Secured Distributed Shared Memory

Project Progress Report

Ashwin Krishnakumar Ramprasath Pitta Sekar Rohit Murali

1. Problem

Distributed Shared Memory mechanism implemented using the MSI protocol pairs memory regions using sockets. Use of sockets is prone to attacks on the data present in the shared memory region. Attacks such as the side channel attacks [5], Buffer overflow attacks [6] and tampering of network data on the fly [7] are some examples of how data integrity can be compromised. In [5], the major purpose of sharing identical memory pages between the processes being executed is to reduce the memory footprint of the system.

Interference because of shared pages in a system resulting from the shared use of the processor cache is mentioned in [5]. The cache behaviour of processors is used to extract information on access to shared memory pages in Side Channel attacks. The contents of a shared page in a particular memory location are cached whenever it is accessed by a process executing in the system. Side channel attack is implemented to remove the shared memory locations from the cache and then observe whether the same contents are loaded back into the cache so get information about which shared page is being accessed in the memory.

Buffer overflow attacks mentioned in [6] is mainly caused due to the issues involved in handling page faults. Overflow issues caused by overwriting the memory of an application is exploited in this attack. This helps the attacker get access to more private files and information. Stack-based buffer overflows attack is the most common example of this type. They leverage stack memory that only exists in the system during the execution time of a function. Heap-based attacks are harder to carry out and involve flooding the memory space allocated for a program beyond memory used for current runtime operations.

[7] deals about the tampering of network data on the fly using attacks such as Man in the middle, Denial of service, etc to modify the information which is being sent. Such attacks are implemented by finding out sensitive information present in the TCP/IP packets which are transferred for updating the shared memory system. Once this data is found out, the attacker can act spoof TCP data packets and act as one of the communicating processes. The attacker can then reset the connection from the original process and continue to keep communicating without the other process realizing that it is sending sensitive data to a third-party user. A hash-based comparison of incoming and outgoing packets of the system is what we propose to detect such an attack. Whenever the data is requested, the sender sends both the data and the signature which is later verified by the requesting machine for any corruption. Through this we can detect side channel and buffer overflow attack. The network data packet when on the air or wire, can be attacked using various networking attacks. Once the data is received at the requesting machine, we verify the signature and can raise an attack alert on the infrastructure.

In this project, RSA based signature verification is included for data security related to distributed memory pages. During synchronization request, the current design requests the page and copies the content providing sharing of memory across machines but does not validate the data content which is shared between the systems. We are currently including digital signature verification [3] for the shared page contents. This will detect and prevent on the fly modification of the distributed data which will ensure integrity of the shared data. We utilize the RSA method [4] for signature verification and SHA256 for hash calculations.

2. Motivation

The implementation of cyber security into modern day systems has been on the rise due to a variety of attacks on existing IoT devices and an increasing number of devices being connected to the internet. [1] provides vulnerability analysis whereby, evidence is provided regarding a myriad of attacks through wireless connecting systems such as Infotainment system, Bluetooth, and cellular radios. This paper talks about attack executed using the infotainment system which leads up to remote control of entertainment system and various critical applications. The issue of cyber-physical security comes into picture where each application needs to improve its security standards in order to avoid creating a security loophole which can be exploited to compromise other systems as well. Such attacks can lead to of denial of safety critical services such as braking systems and engine ECU where the access to these systems are obtained by exploiting other systems. Since protecting the system from wireless and wired attacks requires a well-planned and secure communication interface. This interface must be protected from available to malicious inputs from unknown entities.

The JEEP car attack [9] is the best example of an infamous attack penetrating the system. It attacked the compromised infotainment system to control features of the car which only a driver can perform such as controlling the breaks, turning on wipers even worse, stopping the car on the highway, all wirelessly. The driver's input could be overridden by the attackers to deny him any service. There was a wireless network that had an open port and was performing blank communication without any digital security scheme. This communication did not encrypt the data or even worse use any verification scheme, because of which the hackers were able to pump data into CAN bus without any problem. The hackers were able to record the messages in the vehicular network which were a bunch of unencrypted messages and were ultimately able to construct an event-based control database. Using the database, the hackers managed to gain control of the car up to the level of causing physical injury to the driver. This project's intention on making the existing Distributed Shared Memory is to avoid providing attackers any passage to modify the shared memory and pump malicious data to the system. This is not contained to prove beneficial to automotive applications, but also to any IoT devices which are usually developed without any precaution on the effect of attack surface exposure.

Although the implementation of data security in our project is to provide cryptographic solutions to be implemented in IoT devices, there are other problems that follows these small scales embedded and SOC chips solutions. Systems with high computational capacity like server machines utilize complicated mechanisms such as firewalls to protect the systems. Porting firewall scale solutions into embedded systems is difficult due to the resource constrains. Recently firewall equivalent mechanisms are brought into the vehicular network, but these solutions are not yet developed to an achievable security standard [2].

The future in the vehicle industry will have autonomous vehicles where a greater number of sub systems would be connected to the internet and because of which much more additional security measures would be required. [10] talks about the discussions on identifying the misbehaviour of software entities that are a part of the network. The importance of packet analysis in observing TCP packets during socket communication, to identify presence of malicious entities in network is discussed in this paper. This paper uses cipher suits effectively to identify the presence of attacker. The importance of digital cryptography has taken an important research topic of interest as the world as it moves towards more connectivity. The recent trend has included cipher suite-based systems for digital signature and MAC verification in order to preserve message integrity and verification. We have discovered research trends on implementing techniques such as port knocking for connected embedded systems [11].

