VmiCVS: Cloud Vulnerability Scanner

Anil Kumar Konasale Krishna[*]
University of Utah
Salt Lake City, USA
akumarkk@cs.utah.edu

Robert Ricci[†]
University of Utah
Salt Lake City, USA
ricci@cs.utah.edu

ABSTRACT

Every service that runs in cloud systems comes with its own set of vulnerabilities. It is important to detect and assess those vulnerabilities to provide seamless and secure service to the users. Various scanners such as Port scanner, Network scanner, Web application security scanner, Database security scanner, Host based vulnerability scanner etc provide security assessment. But these scanners use methods that an attacker uses to attack in order to expose the vulnerabilities. As a result, application ecosystem might get disturbed and hard-to-attack vulnerabilities might left undetected. A yet another set of scanners check version of the service through protocol level messages in order to determine the vulnerabilities applicable to that particular service version. With this approach, certain vulnerabilities are not discovered when a particular software piece(example : glibc) is not directly exposed to the remote user.

We propose a novel Cloud Vulnerability Scanner, VmiCVS (Virtual Machine Introspection based Cloud Vulnerability Scanner). It provides security assessment of vulnerabilities even if the software is hidden from remote user and without disturbing application ecosystem. It can be used by cloud provider to provide Vulnerability scanning-as-a-service where detected vulnerabilities are reported to tenant for additional incentives. We have evaluated our scanner by assessing the vulnerabilities of services such as *sshd* and hidden(from remote user) libraries such as *glibc* and *libcrypto*.

Keywords

Virtual Machine Introspection; Vulnerability Scanner; Version Scanner

1. INTRODUCTION

Every service that runs in cloud systems comes with its own set of vulnerabilities. It is important to detect and assess those vulnerabilities to provide seamless and secure service to the users. The recent Distributed Denial of Service (DDoS) attack on American Banks, Bash's remote code execution vulnerability(shell shock) [8], buffer overflow bug exploited by [5], OpenSSL heart bleed bug, OpenSSL Null pointer assignment flaw [7] exploited to cause Denial of Service(DoS) attack etc shows the importance and imminence of Cloud vulnerability Assessment tools. Intrusion Detection and Prevention Systems(IDS/IPS), Firewalls and Anti-Virus are used as security solutions. Attacker can still bypass these security solutions and exploit known vulnerabilities. As reported in 2015 Data Breach Investigations Report [1], 99.9% of the exploited vulnerabilities had been compromised more than a year after the associated CVE was published. As reported in Vulnerability Assessment report [12], one of the 70 largest security breaches were achieved through the exploitation of known vulnerability in spite of the presence of correctly installed Firewalls, IDS/IPS and Anti-virus. These reports indicates the need for vulnerability assessment systems even in presence of other security solutions.

Existing Vulnerability Assessment (VA) Scanners do not produce accurate results. VA scanners produce many *false negatives.*As stated in [10], nearly all VA scanners rely on version checking as their primary method of known vulnerability assessment. Version number is extracted from the response header. But this method of extracting version number is not reliable as a) Header does not contain enough information such as patch number b) Application itself is configured to not reveal the version information in the header to make the attack harder c) version of the shared libraries is not revealed d) Firewall could manipulate the header information or header can be hidden and suppressed. *System library version information is not included in the protocol header fields.* So version analysis does not list the vulnerabilities in the system libraries. Though VA scanners perform Behavior Analysis, as the potential known vulnerability list is huge, it is not possible to assess all of them by exploitation tests. Further, evaluation of seven VA scanners performed in [15] shows that automated vulnerability scanning is not able to accurately identify all the vulnerabilities present in the system. Though Authenticated VA scanning produces better results, it requires system credentials.

Though there exist tens of Vulnerability Assessment Scanners in the Market, they still have to evolve. Most of the existing Vulnerability Assessment Scanners are not *safe*. As reported in [2] and [11], scanners can crash the system under testing, disrupt normal network operations etc. In some sense, this is not surprising as scanners use crafted network

[*]
[†]

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

packets to exploit the vulnerability like an attacker. For example, in order to assess buffer overflow bug, scanners send enough data to overflow the buffer and the overflow could cause any unpredictable program execution including system crash.

We propose a novel Virtual Machine Introspection(VMI) based Cloud Vulnerability Assessment Scanner, *VmiCVS*. It is *safe* as it does not use crafted network protocol packets to exploit vulnerabilities in order to assess them. Instead, it produces accurate results by performing version analysis of all pieces of software including system libraries using VMI techniques. It does not need system credentials to produce accurate results.

