Reliable Library Identification Using VMI Techniques

Nick de Bruijn
Leandro Velasco

University of Amsterdam
Faculty of Physics, Mathematics and Informatics
MSc System and Network Engineering
Research Project: 1

February 7, 2017
Introduction

- Enhance cloud security
- Vulnerabilities in libraries can have major consequences
- Efficient way of detecting vulnerabilities in libraries is needed
To which extent can one reliably identify the version of a selected running library using the VMI techniques provided by LibVMI?

How can one identify a running library in a VM where the library name can not be trusted?
Related work virtual machine introspection:


Related work library identification:

Virtual machines

- Hypervisor has access to the binary representation of the virtual memory used by the OS running inside the virtual machine

Figure: Virtual Machine Architecture

Figure: Memory from the hypervisor’s perspective\textsuperscript{2}

\textsuperscript{2}C. A. Schneider. Full Virtual Machine State Reconstruction for Security Applications, 2013
Figure: Memory from the guest’s OS perspective

---

Virtual machine introspection

- Method to interpret/translate the hypervisor’s perspective
- Knowledge of the guest’s OS is needed
A virtual machine introspection library based on XenAccess
*(64-bit VM guest support, KVM support, fixes on bugs and memory leaks)*

Provides a useful application programming interface (API) for reading and writing to a virtual machines memory

Access memory using physical addresses, virtual addresses, or kernel symbols

Overcomes the semantic gap by providing the lacking information
*OS type, location of symbolic information, offsets used to access data*
1. Request to view kernel symbol
2. LibVMI finds the virtual address for kernel symbol
3. Kernel page directory mapped to find correct page table
4. Page table mapped to find correct data page
5. Data page returned to LibVMI Library
6. LibVMI returns the data requested

Figure: LibVMI memory mapping

^http://libvmi.com/docs/gcode-intro.html
Result of LibVMI:

1. Mapped virtual memory view
2. Access to the virtual memory
1. Version number extracting
   - Extract library version from its name or binary

2. Behaviour based identification
   - Look at behaviour of the library (system calls, wrapper functions)

3. Fingerprint identification
   - Extract information from a binary to create a fingerprint
   - Strict vs Fuzzy fingerprints
Printable strings:

- Uses a set of printable strings extracted from the library executable
  \((Error\ messages,\ copyright\ or\ usage\ information)\)

- Tian et al. show that such a list of strings can be an accurate signature of an executable object when used for malware classification

- Thomas Rinsma concludes this to be the most efficient method to identify libraries

- Printable strings can be extracted by using Unix \texttt{strings} command

- Measure similarity of sample sets using the Jaccard index:

\[
J(A, B) = \frac{|A \cap B|}{|A \cup B|}
\]
Figure: Example of strings obtained with the Unix command strings

cptrcент
__programe
mbrtoc32
__IO_free_backup_area
creat
setnetent
wcschr
__strxfrm_l
posix_spawn_file_actions_addclose
argp_err_exit_status
getgrgid_r
__vfprintf_chk
unshare
__seterr_reply
__recv_chk
__IO_getline_info
__fwriting
__finitel
__itoa_lower_digits
inet6_opt_finish
pthread_cond_init
__IO_default_xspun
Experimental Environment

The environment consists of:

- Privileged Host Dom0, in charge of performing the introspection
- Guest VM, system that will be introspected
The program consist of the following components:

- **Library extractor**: This module handles the *introspection* aspects required to extract the library binary from the guest VM memory. It does so by making use of LibVMI.

- **Library Identifier**: This module generates the *fingerprint* of the selected library and then compares it against the reference data base.

- **Reference Data Base**: It contains 151 fingerprints from different versions of different libraries.
Library Extractor Implementation

This module is in charge of:

1. Pause the VM to access the **guest VM memory** in a consistent way
2. Walk through the kernel data structures to identify the relevant memory pages
3. Dump the VM memory pages where the library binary was loaded
4. Resume the VM execution

Kernel Data Structures:

- task_struct
  - comm = htop
  - PID = xxx
  - mm_struct *mm

- mm_struct
  - vm_area_struct *mmap
  - map_count

- struct dentry
  - struct inode
    - d_name = libc-2.23.so

- struct path
  - struct vfs mount *mnt

- struct file
  - struct path
  - struct inode

- vm_area_struct
  - vm_start
  - vm_end
  - vm_area_struct *vm_next
  - vm_flags
  - struct file *vm_file

- memory_page
  - Binary code
This module is in charge of:

- Generate a fingerprint from the extracted library. This is done by executing the Unix command \textit{Strings}.

- Calculate the Match Score for each fingerprint in the reference DB
  
  \[ \text{MatchScore} = \frac{|\text{Sample} \cap \text{Reference}|}{|\text{Sample} \cup \text{Reference}|} \]

- Sort the results and return the top five Match Scores
The following step were followed to create the DB:

1. Download the **source code** from different versions of different libraries. Including the ones that will be tested (**libc** and **libncurses**)

2. Build the different libraries by only passing the argument
   - `- - prefix=<directory>`

3. Generate a fingerprint for each share object created during the building procedure. This is done by executing the Unix command **Strings**
The low match scores are due to the way the DB was built and the fact that some pages may be swapped out.