The incoming of future technologies which will address the resource constraints present in current embedded systems, will improve the efficiency in performing digital crypto schemes. In a recent update on resources present in microcontrollers all microcontrollers are equipped with the CRC module and AES encryption hardware solutions to avoid the performance overhead system design. Currently architectures are present in embedded systems which are implementing processes in dedicated and isolated parts of controller that cannot be accessed using direct memory access which leads to additional security. This is known as the Trusted Execution Environment. The security of system is improved up to the System Bus Level. TEE like environments greatly improve the overall system security as the instructions executed inside the TEE are isolated and not accessible to other processes in the system, because of which attacks such as ROP and buffer overflow, and control to overwrite protected parts of memory can be avoided.

3. System Implementation

The implementation of Distributed Shared Memory system is done using a TCP network socket to communicate the shared data between the two systems. The same application can be made to run as a server, or a client based on the command line argument provided by the user. Once the IP and port number are validated, the application goes into server or client mode based on the input. The number of pages to be allocated for memory sharing between the systems is based on the user input on the server side and is followed by the server waiting for the client connection. When the client connects to the socket, the details regarding the shared memory region i.e. the length of the memory, address and page count are sent from the server to the client. The client receives the data and creates the memory mapping, at the same memory region as the server. Once this transfer of data is successful, both applications register the memory to userfaultfd system call, in order to address page faults in case of page access. Separate threads are used for monitoring and page fault handling on both the applications. The applications maintain a global 2d-array in order to store the page numbers and their current MSI status. Every time either of the nodes want to access the memory for read or write, its current status is checked before proceeding with the requested action. When the user wants to write in one of the node's pages, the node marks the page as "modified" on its side, and sends the other node, a request to "invalidates" the page's entry on its side. When this request is received, the corresponding node marks the page as invalid in its global array and executes the madvise system call on the particular page location, so that a pagefault occurs on its next access. When we request to read a page on a particular node, if the status of the page is "invalid" on a that node, the node requests the other node for the corresponding page data. The node on which the request is made sends the requested page data and marks the page as "shared". The requesting node on receiving the data, also marks the data as "shared", marking successful sharing of data between two nodes. This implementation has been depicted in the Figure 1.

The implementation of the standard MSI protocol was redesigned by us in order to provide additional security. The original implementation of the MSI protocol does not have any security features to secure the data being transferred between shared memory regions. In order to address this issue, we have currently decided to proceed with RSA signature scheme for creating digital signatures to verify the data being shared. We proceed with a straightforward implementation of RSA [8]. This implementation of the RSA algorithm will not be computationally expensive which will be ideal to be included in vehicular networks. We are able to encrypt and decrypt the data out successfully using this algorithm. This was done on two separate systems having each other's public key and sending a sample data. The APIs provided by OpenSSL library were used to successfully encrypt and decrypt the message sent between two nodes.

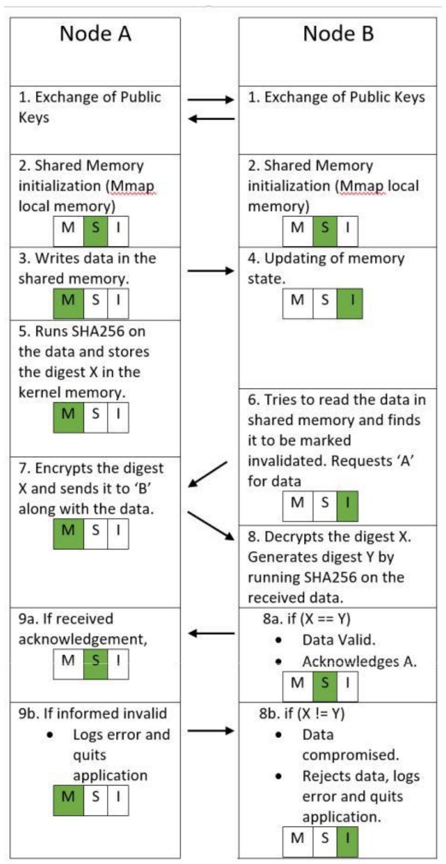

Node A
Node B
1. Exchange of Public Keys
1. Exchange of Public Keys
2. Shared Memory initialization (Mmap local memory)
2. Shared Memory initialization (Mmap local memory)
M S I
3. Writes data in the shared memory.
4. Updating of memory state.
5. Runs SHA256 on the data and stores the digest X in the kernel memory.
6. Tries to read the data in shared memory and finds it to be marked invalidated. Requests 'A' for data
7. Encrypts the digest X and sends it to 'B' along with the data.
8. Decrypts the digest X. Generates digest Y by running SHA256 on the received data.
9a. If received acknowledgement,
8a. if (X == Y)
Data Valid.
Acknowledges A.
9b. If informed invalid
Logs error and quits application
8b. if (X != Y)
Data compromised.
Rejects data, logs error and quits application.

