Cross VM Covert Channel Implementation

Dewank Pant
Johns Hopkins University
1 PUBLICATION **0** CITATIONS
SEE PROFILE

Jibraan Singh Chahal
Johns Hopkins University
3 PUBLICATIONS **4** CITATIONS
SEE PROFILE

Manan Wason
Johns Hopkins University
2 PUBLICATIONS **13** CITATIONS
SEE PROFILE

Covert Channels

Dewank Pant
Information Security Institute
Johns Hopkins University
dewankpant@jhu.edu

Jibraan Singh Chahal
Information Security Institute
Johns Hopkins University
jchahal1@jhu.edu

Manan Wason
Information Security Institute
Johns Hopkins University
mwason1@jhu.edu

ABSTRACT
Covert channels are methods by which data leak can be achieved even across the environments which are considered safe in general. Covert channels make use of both genuine as well as non-genuine methods for such data transfer. Due to their covert nature, they often remain unnoticed and are extremely hard to detect. Additional measures can also be taken to make the detection even harder. Covert channel is a massive threat to confidential data if left undetected. However, covert channels face extremely slow data transfer and very poor noise vs data ration. In the following paper, we have implemented a covert channel which is able to transfer bits successfully between two separate virtual machines existing on a single physical machine. We additionally, tested the performance by iterating the data transfer multiple times to calculate the accuracy of the channel.

Keywords
Virtual Machine Manager, VMware, Covert Channel.

1. INTRODUCTION

Covert channels are a hidden way of communication where two instances of virtual machine (VM) can achieve data transfer via unconventional means of communication by leveraging the use of shared resources. As the virtual machines hosted on a single physical system share resources like disks, memory and CPU therefore, one co resident VM can predict the messages or behaviour of another co resident VM by analysing the latency it faces while accessing the shared resources. Thus, we were able to exploit the co residence and achieve a highly accurate data transfer based on the established covert channel. We implemented the channel using python by creating a sender and receiver script which established the covert channel. Additionally, we were able to enhance the efficiency of the channel by creating a guest machine over the host OS and then making two separate virtual machines over the guest OS thus making use of lesser RAM and a single processor core, which made our results more accurate and noise free. We used VMware v14 and installed Ubuntu as a guest OS. Further, we installed VMware over our guest OS and then installed two additional Kali Linux guest OS over Ubuntu which made our entire system extremely isolated from noise and made the channel even more reliable.

.

2. COVERT CHANNELS

Covert channel is a category of attack that exploit the existing channels of communication by constantly analyzing fluctuations in the access rates of shared resources. Covert channels are hidden and nearly undetectable due to their usage of existing channels which are not intended to communicate in the way they are exploited. If there are shared resources between multiple processes then the latency that occurs in the access rates of those resources can likely act as a covert channel. Foremost issue with such type of information leak is that these communication channels can't be detected and the extent of severity and information leakage on such systems can't be known with accuracy.

NCSE (National Computer Security Center) established by the NSA (National Security Agency) introduced the TCSEC criteria, where TCSEC stands for Trusted Computer Security Evaluation Criteria. TCSEC defines to types of covert channels namely:

Storage Channels: The type of channels where the communication is established when two virtual machines modify a resource based on the storage location. E.g.: Hard disks.

Timing Channels: These are the type of channels that establish communication based on the varying latencies and the real-time response delays faced by the receiver.

The two types of channel are elaborated in detail below.

