EPiC Series in Computing

Volume 89, 2022, Pages 72–80

Proceedings of 35th International Conference on Computer Applications in Industry and Engineering

Building a Distributed System for Live Virtual Machine Introspection

Steven Valle,
Nagarajan Prabakar
School of Computing and Information Sciences
Florida International University
Miami, USA
{svall052, prabakar}@fiu.edu

Himanshu Upadhyay
Applied Research Center
Florida International University
Miami, USA
upadhyay@fiu.edu

Abstract

The introspection of virtual machines is an important aspect of protecting against the threat of malware that can hide from traditional automated malware-detection systems. A distributed system for live virtual machine introspection is presented utilizing the Xen Project hypervisor and LibVMI for introspection. The system incorporates the importing of VMs through the OVF specification, VM management through libvirt, and the streaming of various kernel data structures and system calls into data stores with no delay between introspection and storage.

1 Introduction

The Cyber Threat Automation and Monitoring System (CTAM) is a project at the Applied Research Center funded by the Test Resource Management Center (TRMC) that provides a testbed environment for the live introspection of virtual machines for use in anomaly detection. As a project mainly implemented by undergraduate and graduate students (with several student transitions over the years), it was important for the system being built to have the ability to evolve over time and be modular to the point where new students to the team could have contributions. In this paper, we will be presenting the design and implementation of the backend system used to support the CTAM project.

The main goal behind our introspection system is to package the full stack of tools needed to perform live virtual machine introspection. The following specifications were crucial in building it:

- Agentless introspection support for x86 Windows and Linux virtual machines
- Performance of VMs must be minimally affected under introspection
- Virtual machine management support
- Scalability for multiple hypervisors and virtual machines
- API support for web front-end

In the following sections, we will outline the decisions taken for building the infrastructure, programming languages, and libraries selected to enable management and streaming pipelines, and problems we encountered in scaling the system and our solutions to these problems.

2 Designing the system

The main objective of the CTAM project involved implementing machine learning and deep learning models to detect anomalous behavior in processes operating inside virtual machines. Therefore, the first step in building a testbed for the extraction of this data is picking an introspection library and hypervisor. The second step was deciding on how to store the extracted data and operate on it. The third step involved designing API endpoints for executing introspection jobs and managing hypervisors and VMs in the system.

2.1 Introspection library and Hypervisor

The decision of which introspection library and hypervisors to focus on was a decision made in tandem. As open-source projects, LibVMI and the Xen Project were the library and hypervisor chosen. LibVMI [12] is a project born out of the XenAccess Project [1] whose focus is on reading and writing memory from virtual machines, accessing CPU registers, pausing and unpausing VMs, and trapping on hardware events (cite), giving us the tools necessary to build introspection applications. With its integration into introspection tools like Volatility [7] and Rekall [8], we felt that it was the best choice to get started with.

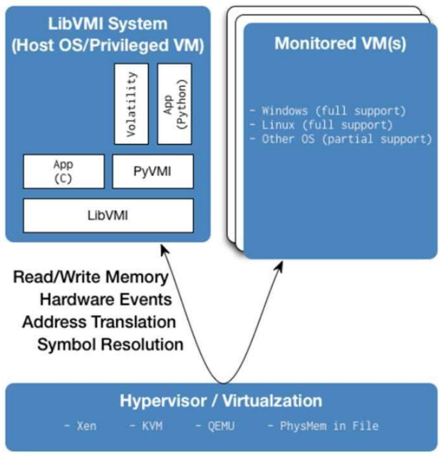


Figure 1 [5]: LibVMI and its supported platforms, supported OS, and application structure.

As shown in Figure 1, LibVMI must reside on either the Host OS or a privileged VM of a Hypervisor. Our Hypervisor being Xen, the logical choice to place our introspection application is inside of its privileged VM, Domain 0. The structure of the Xen Hypervisor, as shown in Figure 2, dictates that only this privileged VM has direct access to hardware resources and oversees communication with the unprivileged VMs.