Figure 1 DSM without Security

We calculate the signature of a memory content in order to create a digest for the memory. Before marking any page as modified, the hash value corresponding to it is calculated. For keeping the hash value secure we plan to store it inside the kernel. When the contents of a page need to be accessed, we send the signature generated along with the page contents to the other node. The signature is calculated based on the hash value stored inside the kernel. In order to verify that the correct data is transferred, when the requesting node receives the signature, it is decrypted and compared with the locally calculated hash value. When both the values match, we then only mark it as shared. Using this mechanism, we are able to provide and ensure data integrity. We have not modified any part of the proposed project, but we have added storage of hash values inside the kernel. While implementing this algorithm we consider that the kernel is not compromised and is a trusted part of the system. This is due to the reason that there are strict boundaries across kernel memory in the system and to access this data, the only way is to go through the system call. We have created a custom system call to create a X Array of cryptographic sensitive data as shown in Figure 3. For the purpose of scalability, a unique token (for example like a file descriptor) is created to choose the signatures that the application requested the memory to be created for. The corresponding list will be iterated over whenever a particular page is being requested by another machine. We have allowed the reading of the hash to be done only when the data is being requested or when the status of the page is modified in the requested machine. This is done in order to ensure the right order of operations. The flow of this execution is depicted in Figure 2 below.

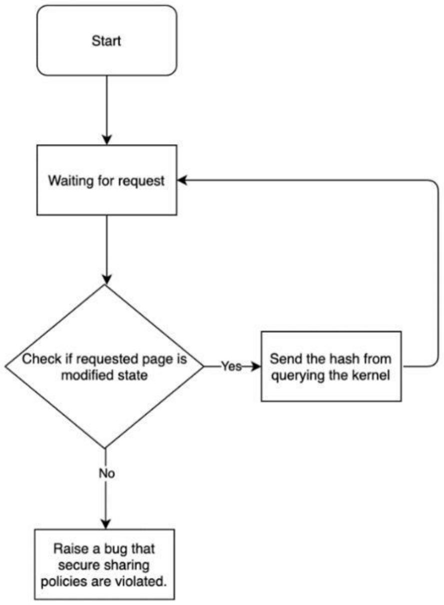

Start
Waiting for request
Check if requested page is modified state
Yes
Send the hash from querying the kernel
No
Raise a bug that secure sharing policies are violated.

Figure 2 Hash accessing flow

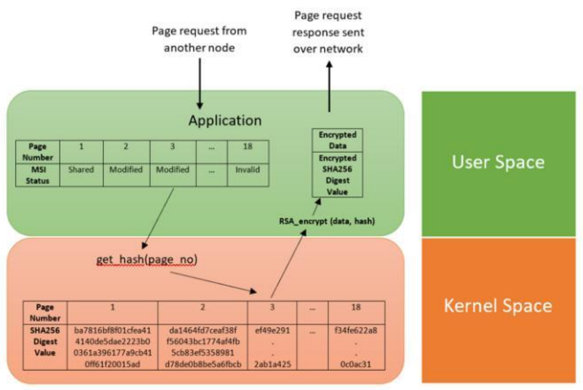


Figure 3 Sensitive data Isolation

System Call implementation –

The hash value on each of the page is stored in the Linux kernel. This is achieved by a system call – store_hash() which enables storage and retrieval of hash values from user level to kernel level. The system call store_hash() takes in 3 parameters – options, page_value, hash. The option parameter is used to specify whether we want to create the entries or read a hash value or write a hash value. The second parameter page_value is used to specify either number of pages in case of creations of pages, the page number in case of read or write. In case of write, the hash value of the page to be stored is given as the third parameter.

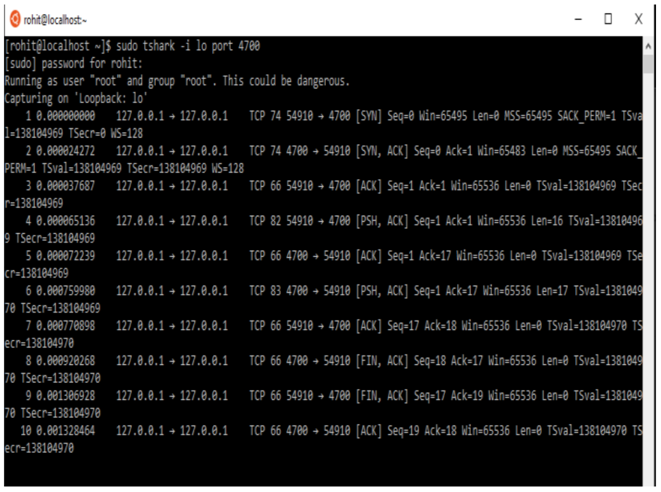
We use Xarray to store the hash values of the page. Xarray being performance-centric will be useful while searching hashes for large number of entries. Initially, the entries for the pages are created by supplying option "0" followed by the number of pages. Whenever the page is modified, the new hash value is calculated, and the updated value is stored in the kernel using the system call. For this, the system call is executed with option "2" followed by the page value and hash value. When the other client is demanding for a page value, the hash value stored is accessed by running the system call with option "1", followed by the page value. This page value will now be encrypted with the receiver's public key using AES and then sent along with the data. The receiver decrypts the hash value using its private key and verifies the integrity of the data received. By this way, any man-in the middle attack can be thwarted even though the data is sent using an open socket with security vulnerability. If the distributed memory application exits, the system call store_hash() needs to be called with option "4" in order to free-up all the entries that were created on the Xarray.