The match score obtained with the original .so that was loaded in memory is: 97.06%

Less than 9% is considered a mismatch.
Efficiency and Effectiveness Experiments Design

- The program was executed 100 times per load configuration and per library (libc-2.23 or libncurses-5.9)

- Each load configuration represents either the hypervisor’s CPUs or the guest VM’s CPUs stressed at 0% (low), 50% (mid) or 100% (high)

- Data gathered during the experiments:
  - Pause Time
  - Identification Time
  - Memory Usage
  - CPU Usage
  - Match Score

- For each of the above values, the mean and the standard deviation were calculated

- Two extra experiments were executed in which either the hypervisor’s memory or the guest VM’s memory was stressed at 100%
Pause Time Results - libc

VM Low Load

<table>
<thead>
<tr>
<th>Hypervisor Load</th>
<th>Time in (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low</td>
<td>97.65</td>
</tr>
<tr>
<td>Mid</td>
<td>146.5</td>
</tr>
<tr>
<td>High</td>
<td>312.3</td>
</tr>
</tbody>
</table>

VM High Load

<table>
<thead>
<tr>
<th>Hypervisor Load</th>
<th>Time in (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low</td>
<td>99.81</td>
</tr>
<tr>
<td>Mid</td>
<td>200.5</td>
</tr>
<tr>
<td>High</td>
<td>416.4</td>
</tr>
</tbody>
</table>
Identification Time Results - libc

VM Low Load

Time in (ms)

Hypervisor Load

Low 320.3
Mid 485.5
High 785.2

VM Low Load

Time in (ms)

Hypervisor Load

Low 325.8
Mid 642
High 1018.7
Effectiveness Results

- Match scores are not affected by the CPU load
- However they are affected by the memory load as shown in the following table:

<table>
<thead>
<tr>
<th>Library</th>
<th>Not Stressed Memory</th>
<th>VM Memory at 100%</th>
<th>Xen Memory at 100%</th>
</tr>
</thead>
<tbody>
<tr>
<td>libc</td>
<td>20.606% ±0.007</td>
<td>20.585% ±0.007</td>
<td>20.248% ±0.005</td>
</tr>
<tr>
<td>libncurses</td>
<td>15.500% ±0.000</td>
<td>15.500% ±0.000</td>
<td>15.489% ±0.020</td>
</tr>
</tbody>
</table>

**Table:** Effectiveness Under Heavy Memory Load
Library Tampering Experiments (1/2)

• Are the strings containing version information relevant to the library identification?

<table>
<thead>
<tr>
<th></th>
<th>GLIBC_2.22</th>
<th>GLIBC_2.23</th>
<th>GLIBC_2.24</th>
<th>glibc 2.24</th>
<th>NPTL 2.24</th>
</tr>
</thead>
<tbody>
<tr>
<td>GNU C Library (Ubuntu GLIBC 2.24-0ubuntu5)</td>
<td>stable release version 2.24, by Roland McGrath et al.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure: Example of strings containing version information of libc-2.24

• Manually tamper the sample fingerprint to include strings containing version information of libc-2.24

<table>
<thead>
<tr>
<th>Sample Fingerprint</th>
<th>Libc-2.23 Ref. Fingerprint</th>
<th>Libc-2.24 Ref. Fingerprint</th>
</tr>
</thead>
<tbody>
<tr>
<td>libc-2.23 original</td>
<td>20.60%</td>
<td>19.82%</td>
</tr>
<tr>
<td>libc-2.23 tampered</td>
<td>20.59%</td>
<td>19.83%</td>
</tr>
</tbody>
</table>
Remove every string containing version information from the sample and reference fingerprint

<table>
<thead>
<tr>
<th>Match</th>
<th>Fingerprint in the DB</th>
</tr>
</thead>
<tbody>
<tr>
<td>20.59%</td>
<td>libc-2.23.so.strings</td>
</tr>
<tr>
<td>19.73%</td>
<td>libc-2.22.so.strings</td>
</tr>
<tr>
<td>19.71%</td>
<td>libc-2.24.so.strings</td>
</tr>
<tr>
<td>19.34%</td>
<td>libc-2.21.so.strings</td>
</tr>
<tr>
<td>18.78%</td>
<td>libc-2.20.so.strings</td>
</tr>
<tr>
<td>18.25%</td>
<td>libc-2.19.so.strings</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Match</th>
<th>Fingerprint in the DB</th>
</tr>
</thead>
<tbody>
<tr>
<td>20.54%</td>
<td>libc-2.23.so.stripped</td>
</tr>
<tr>
<td>19.70%</td>
<td>libc-2.22.so.stripped</td>
</tr>
<tr>
<td>19.68%</td>
<td>libc-2.24.so.stripped</td>
</tr>
<tr>
<td>19.31%</td>
<td>libc-2.21.so.stripped</td>
</tr>
<tr>
<td>18.74%</td>
<td>libc-2.20.so.stripped</td>
</tr>
<tr>
<td>18.22%</td>
<td>libc-2.19.so.stripped</td>
</tr>
</tbody>
</table>

Table: Normal Scenario

Table: Stripped Scenario
Implementation Limitations

1. Unix only

2. One to many comparison

3. Dynamically linked libraries only

4. Identification time directly depend on the amounts of records in the reference database

5. LibVMI offsets requires guest kernel access

6. Swapping of memory pages affect the results

7. When a library that is not included in the reference database goes through the identification process, false positives can be observed
Conclusion

1. LibVMI can be used to efficiently extract libraries from the VM’s memory.

2. Printable strings can be used as fingerprints to accurately identify a library when the library is in the database.

3. Performance measurements show that our implementation performs in a reasonable manner, even under high system load.

4. Accuracy of identification was not affected by the load of the system.
Future work

- Explore ways to;
  - improve the database creation to obtain better matching results
  - improve the scalability of the program
  - identify library behaviour using VMI techniques

- Extend the functionality of our program to support vulnerable library scanning