Most of the software libraries and services have their version number stored in a symbol in their binaries. These constant symbols are loaded into *data segment* of the process memory layout. *stackdb* is a VMI library, which provides APIs to read value of any particular symbol of a running process or a library linked to running instance of a binary. It also provides the most powerful and simpler APIs to read the virtual memory pages of a process. VmiCVS reads the virtual memory pages of a process through *stackdb* APIs and runs a *version parser* program to extract the version string. Through this method, even the version of libraries such as *glibc*, *libcrypto* etc can be determined.

We have evaluated VmiCVS by scanning the version of three different pieces of software; *sshd*,running instance of a process binary, *libcrypto*, library that contains version number as a part of its name, *glibc*, library that do not contain version number in its name.

The rest of the paper is as organized as follows: In Section 1 we discuss Background. In Section 3 we present our VmiCVS architecture, while the VmiCVS implementation is detailed in the Section 5. Evaluation of VmiCVS is explained in the Section 6 and finally, conclusion and further research directions are given in the Section 7.

2. BACKGROUND

Cloud computing services such as Infrastructure as a Service(IaaS), Software as a Service(SaaS) and Platform as a Service(PaaS) are delivered through virtualized computing resources. Virtualization is the main enabling technology for cloud computing. Hardware Virtualization abstracts underlying physical hardware and allows multiple instances of operating systems to be running on the same physical machine with the help of Hypervisor. Hypervisor/Virtual Machine Monitor(VMM) is a software which provides virtual operating platform for guest operating systems and manages the execution of the guest operating systems.

It is very important to monitor the host in order to detect and report any malicious activity that compromises the host.**Virtual Machine Introspection (VMI)** is used to achieve this. VMI is a technique of introspecting Virtual Machine from outside of it for the purpose of analyzing the software running inside it. VMI is realized by interposing at hardware level which allows to mediate the interaction between hardware and the host software.

Stackdb is a debugging library with VMI support, which allows one to control and monitor the whole system at multiple levels. It can be used to monitor guest operating system itself, process running in guest or language runtime. It directly interacts with the system being debugged through Hypervisor interface. Stackdb provides APIs to install break points, watch the values of symbols in the processes running inside guest, read values of symbols of a process etc.

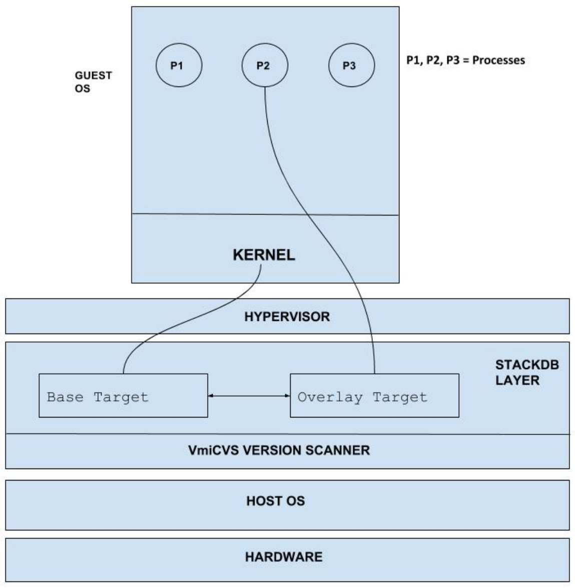


Figure 1: Cloud Vulnerability Scanner Architecture

Vulnerability Assessment Vulnerability can be defined as a flaw or bug in the system that present the opportunity for malicious exploitation and results in security breach or a violation of system security policy. Vulnrability Assessment [13] is a process of identifying, quantifying and prioritizing the vulnerabilities in the system. There exists tens of Vulnerability Assessment tools in the market; Nessus, OpenVAS, SAINT, Nikto, Wikto, nmap, Qualys, Fireeye, Alienvault, skybox security etc.

Behavior Analysis Behavior of a host in response to a specially crafted queries/packets is used to assess a paricular vulnerability. These special queries/packets are constructed to exploit a particular vulnerability.

Version Analysis In this method, version number of a service is detected in order to determine the exploits the server is vulnerable to. Usually existing vulnerability scanners predict the version number from the protocol headers and response messages to a special query packets.

3. ARCHITECTURE

VmiCVS architecture is as shown in the Figure 1

It consists of Machine hardware, Host OS, VmiCVS version scanner, Stackdb layer, Hypervisor and Guest OS. As shown in the Figure1, Host Operating System(OS) runs directly on hardware. VmiCVS version scanner program runs in Host OS and uses stackdb provided API layer to interact with kernel and user processes running in Guest OS. As shown in the Figure 2, it consists of ***Version Scanner program*** and ***Version Parser program***.