Data Integrity Verification –

To ensure the data is not corrupted by the adversary or network errors, we have introduced verification across ends of the shared nodes. Initially we designed the nodes using verification using only hash value. This provides us a good defence against network errors. When the value of the shared data gets corrupted during transfer or at origin, the hash value will be calculated for the corrupted data and we can identify network error or source data corruption. Since hashing the data only provides data with memory corruption, adversaries on the network can modify the data on the fly. If plain hash value is used, due to universality of the SHA256 algorithm, it is possible the adversary can modify and update the hash value. To avoid this, we are using the AES based encryption scheme for the hash value. When the application sends data from one system to another, we encrypt the hash value using a randomly generated and prepared AES key that is shared at the beginning of the handshake in between server and client application. When the recipient of the message gets the data and hash, we recalculate the hash and decrypt the message to compare with the cipher text sent by the other node. This ensures that the application will be resilient to network attacks. We have used libtomcrypt for our hashing and encryption

4. Results

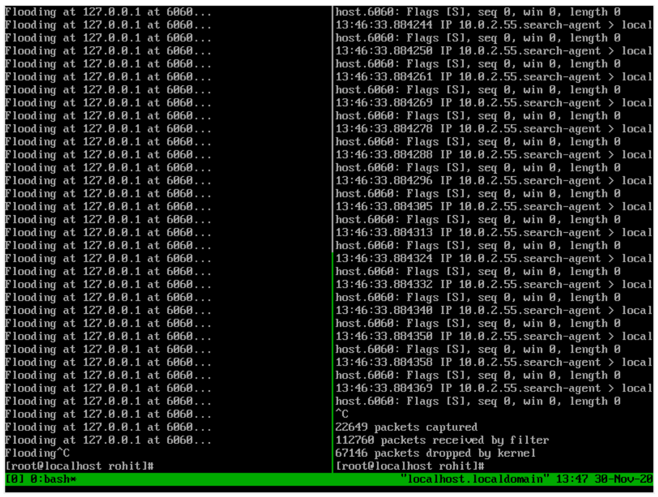
1) We have tested the RSA library and we are able to encrypt and decrypt the data successfully.
2) We have designed our state machines for maximum safety and correctness. We are still improving design to avoid leakage of sensitive information.
3) For implementing an on the fly network attack, we use the command line version of Wireshark known as Tshark to monitor the IP addresses and port numbers of the processes which are involved in the TCP/IP connection. Along with this, Tshark also provides us the information regarding the sequence numbers and acknowledgement numbers which we will use for tampering of data. The information obtained by monitoring the TCP protocol between the client and the server processes is depicted in Figure 4.



```
[rohit@localhost ~]$ sudo tshark -i lo port 4700
[sudo] password for rohit:
Running as user "root" and group "root". This could be dangerous.
Capturing on 'Loopback: lo'
    1 0.000000000    127.0.0.1 → 127.0.0.1    TCP 74 54910 → 4700 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM=1 TSva
l=138104969 TSecr=0 WS=128
    2 0.000024272    127.0.0.1 → 127.0.0.1    TCP 74 4700 → 54910 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_
PERM=1 TSval=138104969 TSecr=138104969 WS=128
    3 0.000037687    127.0.0.1 → 127.0.0.1    TCP 66 54910 → 4700 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=138104969 TSec
r=138104969
    4 0.000065136    127.0.0.1 → 127.0.0.1    TCP 82 54910 → 4700 [PSH, ACK] Seq=1 Ack=1 Win=65536 Len=16 TSval=13810496
9 TSecr=138104969
    5 0.000072239    127.0.0.1 → 127.0.0.1    TCP 66 4700 → 54910 [ACK] Seq=1 Ack=17 Win=65536 Len=0 TSval=138104969 TSe
cr=138104969
    6 0.000759980    127.0.0.1 → 127.0.0.1    TCP 83 4700 → 54910 [PSH, ACK] Seq=1 Ack=17 Win=65536 Len=17 TSval=1381049
70 TSecr=138104969
    7 0.000770898    127.0.0.1 → 127.0.0.1    TCP 66 54910 → 4700 [ACK] Seq=17 Ack=18 Win=65536 Len=0 TSval=138104970 TS
ecr=138104970
    8 0.000920268    127.0.0.1 → 127.0.0.1    TCP 66 4700 → 54910 [FIN, ACK] Seq=18 Ack=17 Win=65536 Len=0 TSval=1381049
70 TSecr=138104970
    9 0.001306928    127.0.0.1 → 127.0.0.1    TCP 66 54910 → 4700 [FIN, ACK] Seq=17 Ack=19 Win=65536 Len=0 TSval=1381049
70 TSecr=138104970
   10 0.001328464    127.0.0.1 → 127.0.0.1    TCP 66 4700 → 54910 [ACK] Seq=19 Ack=18 Win=65536 Len=0 TSval=138104970 TS
ecr=138104970
```

Figure 4 Screenshot of TCP Protocol data

4) We have implemented a SYN flooding attack on the port which is binded to the server. The concept of SYN flooding is to spam the port of a particular server with packets with random sequence numbers so that the server becomes flooded with incoming packets that it is not able to allot resources for any other clients to transfer data over TCP socket programming protocol. The below figure represents the flooding of port 6060 by an attacker who does it by forging a random IP address. Attack needs to be run as root user or using sudo command.