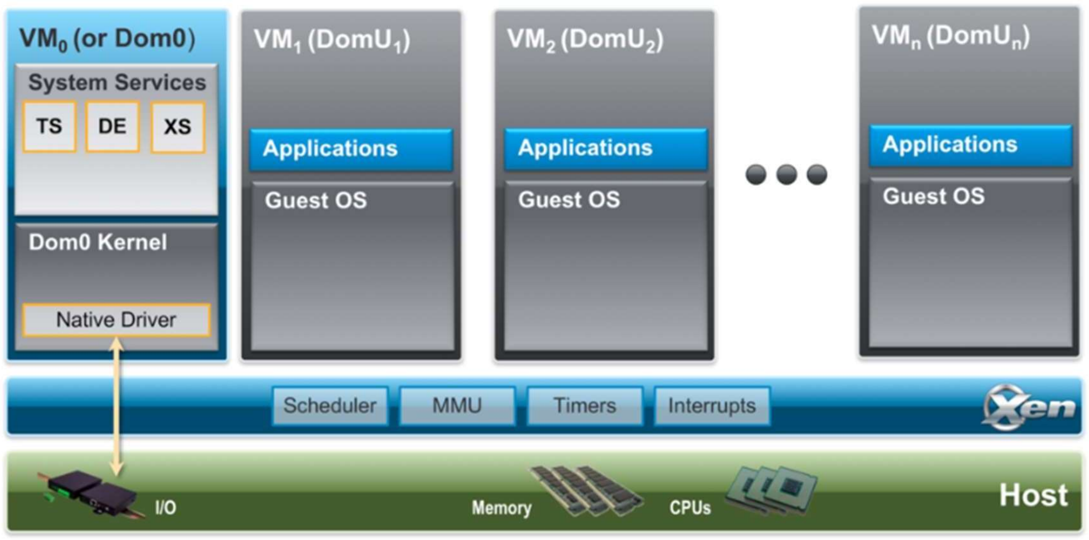


Figure 2 [2]: Xen architecture**.**

2.2 Data Storage and Analytics

The data we were set to extract from virtual machines was inherently very structured. As an example, for process monitoring on Windows and Linux, we were to extract the Executive Process Block and task_struct data structures for active processes on each respectively. Due to the nature of the data, a relational database system seemed to be the best choice at the time.

Coincidentally, during the design phase of the project, MSSQL 2017 was released with a new service named Machine Learning Services that integrated a Python/R environment into the MSSQL databases. With machine learning being a big part of the project, MSSQL 2017 seemed to be the perfect choice for not only storing data but also performing analytics. The usage of stored procedures in MSSQL made the workflow of loading data, storing, building models from the data, and serving the models for prediction as effortless as possible.

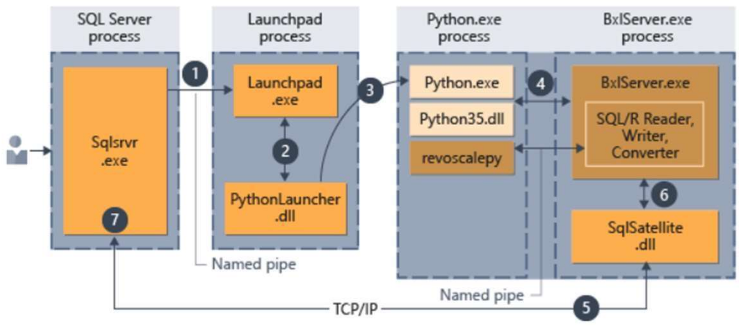


Figure 3 [3]: MSSQL Machine Learning Services workflow for executing Python scripts in-database**.**

The uses for the MSSQL database then included the following:

a) Tables describing Hypervisor and VM metadata for management

b) Tables describing extracted data structures for Windows and Linux
c) Python scripts for building machine learning models and storing them
d) Python scripts for serving machine learning models and storing predictions

2.3 Overview

There is one application for management and launching introspection jobs that is collocated with the hypervisor, exposing a SOAP endpoint. One application for introspecting VMs that is launched by the introspection manager, and python scripts coupled with the MSSQL database for building machine-learning models and ingesting data. This essentially creates two environments that can be scaled independently.