- *version scanner* dumps the data source containing version information into a file. Source of version number can be virtual memory pages of a process or shared library filename.
- *version parser* program matches the pattern specific to a software in order to extract the version number.

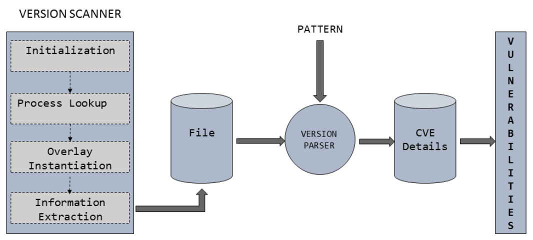


Figure 2: VmiCVS Version Scanner

- version number and software name is looked up in CVE database in order to list all the exploits a software is vulnerable to.

version scanner and *version parser* are explained in detail in the below section.

- *Version Scanner Program* It extracts the version information of desired library/binary running in the guest OS using stackdb APIs and dump it into a File. Version number of a library can be extracted from either the name of the library or through the virtual memory pages of the library. Based on how the version information is extracted, there are two types of version scanners

  1. *Virtual Memory based Version Scanner* It dumps the virtual memory page contents of a process into a file in order to extract the version number. Usually version number is stored in a symbol/macro in binary/library executable. For example, *glic* version number is stored in a symbol, `banner`. So virtual memory pages pointing to the *text segment* of a library/binary is the source of version number information in this case. It walks through the task list until it finds the desired process and dump all the virtual memory page contents into a file through stackdb's *overlay target*.
  2. *Name based Version Scanner* It dumps all the shared library names linked to a particular process into a file. Some shared library names contain version number. So shared library name is the source of information in this case. To list all the shared libraries of a process, it attaches to the guest operating system through base driver and walks through the task list until it finds the one which carries the version number of desired software piece and dump all its shared library names.

  Both the Version Scanners include three step process to dump the version information :

  - *Initialization* Initializes stackdb and VMI environment.
  - *Process Lookup* In this step, start of the task list of guest operating system is retrieved and lookup for the required process is performed by traversing the task list.
  - *Overlay Instantiation* stackdb overlay target handler corresponding to the desired process is retrieved in this step. Overlay target is stacked on top of the base target and is used to get state information of a process.
  - *Information Extraction* Information such as virtual memory contents or shared library names is dumped into the file on a disk.

- *Version Parser Program* It parses the output file produced by the version scanner program, based on the version pattern and report the version number. For example, pattern to extract the version of *glibc* is *Ubuntu EGLIBC %d.%d-0ubuntu%d.%d*

4. DESIGN

VmiCVS consists of two types of version scanners

- **Name based Version Scanner** As described in the paper [14], virtually all the systems incorporate some form of version number in the filename of each library in order to indicate and manage the upward compatibility. GNU libraries follow

  lib<name>.so.<major>.<minor>.<rls>

  format for the shared library names.For example libraries generated by Cyrus SASL package [4] such as *liblogin.so*.2.0.25, *libplain.so*.2.0.25, *libdigestmd5.so* - .2.0.25, *libsasldb.so*.2.0.25, *libsasl.so*.2.0.25, Linux PAM package [6] such as *libpam_misc.so*.0.82.1, *libpam.so*.0. 84.1, *libpamc.so*.0.82.1 and CrackLib package [3] libraries such as *libcrack.so*.2.9.0. So version number of such libraries can be extracted by finding the file name of the library. Tools such as *libtool* are available to automatically build libraries with version number in their file names.

  Shared libraries are linked against process binaries. In a running process, each such shared library file is loaded as a memory mapped file and its information is stored in kernel memory map data structure. As shown in the Figures3, each process is represented by a kernel data structure, `task_struct`. Information about memory mapped files is stored in `struct mm_struct *mm` member of `task_struct`. Each shared library linked against a process consists of text segment, data segment and bss. Each segment is mapped to a particular piece of process address space. This mapping information including shared library file name and other attributes are stored in `vm_area_struct` corresponding to data, text and bss segments of a library. `vm_area_struct` structure associated with each segment is linked with `vm_area_struct` of another segment through doubly linked list. The head of the link list is stored in `mmap` member of `struct mm_struct` shown in the Figure 3. The file name of the shared library can be obtained by scanning all the linked structures as shown in the Figure 3.