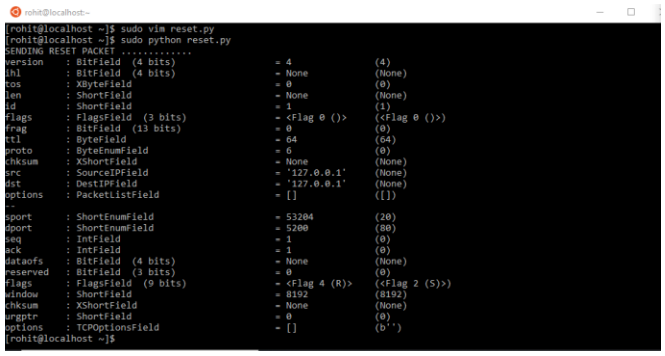


```
Flooding at 127.0.0.1 at 6060...  host.6060: Flags [S], seq 0, win 0, length 0
Flooding at 127.0.0.1 at 6060...  13:46:33.884244 IP 10.0.2.55.search-agent > local
Flooding at 127.0.0.1 at 6060...  host.6060: Flags [S], seq 0, win 0, length 0
Flooding at 127.0.0.1 at 6060...  13:46:33.884250 IP 10.0.2.55.search-agent > local
Flooding at 127.0.0.1 at 6060...  host.6060: Flags [S], seq 0, win 0, length 0
Flooding at 127.0.0.1 at 6060...  13:46:33.884261 IP 10.0.2.55.search-agent > local
Flooding at 127.0.0.1 at 6060...  host.6060: Flags [S], seq 0, win 0, length 0
Flooding at 127.0.0.1 at 6060...  13:46:33.884269 IP 10.0.2.55.search-agent > local
Flooding at 127.0.0.1 at 6060...  host.6060: Flags [S], seq 0, win 0, length 0
Flooding at 127.0.0.1 at 6060...  13:46:33.884278 IP 10.0.2.55.search-agent > local
Flooding at 127.0.0.1 at 6060...  host.6060: Flags [S], seq 0, win 0, length 0
Flooding at 127.0.0.1 at 6060...  13:46:33.884288 IP 10.0.2.55.search-agent > local
Flooding at 127.0.0.1 at 6060...  host.6060: Flags [S], seq 0, win 0, length 0
Flooding at 127.0.0.1 at 6060...  13:46:33.884296 IP 10.0.2.55.search-agent > local
Flooding at 127.0.0.1 at 6060...  host.6060: Flags [S], seq 0, win 0, length 0
Flooding at 127.0.0.1 at 6060...  13:46:33.884305 IP 10.0.2.55.search-agent > local
Flooding at 127.0.0.1 at 6060...  host.6060: Flags [S], seq 0, win 0, length 0
Flooding at 127.0.0.1 at 6060...  13:46:33.884313 IP 10.0.2.55.search-agent > local
Flooding at 127.0.0.1 at 6060...  host.6060: Flags [S], seq 0, win 0, length 0
Flooding at 127.0.0.1 at 6060...  13:46:33.884324 IP 10.0.2.55.search-agent > local
Flooding at 127.0.0.1 at 6060...  host.6060: Flags [S], seq 0, win 0, length 0
Flooding at 127.0.0.1 at 6060...  13:46:33.884332 IP 10.0.2.55.search-agent > local
Flooding at 127.0.0.1 at 6060...  host.6060: Flags [S], seq 0, win 0, length 0
Flooding at 127.0.0.1 at 6060...  13:46:33.884340 IP 10.0.2.55.search-agent > local
Flooding at 127.0.0.1 at 6060...  host.6060: Flags [S], seq 0, win 0, length 0
Flooding at 127.0.0.1 at 6060...  13:46:33.884350 IP 10.0.2.55.search-agent > local
Flooding at 127.0.0.1 at 6060...  host.6060: Flags [S], seq 0, win 0, length 0
Flooding at 127.0.0.1 at 6060...  13:46:33.884358 IP 10.0.2.55.search-agent > local
Flooding at 127.0.0.1 at 6060...  host.6060: Flags [S], seq 0, win 0, length 0
Flooding at 127.0.0.1 at 6060...  13:46:33.884369 IP 10.0.2.55.search-agent > local
Flooding at 127.0.0.1 at 6060...  host.6060: Flags [S], seq 0, win 0, length 0
Flooding at 127.0.0.1 at 6060...  ^C
Flooding at 127.0.0.1 at 6060...  22649 packets captured
Flooding at 127.0.0.1 at 6060...  112760 packets received by filter
Flooding^C                        67146 packets dropped by kernel
[root@localhost rohit]#           [root@localhost rohit]#
[0] 0:bash*                            "localhost.localdomain" 13:47 30-Nov-20
```