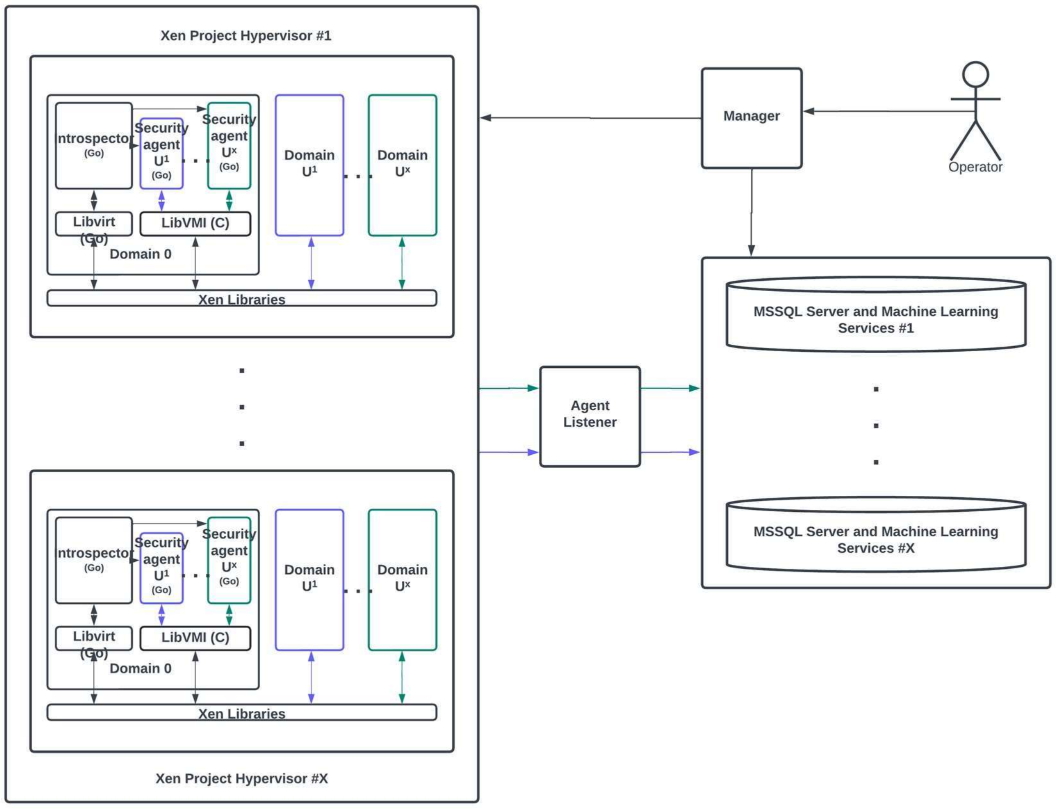


Figure 4: CTAM introspection.

3 Introspection and Management Server

The Introspector application is a server application exposing a SOAP API that enables the management of virtual machines and the execution and coordination of introspection jobs on those virtual machines. The application is written in Go.

3.1 Virtual Machine Management

The ability to manage virtual machines in the environment is an essential feature that helped the system to become operational. With this feature, we added the ability to perform the following operations through the SOAP API:

a) Resume VM
b) Start VM
c) Pause VM
d) Force Stop VM
e) Restart VM
f) Create VM
g) VCPU Configuration
h) RAM Configuration
i) Import VM
j) View VM

Operations a - h were fully implemented through the libvirt [6] Go API. The *Create VM* operation worked through a templating system where previously allocated VMs were abstracted into having three sets of QCOW2 images. These images consisted of Master images, Template images, and Operational images. The master image is the original VM image unused, template images are copies of the master image, and operational images are template images renamed and designated to a new VM. Template images were configured to be self-replicating up to a specified amount so that the creation of a VM does not have to wait until a master image is finished being copied. The three types of VM images were stored on the hypervisor file system.

The import operation brought additional file types besides the QCOW2 image of the VM. To give the ability for users to upload their exports and have the Introspector access them, we used a Samba share as an intermediary. Users could upload their files to the Samba share and using the SOAP API specify to the Introspector what to import. This essentially built a repository of OVA and OVF packages that the Introspector had access to.

The View VM operation was implemented by utilizing the *noVNC* [10] server application. This VNC server provides the capability to view and use the VM through a browser with the addition of a Web Socket proxy that was implemented in the Introspector.

3.2 Importing Virtual Machines

When it came time to expand our capabilities and include support for introspecting arbitrary VMs, the import feature was implemented. To target the widest range of virtual machine managers and their exporting mechanisms, we decided to implement our import functionality by adhering to the Open Virtualization Format (OVF) [9].