- **Virtual Memory based Version Scanner** process binaries and some libraries such as *glibc* do not contain version number as a part of their file names. For such

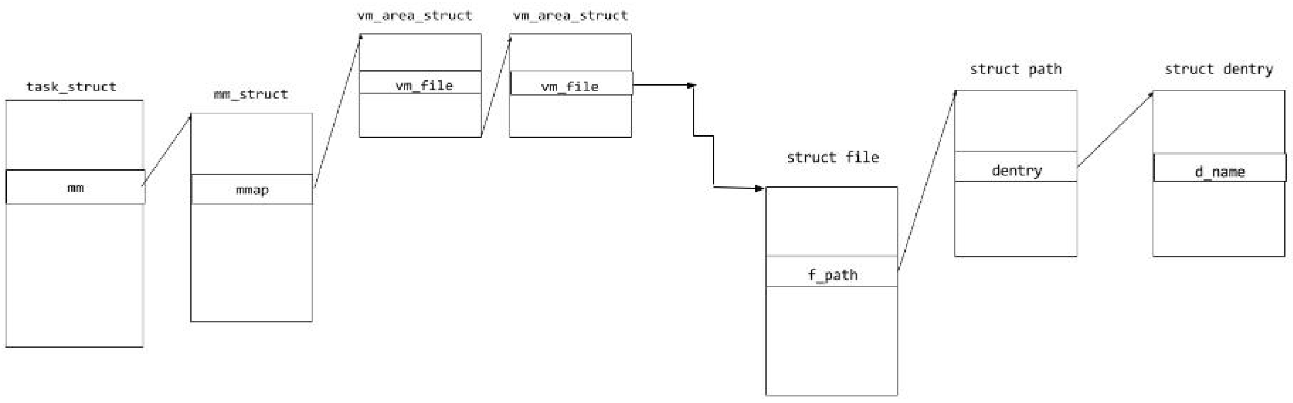


Figure 3: Kernel data structures traversed to read the shared library file name

libraries/binaries, this method is applied to obtain the version number.

Most of the shared libraries and binaries have their version number stored in a symbol as a constant literal or stored as a part of banner string. For example, version number of *glibc* is stored as a part of banner string in a variable `banner`, release version number of *sshd* binary is coded in the macro `SSH_VERSION`. These pre-defined version strings are stored in *text segment* of the library and are loaded into data segment based on the platform and operating system. As these segments are mapped to virtual memory pages of the process, contents of the memory pages are dumped into a file by version scanner program using stackdb APIs. Later *Version Parser* is used to extract the potential version string.

5. IMPLEMENTATION

Version string of a process binary or shared library is obtained by reading the virtual memory pages or by listing the shared library names. VmiCVS version scanner programs need to initialize the stackdb environment through stackdb APIs before probing or monitoring processes in guest operating system. VmiCVS program also provides command line options to specify the process or library name, whose version number needs to be scanned.

- *–vscanner-log-file* specifies log file name where to dump virtual memory contents of a process.
- *–vscanner-olay-process* option to specify overlay process, of which version user interested in.
- *–vscanner-olay-library* option to specify library name whose version user is looking for.

Implementation of two types of Version Scanners is explained below.

1. *Name based version Scanner* It consists of three steps
   - *Initialization* includes initializing stackdb and VMI environment and attaching to the target Guest OS.
   - *Process Lookup* step retrieves head of guest OS's task list and traverses the list to find the `task_struct` corresponding to the desired process.
   - *Information Extraction* As explained in the design section, shared library files are loaded into process address space as memory mapped files. Hence memory map data structures are read to obtain the file name of a particular shared library.

     stackdb provides APIs to read the values of any structure member given structure and member name. For example, to read the value of `comm`, `task_struct` structure and *comm* as a query are passed to stackdb APIs. Initially `comm` member of `task_struct` is read to check the process name. Next, as shown in the Figure3, values of `mm`, `mmap`, `vm_file`, `f_path`, `dentry` and `dname` symbols are read to list the name of memory mapped shared libraries.
2. *Virtual Memory based version Scanner*
   - *Initialization* includes initializing stackdb and VMI environment and attaching to the target Guest OS.
   - *Process Lookup* step retrieves head of guest OS's task list and traverses the list to find the `task_struct` corresponding to the desired process.
   - *Overlay handle retrieval* stackdb handle holding the meta data of the desired process needs to be retrieved from the base handle pointing to the guest OS using *stackdb* API `os_linux_list_for_each_struct()`. This API invokes a callback function supplied as argument to it on each structure.
   - *Information Extraction* Contents of the virtual memory pages of a process are read to extract the version number of a library/binary. As shown in the Figure 4, stackdb process memory model consists of an address space of a process which is divided into *regions* and *regions* are divided into *ranges*. *range* corresponds to a contiguous region of memory with same permission flags. *region* is a