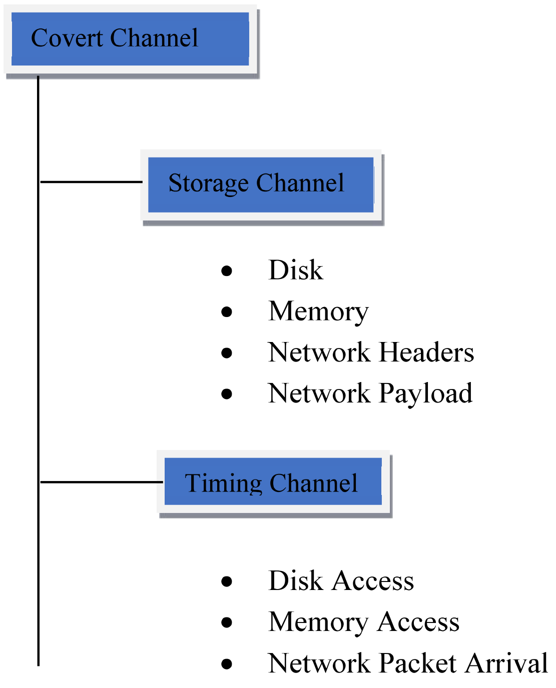
2.1 TIMING CHANNEL

For a covert channel to satisfy the conditions to be considered as a timing channel, it should involve real-time response delay on the receiver's end. Thus, in case of timing channels when sender system is writing or accessing certain resources then the receiver system will eventually extract information from the channel by analyzing the real-time delay in the response time.

The time based covert channel can have a serious flaw if multiple processes are running on the sender system as that might create a lot of noise and make the receiver receive anomalous data. However, such drawback can be overcome by accessing the channel at a time when there is less traffic, i.e.: a situation where lesser number of process are being executed on the sender.

2.2 COVERT STORAGE CHANNEL

For a covert channel to be considered a storage based channel, it should specify the minimum condition for the sender and receiver working on the same physical shared resource. In a typical storage channel one system writes to a shared resource and the other reads from it. One of the very well-known case for this type of channel is the example of printing queue. In case of a printing queue one process of high level is able to fill up the printing queue with a value of 1 when there is a printing job and goes back to the value 0 when its idle. Therefore, in such case any lower privilege system can get the exact status of printing queue directly by polling.



3. LITERATURE REVIEW

For the following assignment, we reviewed the following research paper titled "Hey, you get off of my cloud: Exploring Information leakage in third-party compute clouds". This research explains in detail how the authors have tested covert channels and side channel attacks in the cloud service provided by Amazon. The authors were more focused on achieving co-residence parallel to a machine running on the same base physical system on the Amazon's EC2.
In the research, the authors followed the approach of an attacker and tried to achieve co-residence in order to get a malicious machine running parallel to the victim's machine running together on the same physical system. Amazon provided services in 2 regions (USA and Europe). Both the regions consisted of 3 separate zones and that additionally had five different Linux instances. Thus, the authors were able to explicitly specify the zones and instance types.
Further, to achieve co-residence the authors discovered cloud cartography by doing network probing. Additionally, they determined co-residence by following observations:

- Checking the packet round-trip duration.
- Checking how close internal IP address are.
- Matching the Dom0 IP address.

4. EXPERIMENTAL SETUP

The covert channel experiment was conducted on a MacBook Pro Host machine with MacOS High Sierra operating system. The host machine had the following specs:
RAM: 16 GB 2133 MHz LPDDR3
SSD: 1Tb,
CPU: 3.5GHz 7th-generation Intel Core i7 processor

Two virtual machines installed directly over the host machine were not able to give us perfect isolation. This is because running two VM's on the host machine with 1 core assigned to each did not guarantee that core to be the same. Thus, to overcome this problem we went one step further and installed two guest OS over our "host VM" making 100% sure that we were dealing with processes taking place in the same core of the processor. This step substantially decreased the level of noise in our covert channel due to 3rd degree of isolation. As the host VM was given a single core, the two VM's running on it were bound to share the same core. VMware v14 was used as the virtual machine manager for both the host VM and testing VM's in it. The OS on the host VM was Ubuntu and the two VM's inside were Kali. The virtual machines had the following specs:
Ubuntu:
Hard Disk: 251 GB SSD, Memory: 4 GB //TODO processor/core
Kali:
Hard Disk: 25GB SSD, Memory: 1 GB
We called this *"inception"* of VMs as there were 2 VM inside another VM. We observed that this is a really good method to achieve greater level of isolation and noise free environment to create a proof of concept for covert channel. Additionally, we created large txt files for both the virtual machines. The file was taken large so we are able to detect noticeable fluctuations in the file read and write time. Thus, able to communicate via covert channel.