The OVF specification lays out an industry standard packaging format for the distribution of virtual systems onto and between virtualization platforms. The structure of an OVF package includes the following files:

- One OVF descriptor with extension .ovf
- Zero or one OVF manifest with extension .mf
- Zero or one OVF certificate with extension .cert
- Zero or more disk image files
- Zero or more additional resource files, such as ISO images

An OVF package can be stored as a set of files or as a compressed file with the .ova extension. For our purposes, we focused on parsing OVF packages by analyzing the OVF descriptor in its entirety but only implementing the necessary fields needed to get a VM functional. These fields are:

- VirtualSystem for descriptive information of VMs
- VirtualSystemCollection for allowing the import of multiple VMs form a single package
- ResourceAllocationSection for identifying the resources needed to boot the VMs
- References for linking file names to what they represent in virtual hardware
- DiskSection for identifying disks attached to VMs
- BootDeviceSection for identifying the virtual device to boot from

Once the OVF descriptor has been parsed for creating the VMs, the next step is to convert the file references that link to virtual disks into QCOW2 disk images for use in Xen. The *qemu-img* utility was used for this purpose and supports conversions with the QCOW2, QED, RAW, VDI, VHD, and VMDK formats. After conversion, an XML following the libvirt specification is generated for importing the VM into Xen.

3.3 Preparing Virtual Machines for Introspection

One of the requirements needed for efficient and accurate introspection is the utilization of symbol files. Symbols files contain debugging information and are subsequently used by debugging applications. Kernels also have symbol files for debugging them. Through the parsing of these symbol files, profiles for specific kernels are created that inform introspection applications where data structures are located in memory.

For virtual machines using Linux kernels, we kept a repository of Rekall and Volatility profiles that described it. For virtual machines using a Windows operating system, we built the profiles using the *vmi_win_guid* tool from LibVMI, the Windows symbol server, Rekall's *parse_pdb* utility, and Volatility's *pdbconv* utility. After generating these profiles, virtual machines and these profiles would be linked together in the MSSQL database for use later in introspection.

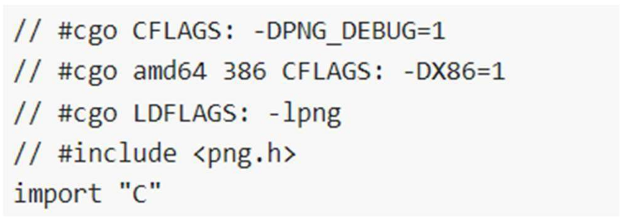
4 Data extraction

The LibVMI library gives unrestricted access to virtual machines and their memory through its usage of Xen management APIs. This library is accessible directly through C applications and indirectly through a Python interface. We chose to stick with Go for developing all applications in the introspection system and as such, the security agent process (introspection application) had to gain access to the C library.

Besides our own introspection implementation using LibVMI, we also utilized the Drakvuf utility to monitor system calls in VMs. In the following subsections, we will explain how we interfaced Go with C using CGO, how we integrated Drakvuf into our solution, and how we developed a stream processing pipeline in Go.

4.1 Interfacing with LibVMI through CGO

CGO is a pseudo-package in Go that allows for calling C functions and variables from Go and vice-versa. The way to interface C with Go starts at the top of the file where the "C" package is imported. Any comment before this import, known as the preamble, defines the environment that C code will run in.



```
// #cgo CFLAGS: -DPNG_DEBUG=1
// #cgo amd64 386 CFLAGS: -DX86=1
// #cgo LDFLAGS: -lpng
// #include <png.h>
import "C"
```

Figure 5: The #cgo directive in lines 1-3 is used to tweak the behavior of the compiler, line 4 imports a header file in C, line 5 is the "C" import in Go.

When structuring the way in which we interact with the C portion of our code, aptly named ARC-VMI, it was essentially treated as the library to a normal C application with no dependencies on Go code. When it came to extracting data, a C function is called from Go that returns a C structure containing the information for different data structures, usually containing some integer values, character arrays, and dynamically allocated arrays of other C structures. For every C structure there will be an equivalent Go structure to wrap the data around once it is generated.

4.2 Drakvuf

The Drakvuf [11, 4] utility is an open-source project that provides a virtualization-based agentless black-box binary analysis system. Among its many plug-ins, we utilized its system call monitoring plugin for feeding into a machine-learning model. A model that learns the behavior based on the usage of system calls is created for each process of concern. Drakvuf is a shell application and as such, its output is printed to the standard output, in the normal case being the shell. For our purposes, we attach a pipe from Go into the Drakvuf applications standard output and consume the data in a stream.