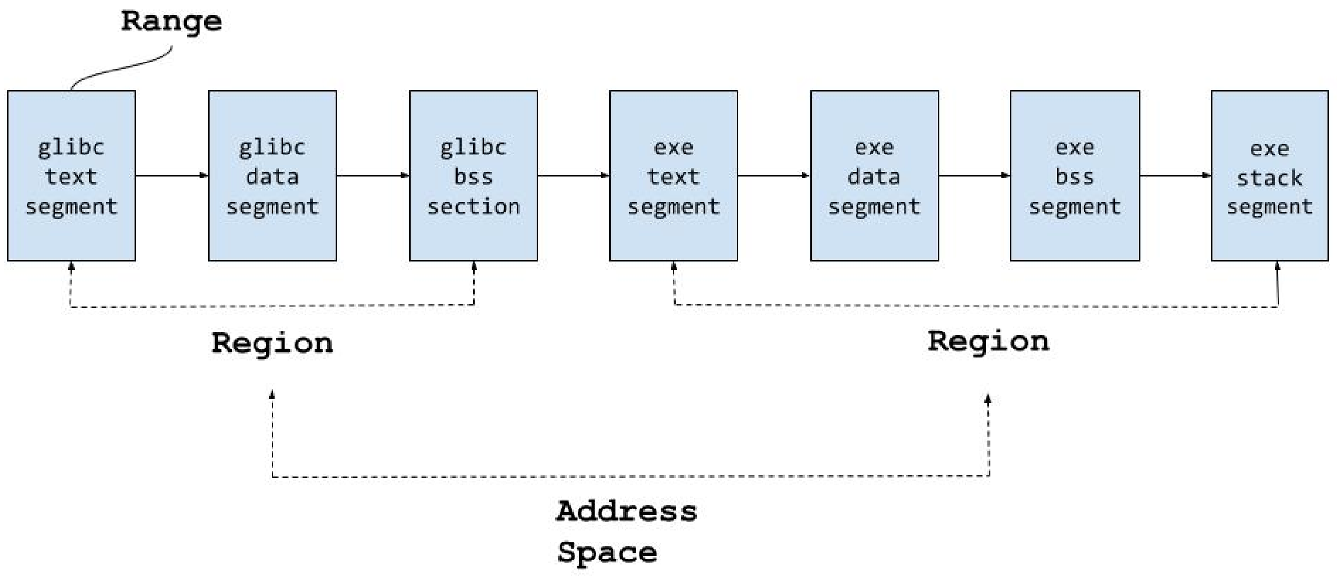


Figure 4: Stackdb Memory Model of a Process

group of related *ranges*. For example, a memory mapped library file(example: *glibc*) corresponds to a *region* and each segment(text, data and bss) of a mapped file corresponds to a range. An entire address space content of a process is dumped by walking through regions and each of the ranges in the regions.

Following are the APIs used in *initialization*, *Process Lookup* and *Information Extraction*

- `target_open()` *stackdb* API to open the target given target spec structure.
- `vmi_version_scanner()`

  input : target structure, version scanner arguments

  output : returns success on successful scanning of virtual memory pages of requested process, else returns failure code

  description : dump virtual memory content of requested process in specified log file. Log file and process are specified in argument structure

  – `target_lookup_sym()` call it to get the `struct bsymbol` corresponding to `init_task`, which is the head of doubly linked list of `task_struct`
  – `os_linux_list_for_each_struct()` used to traverse through the linked list of `task_struct`. `gather_version_scanner_info()` callback function is passed as an argument, which will be called on each `task_struct`
  – `gather_version_scanner_info()` handler function called for each `task_struct`. It invokes `version_scanner()` to dump the virtual memory contents of the interested process.
  – `version_scanner(target, vscanner_args)` Entire address space of a process is divided into number of regions. Each region is divided into number of ranges. `version_scanner()` walks through each of the ranges of the whole address space and dump the contents into a given file.