Figure 5 Screenshot of TCP SYN Flooding

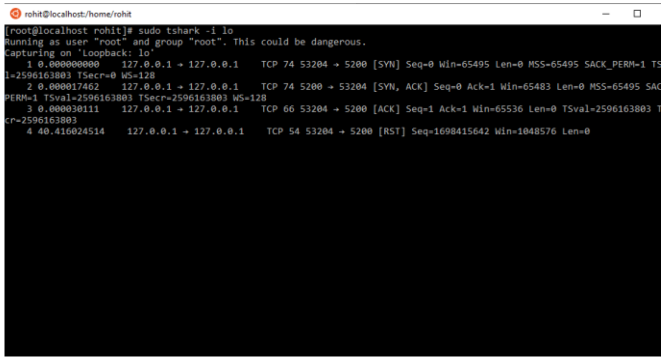
We have flooded the respective server port, but this does not stop the TCP communication between processes. This is because linux implements SYN cookies mechanism which does not allocate resources at all after the server has only received the SYN packet. It allocates resources only if the server has received the final ACK packet. Attackers can do the ACK flooding which causes the server to allocate resources. This is more harmful than the SYN flooding attack because resources allocated for a completed connection are more than for a half-open connection. The SYN cookies idea provides a solution to this problem. When the server receives a SYN packet, it calculates a keyed hash value from the TCP field information in the packet using a secret key that is only known to the server. This hash value H will be used as the initial sequence number placed in the server's SYN+ACK packet sent back to the client. The value H is known as SYN cookies. When the server receives the packet back it must have the ACK number as H+1, else it will drop the packet and not allocate resources.

5) The next attack what was executed was TCP reset attack to terminate an existing connection between a client and a server. This was done using the scapy library in python. The information needed for constructing the TCP packet is obtained from executing the Tshark command. The packet is given the sequence and acknowledgement numbers based on the next calculated packet number. To prevent this attack linux recognizes this as a spurious packet and assigns random sequence and acknowledgement numbers to the packet due to which the connection between server and client does not reset. This is shown in the figures below.



```
[rohit@localhost ~]$ sudo vim reset.py
[rohit@localhost ~]$ sudo python reset.py
SENDING RESET PACKET .............
version    : BitField  (4 bits)                = 4             (4)
ihl        : BitField  (4 bits)                = None          (None)
tos        : XByteField                        = 0             (0)
len        : ShortField                        = None          (None)
id         : ShortField                        = 1             (1)
flags      : FlagsField  (3 bits)              = <Flag 0 ()>   (<Flag 0 ()>)
frag       : BitField  (13 bits)               = 0             (0)
ttl        : ByteField                         = 64            (64)
proto      : ByteEnumField                     = 6             (0)
chksum     : XShortField                       = None          (None)
src        : SourceIPField                     = '127.0.0.1'   (None)
dst        : DestIPField                       = '127.0.0.1'   (None)
options    : PacketListField                   = []            ([])
--
sport      : ShortEnumField                    = 53204         (20)
dport      : ShortEnumField                    = 5200          (80)
seq        : IntField                          = 1             (0)
ack        : IntField                          = 1             (0)
dataofs    : BitField  (4 bits)                = None          (None)
reserved   : BitField  (3 bits)                = 0             (0)
flags      : FlagsField  (9 bits)              = <Flag 4 (R)>  (<Flag 2 (S)>)
window     : ShortField                        = 8192          (8192)
chksum     : XShortField                       = None          (None)
urgptr     : ShortField                        = 0             (0)
options    : TCPOptionsField                   = []            (b'')
[rohit@localhost ~]$
```

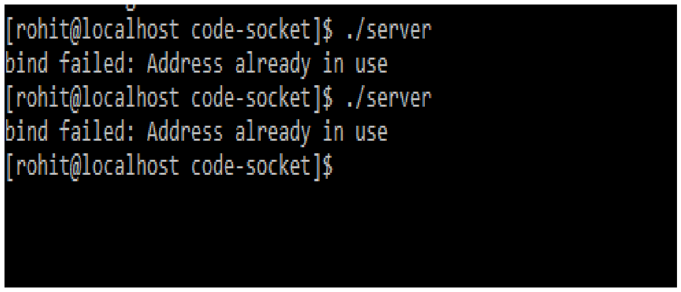
Figure 6 Python program spoofing TCP packet



```
[root@localhost rohit]# sudo tshark -i lo
Running as user "root" and group "root". This could be dangerous.
Capturing on 'Loopback: lo'
    1 0.000000000    127.0.0.1 → 127.0.0.1    TCP 74 53204 → 5200 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM=1 TS
l=2596163803 TSecr=0 WS=128
    2 0.000017462    127.0.0.1 → 127.0.0.1    TCP 74 5200 → 53204 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SAC
PERM=1 TSval=2596163803 TSecr=2596163803 WS=128
    3 0.000030111    127.0.0.1 → 127.0.0.1    TCP 66 53204 → 5200 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=2596163803 T
cr=2596163803
    4 40.416024514    127.0.0.1 → 127.0.0.1    TCP 54 53204 → 5200 [RST] Seq=1698415642 Win=1048576 Len=0
```

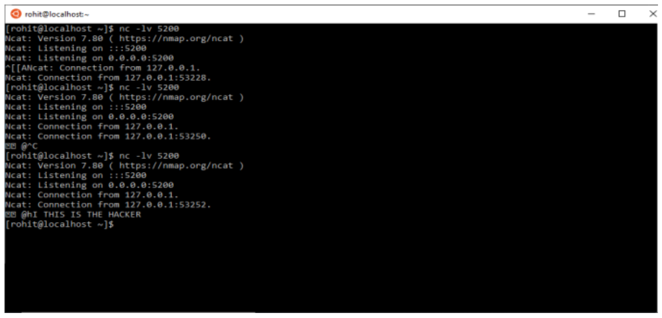
Figure 7 Tshark recognizing spurious reset data packet

6) The next attack we try to implement is the session hijacking attack. Once the attacker identifies the port binded with the server, he can use the netcat software to form a TCP connection with one of the processes to communicate with it. The server node will not be able to execute since that port is being used in the netcat command in the attacker terminal. The attacker can now send forged data to the client node which will treat it as data expected from the server node. The attacker can continuously keep executing the netcat command to gain control of the server port to communicate with the client. Once the 2 nodes finish communication, the netcat command is executed just before the server process is started to communicate with the client. The process is depicted in the figures below.