5. EXPERIMENT

The experiment was carried out using two python scripts, sender.py and receiver.py on the sender virtual machine and the listener virtual machine respectively. For reading the file on both, we used the module MMAP. This module maps the file to memory-mapped file which is a part of virtual memory that is

correlated byte-by-byte to the file. This correlation between the file and the memory space permits applications to treat the mapped portion as if it were primary memory. By using this module, we can access each character of the file just like by passing an index to an array. The index starts from 0.
We observed that reading just one memory location was too fast, so instead we decided to read a chunk of the file for both sender and receiver in order to achieve an observable I/O contention. The receiver always reads from a fixed location; a fixed chunk of the file. The sender randomly chooses a chunk of the file to read from, when sending the one bit. These random locations were decided by random numbers as start and end indices of the mapped file array. The random numbers were kept under threshold, because if they were too close together, there was not enough competition for the disk. As there was noise whenever the receiver was reading even with the sender turned off, the mmap read function in the sender was looped over a couple of times. Reading multiple times did increase the time to send the '1' bit however, it differentiated the actual bits from the noise and gave a significant latency. For sending a '0' bit the sender was put to sleep for 4.5 seconds. This time is close to double of the time the receiver takes to read the fixed chunk without influence.
The message to be sent was padded with a '1' bit at the beginning and at the end. This was done to make the message have a specific start and end. On the receiver side after reading the message the leading and trailing bit was removed. This made the messages starting or ending with a '0' bit to be read correctly as without the padding the receiver would not know where the message ends.

5.1 ALGORITHM

We have a sender and a receiver python script where the sender is writing bits to our covert channel and the receiver is reading the same. The receiver is using a complex algorithm to figure out the sequence of bits being written by the sender VM and we will now discuss the algorithm. We first define a user input that corresponds to the bits the receiver VM is to read in this stream. The bits can be figured out on the basis of 'read time values'. These 'read time values' are defined as the time taken to read the file for a fixed number of times in the receiving VM. These time values are then analyzed to figure out the bits sent by the sender. The way to determine this is that we experimentally calculate a time threshold that helps us categorize a bit as a '0' or a '1'. If the observed time is greater than the threshold for 6 consecutive reads, we categorize it as a '1' and if it is smaller for two consecutive reads, it's a '0'. The only purpose of checking for consecutive reads is to minimize noise in our covert channel. This noise might arise from various factors like background processes, cursor movement and keyboard input to name a few.

Through our experiments, we determined the standard threshold to be 2.8 seconds. Also, since each iteration read time values won't necessarily be below the threshold, it was rounded to the first decimal place. This removes the ambiguity in determining our results.
While reading a bit stream, a '1' prefix and suffix is added that acts as padding. This padding signifies the start and end of the stream which makes it easier to read the correct bits. This doesn't have to be taken care of by the user as the code takes care of it.
Now to the actual implementation, the algorithm start recording the time values as soon as the script is started. The read times are recorded until the user specified time and then store it into a python list. When this is done, the list is analyzed to retrieve the bit stream. This is done by going over the length of the list and checking to see if we get 5 read time values greater than the threshold or 2 time values lower than our threshold. The former will count as a '1' and the latter counts as a '0'. Once we encounter a '1' or a '0', we start a counter to determine the occurrences of the same. A pseudocode implementation of our code goes as following**:**

```
times = list()
prev = -1;
flag = 1;
while start < times.length():
  Count = 0;
w_end = w_start;
while round(times(j) > threshold):
  if (w_end + w_start == len(temp)):
    break;
  else :
    w_end = w_start + 1;
count = count + 1
If count >= 5:
  If prev != -1:
  for k from 0 to w_start - prev / 2:
  print flag;
flag = 1;
for k from 0 to w_end - w_start / 5:
  print flag;
flag = 0;
prev = w_end;
else :
  for k from 0 to w_end - w_start / 5:
  print flag;
flag = 0;
prev = w_end;
w_start = w_end;
else :
  w_start = w_start + 1
```

6. PERFORMANCE

In this experiment, we were able to achieve quite high bit transfer rate between the VMs correctly. The error rate started to increase rapidly when the message was longer than 18 bits. In addition, whenever in the message there were more than four consecutives '0's, one of them seemed to drop now and then. The reason behind this was the noise. Normally, two reads lower than the average threshold meant that a '0' was transmitted. But when there was noise the receiver took more time in one read and in the meantime the sender sends another

‘0’. This caused the bit sent during the noise to be missed. Some factors that added extra noise and deviation of the test results from each other are listed below:

Applications running in the background

Running tests with Wi-Fi/Bluetooth on and off made difference.

