtualization in most real world situations - that a compromised guest cannot compromise the host!)

4 Criticisms

The typical use cases given are tasks that are not frequently executed, likely hourly or daily at best. They note up to 5% overhead on operations in the host OS - a permanent overhead of this magnitude in operation is significant!

They note that their system should be extensible to any POSIX compatible OS, however they only ever test different Linux versions.

How does this system handle multiple guests? How does it distinguish which guest to run the syscall in? What performance impact would this have? It appears from reading the paper that this is not implemented or tested at all, yet it would be the common real world use case. Aspects such as using specific file descriptor ranges and filename annotations would introduce new challenges and limitations for supporting multiple guests.

No mention is made of any concurrency being possible - in using a single helper process within the guest to execute every syscall, and using that helper process's kernel context, it is likely that they only allow for a single utility process to access the guest. Otherwise, two competing utilities expecting different context - such as the process's current working directory - would conflict.

Similarly, most management utilities expect a clean process context - many programs do not 'clean up' their execution context, such as closing files and changing the working directory, on exit. If we use a daemonized helper process, any later management utilities executed would inherit this state, producing potentially unwanted results.

Security wise, the ability of the management utilities to run outside of the normal authentication methods, including encryption, and to do so without any logging, is a notable limitation to this in real usage. In particular, being able to bypass full disk encryption in this fashion makes the encryption itself rather futile - you can simply copy the entire contents of the disk in this manner to decrypt the contents, and all of the concerns regarding data breaches reoccur.

They note in particular that one limitation is for the host and guest to have compatible syscall interfaces to allow for reuse of the host utilities. It would have been interesting to see a discussion of

As noted in the benchmarking results, a non-daemonised helper process, namely one that is created each time it is required, is very expensive due to the need to intercept every syscall for up to 5s. However, where the overhead of having the daemonised helper running is an issue, and syscall tasks are executed infrequently, we essentially suffer this overhead on every syscall (or rewrite the management utilities to spawn a helper at the start of execution, and kill it at the end of execution, which is unwanted.) It would be good for the tradeoffs involved here - and types of workload that each option would be viable for - was discussed.

Benchmarking results

They use a 'representative subset' of utility software from selected packages - while their justification is reasonable, it is of course easy to omit any specific utility that might show a particularly poor result. These applications, at least, are reasonable real world usage for their use-case, so the results are potentially useful.

The timing results for all of the utilities are normalised for each utility, at which point they calculate the arithmetic mean (not geometric!) They also omit details such as standard deviation.

The 'average' times for base and hypervisor execution have very little intuitive meaning, particularly since the bulk of the times are less than 0.2ms - with 5 outliers out of the hundred that dominate those times. If the top utility was removed, the average 'time' they quote would go down to 2 - 3ms!

For LMBench, they again focus on specific tests. Notably they omit the null syscall - which would have been a useful result.

They note that the overhead is mainly 'from the data exchange and synchronization' with no justification or discussion. They note that there is particularly poor performance in the ctxsw micro benchmark, but only note that this is due to it testing context switch timing, with no discussion of how this might be mitigated, or affect performance in general.

Percentage slowdown appears to mean percentage of the new execution time that is overhead - I'm uncertain if this interpretation is common in other papers, but it seems unusual.

Antivirus case study: a very useful additional timing result would have been executing the virus scan without disk encryption, as a comparison to the scans with encryption.