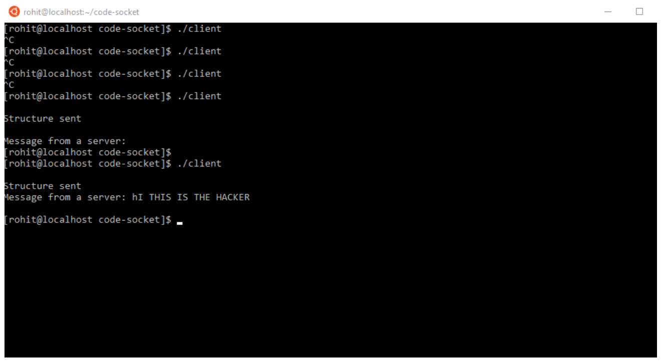
```
[rohit@localhost code-socket]$ ./server
bind failed: Address already in use
[rohit@localhost code-socket]$ ./server
bind failed: Address already in use
[rohit@localhost code-socket]$
```

Figure 8 Screenshot of Server program not failing to execute



```
[rohit@localhost ~]$ nc -lv 5200
Ncat: Version 7.80 ( https://nmap.org/ncat )
Ncat: Listening on :::5200
Ncat: Listening on 0.0.0.0:5200
^[[ANcat: Connection from 127.0.0.1.
Ncat: Connection from 127.0.0.1:53228.
[rohit@localhost ~]$ nc -lv 5200
Ncat: Version 7.80 ( https://nmap.org/ncat )
Ncat: Listening on :::5200
Ncat: Listening on 0.0.0.0:5200
Ncat: Connection from 127.0.0.1.
Ncat: Connection from 127.0.0.1:53250.
 @^C
[rohit@localhost ~]$ nc -lv 5200
Ncat: Version 7.80 ( https://nmap.org/ncat )
Ncat: Listening on :::5200
Ncat: Listening on 0.0.0.0:5200
Ncat: Connection from 127.0.0.1.
Ncat: Connection from 127.0.0.1:53252.
 @hI THIS IS THE HACKER
[rohit@localhost ~]$
```

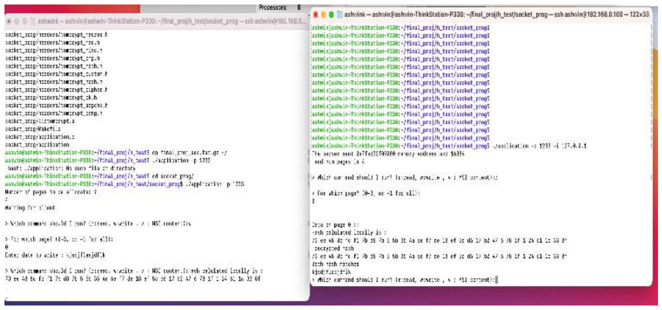
Figure 9 Screenshot of attacker sending and receiving client data



```
[rohit@localhost code-socket]$ ./client
^C
[rohit@localhost code-socket]$ ./client
^C
[rohit@localhost code-socket]$ ./client
^C
[rohit@localhost code-socket]$ ./client

Structure sent

Message from a server:
[rohit@localhost code-socket]$
[rohit@localhost code-socket]$ ./client

Structure sent
Message from a server: hI THIS IS THE HACKER

[rohit@localhost code-socket]$ _
```