Moving the mouse pointer while the test was going on

Battery of the machine to go under 5%

Whether the machine was plugged in or not

It was seen that the average threshold could not be taken as the average of some number of read times without sender sending any data. This was because while reading for quite a while, there was noise and it made the average go quite high. The threshold is an experimental value decided by observing the read time for ‘0’ bit in between the ones and ignoring the noise; for this experiment the threshold was set to be 2.8 seconds. When the sender wants to send a ‘1’ bit, the chunk of the file to be read cannot be completely randomized. This is because if the size of the chunk turns out to be very large or very small the read time will deviate and so will the I/O contention. To overcome this the randomization is done under size constraints so that the chunk, along with being random, does show erratic behavior. We also observed that with keeping all the parameters on the receiving side to be same there was a tradeoff between the transfer time and accuracy. Increasing the read time for each bit in the sender, there was an increase in the accuracy. Therefore, in this experiment, we can have either a fast and less accurate covert channel or more accurate but slower

7. OBSERVATIONS

In Figure 1. It is shown how the error rate is dependent on bandwidth.

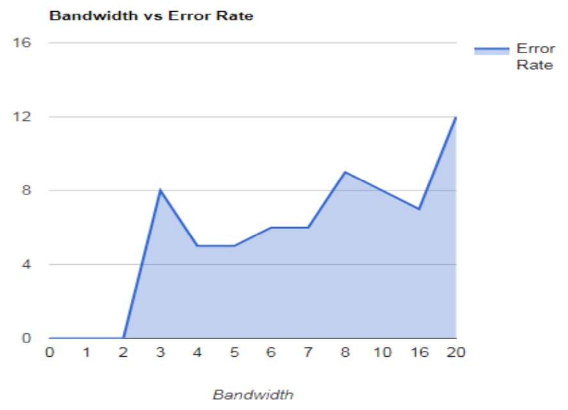
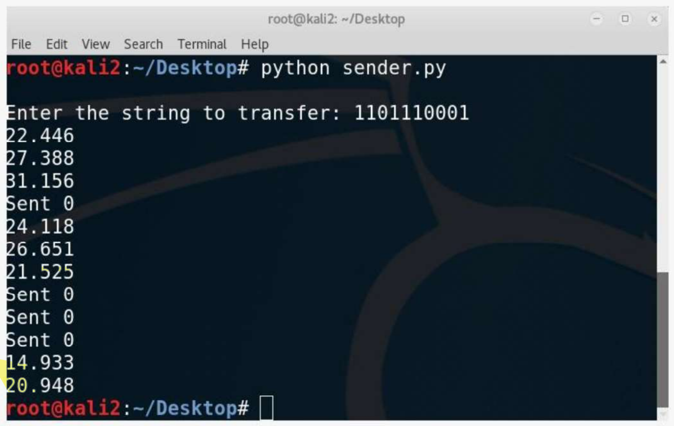