5.1 Limitations

1. *false positives* If some patches are applied to a binary/library that do not bump the version number to fix a particular vulnerability, VmiCVS does not detect that.
2. It is assumed that VM admins do not try to hide the version of binaries and libraries. That is VM admins do not morph the binaries and libraries to have different version than their original version.
3. certain softwares such as *libz*, *rpcbind* do not have their version string encoded either in their names or in their binary code. So it is not possible to obtain the version of such softwares using our *VmiCVS*

6. EVALUATION AND RESULTS

The main goals of our evaluation of VmiCVS are

1. To show the extraction of version number of system libraries which are not directly exposed to remote user through any protocols
2. To show the version number extraction of service or daemon binaries
3. Quantifying the disruption caused during version number scanning
4. Comparison of *Name Based* and *Virtual Memory* based Version Scanners

We have not considered *version parser* program for evaluation as it is a simple pattern matching script.

We ran our Cloud Vulnerability Scanner, VmiCVS on emulab's *d820* machine. It consists of Four 2.2 GHz 64-bit 8-Core E5-4620 Sandy Bridge processors with 7.20 GT/s bus speed, 16 MB cache and VT (VT-x, EPT, and VT-d) support. Memory includes 128 GB 1333 MHz DDR3 RAM (8 x 16GB modules) and 250GB 7200 rpm SATA disk, 6 x 600GB 10000 rpm SAS disks. It was running host operating system as Ubuntu 15.04 with kernel $3.19.0 - 16 - generic$. Virtual Machine had Ubuntu 14.04.3 LTS with $3.8.0 - 34 - generic$ kernel.

6.1 Version Number Extraction

We have chosen three different categories of software to verify and analyze our VmiCVS version scanners. In this section, we present the version number extraction of these three different pieces of software

- *libraries that contain version number in their filenames* As explained in the section 4, most of the shared libraries contain version number in their file names. Cyrus SASL libraries required by OpenLDAP's *slapd* and Linux PAM library needed by *sshd* contain version numbers in their file names. We extracted the version numbers of these libraries using *Name Based Version Scanners*.

  SASL libraries In our setup, Virtual Machine was running *slapd* linked to 2.0.25 version of the SASL libraries ;*liblogin.so*.2.0.25, *libplain.so*.2.0.25, *libdigestmd5.so* .2.0.25, *libsasldb.so*.2.0.25, *libsasl.so*.2.0.25. *Name Based version Scanner* dumped all the libraries linked to *slapd* by taking *slapd* as input as shown in the Figure 5.

  Linux PAM libraries Similarly, we ran version scanner to extract the version of *libpam* linked to *sshd* binary. It dumped all the shared libraries linked to *sshd* process.

- *libraries that do not contain version in their filename* In our setup, VM was running *init* process linked to *libc* library which had no version number in its filename. VmiCVS *Virtual Memory Based Version Scanner* dumped all the virtual memory pages of *init* into a file. It had the following string which contains version information :

  ```
  GNU C Library (Ubuntu EGLIBC 2.19-0ubuntu6.6) -
  stable release version 2.19, by Roland McGrath-
  et al
  ```

  Version Parser reported 2.19 from the above string.

- *service/daemon binaries* We chose two different kinds of daemons a) remote user facing daemons such as *sshd* and *slapd* b) internal daemons such as *rsyslogd* and *dnsmasq*. *Virtual Memory Based Version Scanner* dumped the virtual memory pages of these processes into a file.

  Main string in virtual memory dumped data file that contains the version numbers is shown below for the above chosen daemons

  sshd `OpenSSH_6.6.1p1 Ubuntu-2ubuntu2.6`

  slapd `buildd@lgw01-53:/build/openldap-2QUgtL/openldap-2.4.31/debian/build/servers/slapd`

  dnsmasq `Dnsmasq version 2.68 Copyright (c) 2000-2013 Simon Kelley`

  rsyslogd `rsyslogd %s, compiled with:`

Version number of any the shared library or daemon binary can be extracted using VmiCVS version scanners.

6.2 Impact

Virtual Machine needs to be paused during version scanning. As described in the section 3 and 5, version scanners consist of four phases; a) *Initialization* b) *Process Lookup* c) *Overlay instantiation* d) *Information Extraction*

Virtual Machine needs to be paused during *Process Lookup*, *Overlay instantiation* and *Information Extraction* phases. ***Name Based Version Scanner*** does not need to intantiate process overlay.

- ***Virtual Memory based Version Scanner*** Total VM pause time and time breakdown for the processes *sshd*, *init*, *syslogd* and *dnsmasq* is shown the Table 1. It shows that Virtual Machine needs to be paused for atleast `3 seconds` during version scanning. Process Overlay instantiation phase contributes maximum share to VM pause time. This can be reduced by re-engineering the stackdb implementation to instantiate the overlay from base target. Also as shown in the Table 2, size of the logfile(virtual memory pages content dump) is in the order of Mega Bytes and hence Information Extraction phase takes milliseconds to dump the memory pages. This can be optimized by dumping the memory pages to fast File Systems such as */shm*.