4.3 Stream processing with Go routines and Channels

With our LibVMI implementation for extracting data (_EPROCESS and _ETHREAD blocks, VAD trees, Interrupt Descriptor Table, System Call Table) and the consuming of Drakvuf extracted system call data, we have two separate pipelines for introspection. Both pipelines are very different from each other and utilize a LibVMI instance tied to the VM so they cannot be run at the same time on the same VM. We will now go through the details of both workflows and how we implemented stream processing and insertion in Go.

ARC-VMI workflow:

- Create a SQL connection pool.
- For each VM being introspected in the hypervisor:
    a. Initialize LibVMI instance.
    b. Parse through Rekall/Volatility offsets and store into C structure.
    c. Create a spout channel for each C data structure containing VM data.
    d. Run a goroutine to insert VM data into MSSQL DB.
    e. Start introspection.

Drakvuf workflow:

- Create a SQL connection pool.
- For each VM being introspected in the hypervisor
    a. Declare Drakvuf command execution.
    b. Attach a pipe to Drakvuf standard output.

c. Feed pipe into an io.Reader interface for reading from a stream.
d. Create a "spout" channel for Drakvuf strings.
e. Create a "sequence" channel where all system calls get inserted into.
f. Create generic "system call type" channels where specific system call types can be filtered for.
g. Run a "stream splitter" goroutine that filters output from the "spout" channel based on the system call type and inputs into the appropriate "system call type" channel.
h. Run a goroutine to insert VM data into MSSQL DB.
i. Start Drakvuf command.

Structuring the execution of Go code in this way essentially allows us to do two things. Firstly, to set up a pipeline for how data will be ingested, pre-processed, and outputted before any introspection starts. Once introspection does start, all operations will be executed concurrently as data is extracted. Secondly, once introspection is done, all the data will be residing in the database with no delay.

4.4 Conclusion

As we set out to research the viability of utilizing machine learning and deep learning techniques for monitoring the runtime state of virtual machines, an all-inclusive system for introspection was built. The problems of preparing virtual machines for introspection, interacting with virtual machines under introspection, and the building of efficient data pipelines were tackled. Alleviating these problems allowed the data science and penetration testing teams to iterate over new data and virtual machines seamlessly.

References

[1] B. Payne, M. Carbone and W. Lee, "Secure and Flexible Monitoring of Virtual Machines," in *Twenty-Third Annual Computer Security Applications Conference*, Miami Beach, FL, USA, 2007.

[2] "Xen project software overview," [Online]. Available: https://wiki.xenproject.org/wiki/Xen_Project_Software_Overview.

[3] "What is SQL Server Machine Learning Services (Python and R)? | Microsoft Docs," Microsoft, [Online]. Available: https://docs.microsoft.com/en-us/sql/machine-learning/sql-server-machine-learning-services?view=sql-server-2017.

[4] "DRAKVUF® Black-box Binary Analysis System," [Online]. Available: https://drakvuf.com/.

[5] D. PAne, S. Maresca, L. K. Tamas and A. Saba, "LibVMI," [Online]. Available: https://libvmi.com/.

[6] "Libvirt," [Online]. Available: https://libvirt.org/.

[7] "Release v1.0.0 · volatilityfoundation/volatility3," Github, [Online]. Available: https://github.com/volatilityfoundation/volatility3/releases/tag/v1.0.0.

[8] "Rekall Forensics," [Online]. Available: http://www.rekall-forensic.com/releases/release171.

[9] "DMTF," 27 August 2015. [Online]. Available: https://www.dmtf.org/sites/default/files/standards/documents/DSP0243_2.1.1.pdf.

[10] "noVNC," [Online]. Available: https://novnc.com/info.html.

[11] S. Lengyel Tamasl Maresca, B. Payne, G. Webster, S. Vogl and A. Kiayias, "Scalability, fidelity, and stealth in the DRAKVUF dynamic malware analysis system," in *Proceedings of the 30th Annual Computer Security Applications Conference, 2014-12-08,p.386-395*, 2014.

[12] H. Xiong, Z. Liu, W. Xu and S. Jiao, "Libvmi: A Library for Bridging the Semantic Gap between Guest OS and VMM," in *2012 IEEE 12th International Conference on Computer and Information Technology, 2012-10, p.549-556*, 2012.