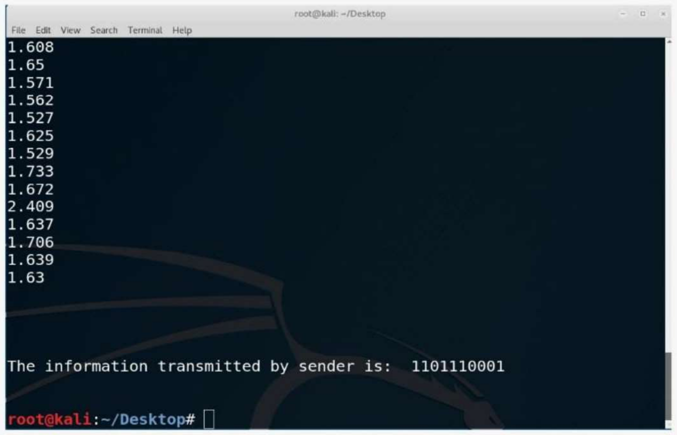
Figure 1



```
root@kali2:~/Desktop# python sender.py

Enter the string to transfer: 1101110001
22.446
27.388
31.156
Sent 0
24.118
26.651
21.525
Sent 0
Sent 0
Sent 0
14.933
20.948
root@kali2:~/Desktop# 
```

Figure 2



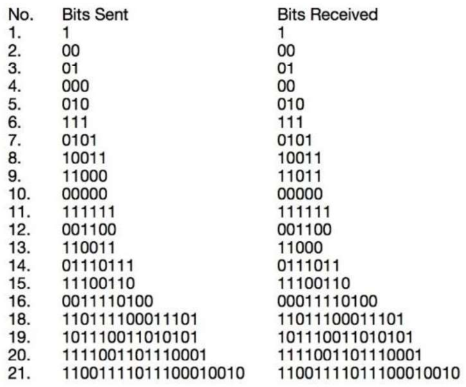
```
1.608
1.65
1.571
1.562
1.527
1.625
1.529
1.733
1.672
2.409
1.637
1.706
1.639
1.63



The information transmitted by sender is:  1101110001

root@kali:~/Desktop# 
```

Figure 3



No.	Bits Sent	Bits Received
1.	1	1
2.	00	00
3.	01	01
4.	000	00
5.	010	010
6.	111	111
7.	0101	0101
8.	10011	10011
9.	11000	11011
10.	00000	00000
11.	111111	111111
12.	001100	001100
13.	110011	11000
14.	01110111	0111011
15.	11100110	11100110
16.	0011110100	00011110100
18.	110111100011101	11011100011101
19.	101110011010101	101110011010101
20.	1111001101110001	1111001101110001
21.	11001111011100010010	11001111011100010010

Figure 4

8. CONCLUSION

In this paper, we have tried to cover and discuss covert channels, their types and limitations. We also described our successful attempt and implementation of the storage covert channel.
In the experiment, we see found that by implementing covert channel over two VMs hosted on another VM we can decrease the error rate as the two VMs are bound to share the same core of the processor. However, doing this decreases the transfer rate of the bits.
Transmitting same bits consecutively, especially '0' was difficult and was more prone to error rate. This was because of the noise in the channel. When the receiver is reading '0's and noise starts, it takes more time than the normal read. And at the same time the sender sends another '0' bit that is missed.
During some tests, the noise sometimes led to a false positive when its read time crossed the average threshold. This problem was solved by stating that a '1' bit is received only if six continuous over the threshold reads have been executed. It works because the noise in the system do not last this long.

9. FUTURE WORK

In the future, we would also like to implement timing based covert channel and further try to analyze which one of the two gives a better result and which is suitable for what situation.
We additionally intend to implement video streaming over a covert channel as we saw at a defcon conference.

10. REFERENCES

1. Gasser, Morrie.*Building a Secure Computer System*. Van Nostrand Reinhold Co, New York, 1988, 67-71.
2. Jaeger, Trent. *Operating system security*. Morgan & Claypool Publishers, San Rafael, 2008, 72.
3. Exploring Information Leakage in Third-party Compute Clouds. In Proceedings of the 16th ACM Conference on Computer and Communications Security, (CCS '09), 199–212.
4. Ristenpart, T., Tromer, E., Shacham, H., and Savage, S. Hey, You, Get off of My Cloud
5. Wu Z, Xu Z, Wang H. "Whispers in the hyper-space: High-speed covert channel attacks in the cloud." In USENIX security Symposium 2012 Aug 8
6. https://en.wikipedia.org/wiki/Memory-mapped_file].
7. Code : https://github.com/mananwason/Cache-based-Covert-Chann