- ***Name based Version Scanner*** Total VM pause time and time breakdown for this scanner for the processes *sshd*, *init*, *syslogd* and *dnsmasq* is shown the Table 4. It shows that VM pause time is in the order of *milliseconds*. Also from the Table 3, it is evident that this scanner requires less storage to dump shared library file names.

6.3 Comparison of VmiCVS version scanners

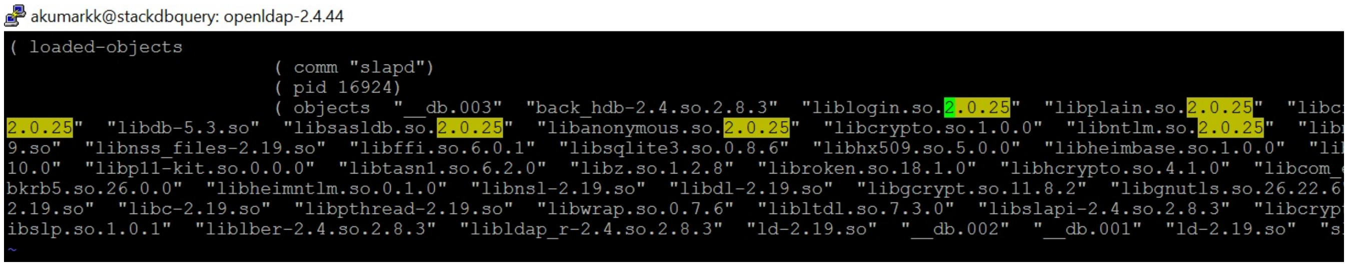
- *Disruption* It is measured by Virtual Machine pause time during version scanning. As shown in the Tables 1 and 4, *Virtual Memory Based Version Scanner* cause more disruption than *Name based Version Scanner* while pausing VM in the order of *seconds*.

- *Storage Name based Version Scanner* consumes less storage space.

- *Virtual Memory Based Version Scanner* can be used to scan the version of both the shared libraries and process binaries.

7. CONCLUSIONS AND FUTURE WORK

VmiCVS performs version analysis through VMI technique and reports the system vulnerabilities. It produces false positives when patches are applied to fix some vulnerabilities. Not all the libraries and binaries have their release version coded. In such cases, VmiCVS fails to report the vulnerabilities.These limitations of the VmiCVS



```
akumarkk@stackdbquery: openldap-2.4.44
( loaded-objects
                          ( comm "slapd")
                          ( pid 16924)
                          ( objects  "__db.003"  "back_hdb-2.4.so.2.8.3"  "liblogin.so.2.0.25"  "libplain.so.2.0.25"  "libc
2.0.25"  "libdb-5.3.so"  "libsasldb.so.2.0.25"  "libanonymous.so.2.0.25"  "libcrypto.so.1.0.0"  "libntlm.so.2.0.25"  "libr
9.so"  "libnss_files-2.19.so"  "libffi.so.6.0.1"  "libsqlite3.so.0.8.6"  "libhx509.so.5.0.0"  "libheimbase.so.1.0.0"  "li
10.0"  "libp11-kit.so.0.0.0"  "libtasn1.so.6.2.0"  "libz.so.1.2.8"  "libroken.so.18.1.0"  "libhcrypto.so.4.1.0"  "libcom_
bkrb5.so.26.0.0"  "libheimntlm.so.0.1.0"  "libnsl-2.19.so"  "libdl-2.19.so"  "libgcrypt.so.11.8.2"  "libgnutls.so.26.22.6
2.19.so"  "libc-2.19.so"  "libpthread-2.19.so"  "libwrap.so.0.7.6"  "libltdl.so.7.3.0"  "libslapi-2.4.so.2.8.3"  "libcryp
ibslp.so.1.0.1"  "liblber-2.4.so.2.8.3"  "libldap_r-2.4.so.2.8.3"  "ld-2.19.so"  "__db.002"  "__db.001"  "ld-2.19.so"  "s
~
```