Figure 10 Screenshot of client receiving malicious data sent By attacker

7) We have implemented the verification of data by sending its signature along with it which is created using the hash value stored in the kernel. The figure below depicts the verification of the data sent between the nodes using encryption and decryption of hash values. Hash values are encrypted to increase system security.



```
socket_prog/headers/tomcrypt_macros.h
socket_prog/headers/tomcrypt_mac.h
socket_prog/headers/tomcrypt_misc.h
socket_prog/headers/tomcrypt_prng.h
socket_prog/headers/tomcrypt_math.h
socket_prog/headers/tomcrypt_custom.h
socket_prog/headers/tomcrypt_hash.h
socket_prog/headers/tomcrypt_cipher.h
socket_prog/headers/tomcrypt_pk.h
socket_prog/headers/tomcrypt_argchk.h
socket_prog/headers/tomcrypt_cfg.h
socket_prog/libtomcrypt.a
socket_prog/Makefile
socket_prog/application.c
socket_prog/application
ashwin@ashwin-ThinkStation-P330:~/final_proj/h_test$ cp final_proj_aes.tar.gz ~/
ashwin@ashwin-ThinkStation-P330:~/final_proj/h_test$ ./application -p 1233
bash: ./application: No such file or directory
ashwin@ashwin-ThinkStation-P330:~/final_proj/h_test$ cd socket_prog/
ashwin@ashwin-ThinkStation-P330:~/final_proj/h_test/socket_prog$ ./application -p 1233
Number of pages to be allocated ?
4
Waiting for client

> Which command should I run? (r:read, w:write , v : MSI content):w

> For which page? (0-3, or -1 for all):
0
Enter data to write : kjsdjflsejdflk

> Which command should I run? (r:read, w:write , v : MSI content):hash calulated locally is :
73 ee 48 6c fc f1 7c d8 7b 6 5b 35 4e 6e f7 de 18 ef 6c d6 17 b2 47 6 78 1f 1 24 61 1c 33 8f
```

```
ashwin@ashwin-ThinkStation-P330:~/final_proj/h_test/socket_prog$
ashwin@ashwin-ThinkStation-P330:~/final_proj/h_test/socket_prog$
ashwin@ashwin-ThinkStation-P330:~/final_proj/h_test/socket_prog$
ashwin@ashwin-ThinkStation-P330:~/final_proj/h_test/socket_prog$
ashwin@ashwin-ThinkStation-P330:~/final_proj/h_test/socket_prog$
ashwin@ashwin-ThinkStation-P330:~/final_proj/h_test/socket_prog$
ashwin@ashwin-ThinkStation-P330:~/final_proj/h_test/socket_prog$
ashwin@ashwin-ThinkStation-P330:~/final_proj/h_test/socket_prog$
ashwin@ashwin-ThinkStation-P330:~/final_proj/h_test/socket_prog$
ashwin@ashwin-ThinkStation-P330:~/final_proj/h_test/socket_prog$
ashwin@ashwin-ThinkStation-P330:~/final_proj/h_test/socket_prog$
ashwin@ashwin-ThinkStation-P330:~/final_proj/h_test/socket_prog$
ashwin@ashwin-ThinkStation-P330:~/final_proj/h_test/socket_prog$
ashwin@ashwin-ThinkStation-P330:~/final_proj/h_test/socket_prog$
ashwin@ashwin-ThinkStation-P330:~/final_proj/h_test/socket_prog$
ashwin@ashwin-ThinkStation-P330:~/final_proj/h_test/socket_prog$ ./application -s 1233 -i 127.0.0.1
The server sent 0x7fa03279?000 memory address and 16384
 and num pages is 4

> Which command should I run? (r:read, w:write , v : MSI content):r

> For which page? (0-3, or -1 for all):
0


Data on page 0 ::
hash calulated locally is :
73 ee 48 dc fc f1 7b d8 7b 1 6b 3b 4a ce f7 de 18 ef 3c d6 17 b2 47 6 78 1f 1 24 c1 1c 33 8f
 decrypted hash
73 ee 48 dc fc f1 7b d8 7b 1 6b 3b 4a ce f7 de 18 ef 3c d6 17 b2 47 6 78 1f 1 24 c1 1c 33 8f
Both hash matches
kjsdjflasjdflk
> Which command should I run? (r:read, w:write , v : MSI content):
```

Figure 11 Screenshot of data verification at client and server nodes

5 Conclusions

In this project we manage to improve system security by verifying integrity of the data transferred between 2 instances of an application who form a shared memory region using the MSI protocol.

6 Future Work

A system with TEE (Trusted Execution Environment) enabled to isolate our cryptographic calculations in a separate core can be purchased, which can bring not just integrity in the system, but also provide provable security from memory-based attacks. The TEE also provides option bytes to providing protection of memory regions. These bits are to be programmed using JTAG, but the hope is dynamic fencing of memory using hardware can soon become a reality.

References:

[1] Edwin Franco Myloth Josephlal, Sridhar Adepu, 2019 IEEE 19th International Symposium on High Assurance Systems Engineering (HASE) Hangzhou, China 2019 Jan. 3 - 2019 Jan. 5

[2] M., Schmidt, K., and Zweck, H., "Hardware/Software Co-Design of an Automotive Embedded Firewall," SAE Technical Paper 2017-01-1659, 2017

[3] I. Echizen, S. Singh, T. Yamada, K. Tanimoto, S. Tezuka and B. Huet, "Integrity Verification System For Video Content By Using Digital Watermarking," 2006 International Conference on Service Systems and Ser-vice Management, Troyes, 2006, pp. 1619-1624, doi: 10.1109/ICSSSM.2006.320788.

[4] Sung-Ming Yen and Chi-Sung Laih. Improved Digital Signature Suitable for Batch Verification. In IEEE Transactions on Computers

[5] Yuval Yarom and Katrina Falkner. Proceedings of the 23rd USENIX Security Symposium ISBN 978-1-931971-15-7

[6] Andrei Kolichtchak, Nizhny Novgorod (RU) no. US20030014667A1

[7] R. Patil and M. P. Tahiliani, "Detecting packet modification attack by misbehaving router," 2014 First International Conference on Networks & Soft Computing (ICNSC2014), Guntur, 2014, pp. 113-118, doi: 10.1109/CNSC.2014.6906649.

[8] https://github.com/andrewkiluk/RSA-Library

[9] https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/

[10] R. Patil and M. P. Tahiliani, "Detecting packet modification attack by misbehaving router," 2014 First International Conference on Networks & Soft Computing (ICNSC2014), Guntur, 2014, pp. 113-118, doi: 10.1109/CNSC.2014.6906649.

[11] B. Mahbooba and M. Schukat, "Digital certificate-based port knocking for connected embedded systems," 2017 28th Irish Signals and Systems Conference (ISSC), Killarney, 2017, pp. 1-5, doi: 10.1109/ISSC.2017.798364