Figure 5: Name Based Version Scanner's version data for *slapd*

Version Scanner Steps	Run Time in Seconds			
	sshd	init	dnsmasq	rsyslogd
Initialization	24.19	24.29	24.29	24.34
Process lookup	1.19×10^{-3}	0.08×10^{-3}	1.26×10^{-3}	0.57×10^{-3}
Overlay Instantiation	3222.96×10^{-3}	3215.59×10^{-3}	3167.85×10^{-3}	3320.45×10^{-3}
Information Extraction	85.27×10^{-3}	60.02×10^{-3}	46.00×10^{-3}	251.01×10^{-3}
Total time VM Paused	3309.42×10^{-3}	3275.69×10^{-3}	3215.12×10^{-3}	3572.04×10^{-3}

Table 1: Virtual Memory based Version Scanner Run Time Break down

process	logfile size in bytes
init	2,912,967
dnsmasq	1,544,569
rsyslogd	4,142,067
sshd	3,138,302

Table 2: Virtual memory pages dump Logfile size of processes

process	logfile size in bytes
init	404
dnsmasq	409
rsyslogd	406
sshd	663

Table 3: shared library file names Logfile size of processes

can be overcome by using code measurement or binary analysis techniques to confirm the presence of vulnerabilities.

Though there exists online tools [9] to list vulnerabilities corresponding to particular version of a software, it is necessary to link such tools to VmiCVS version scanners to list vulnerabilities corresponding to software version.

8. ACKNOWLEDGMENTS

We thank David M Johnson and Richard Li for their help on stackdb. We thank Prashanth Nayak for his indirect help. We have followed shared library listing technique(Name Based version Scanner) from his thesis work [16]. We also thank Jacobus (Kobus) Van der Merwe, Eric Eide, Anton Burtsev and Feifei Li for suggesting improvements.

9. REFERENCES

[1] 2015 data breach investigations report. https://msisac.cisecurity.org/whitepaper/documents/1.pdf.

[2] 7 ways vulnerability scanners may harm website(s) and what to do about it. https://www.whitehatsec.com/blog/7-ways-vulnerability-scanners-may-harm-websites-and-what-to-do-a

[3] The cracklib package. http://www.linuxfromscratch.org/blfs/view/svn/postlfs/cracklib.html.

[4] The cyrus sasl packag. http://www.linuxfromscratch.org/blfs/view/svn/postlfs/cyrus-sasl.html.

[5] Ghost vulnerability. https://access.redhat.com/articles/1332213.

[6] The linux pam package. http://www.linuxfromscratch.org/blfs/view/svn/postlfs/linux-pam.html.

[7] Openssl dos vulnerability, new bagel variants. https://isc.sans.edu/forums/diary/Updated+1345+318+GMT+OpenSSL+DoS+Vulnerability+New+Bagel+Variants/140/.

[8] Shellshock: All you need to know about the bash bug vulnerability. http://www.symantec.com/connect/blogs/shellshock-all-you-need-know-about-bash-bug-vulnerability.

[9] Vulnerabilities and version details. https://www.cvedetails.com/version-list/72/767/1/GNU-Glibc.html.

[10] Vulnerability assessment accuracy. http://www.beyondsecurity.com/va_accuracy_false_positive_negative.html.

[11] Vulnerability assessment guide. http://scitechconnect.elsevier.com/wp-content/uploads/2013/09/Vulnerability-Assessment.pdf.

[12] Vulnerability assessment whitepaper. http://www.beyondsecurity.com/pdf/AVDS_Whitepaper.pdf.

[13] Vulnerability assessment wiki. https://en.wikipedia.org/wiki/Vulnerability_assessment.

[14] D. J. Brown and K. Runge. Library interface versioning in solaris and linux. In *Proceedings of the 4th Annual Linux Showcase & Conference - Volume 4*, ALS'00, pages 40–40, Berkeley, CA, USA, 2000. USENIX Association.

[15] H. Holm, T. Sommestad, J. Almroth, and M. Persson. A quantitative evaluation of vulnerability scanning. *Inf. Manag. Comput. Security*, 19(4):231–247, 2011.

Version Scanner Steps	Run Time in Seconds			
	sshd	**init**	**dnsmasq**	**rsyslogd**
Initialization	24.06	24.17	24.15	24.25
Process lookup	2.00×10^{-3}	0.07×10^{-3}	1.70×10^{-3}	0.76×10^{-3}
Information Extraction	3.38×10^{-3}	0.05×10^{-3}	2.28×10^{-3}	2.76×10^{-3}
Total time VM Paused	5.38×10^{-3}	0.12×10^{-3}	3.97×10^{-3}	3.52×10^{-3}

Table 4: Name based Version Scanner Run Time Break down

[16] P. Nayak. Detecting and mitigating malware in virtual appliances. Master's thesis, University of Utah, 2014.