2. Attacks and Security in Mobile Ad Hoc Networks

Ad Hoc Networks are by nature very open to anyone. Their biggest advantage is also one of their biggest disadvantages; basically anyone with the proper hardware and knowledge of the network topology and protocols can connect to the network. This allows potential attackers to infiltrate the network and carry out attacks on its participants with the purpose of stealing or altering information.

2.1. Essentials and Vulnerabilities of Ad hoc network

**Availability:** It is the most basic requirement of any network. If the networks connection ports are unreachable, or the data routing and forwarding mechanisms are out of order, the network would cease to exist.

**Confidentiality:** This describes the need to protect the data roaming from being understood by unauthorized parties. Confidentiality can be achieved by encrypting essential information so that, only the communicating nodes can analyse and understand it.

**Integrity:** Data can be altered both intentionally and accidentally through hardware glitches or by interference in case of wireless connections. Hence integrity of communication data is required to ensure that the information passed on between nodes has not been altered in any way.

**Non-repudiation:** Non-repudiation means that the messages can be traced back to their senders, without the sender being able to deny having sent it. This makes it possible to detect intrusions and fake messages. Many routing and authentication algorithm implemented in ad hoc networks rely on trust-based concepts; the fact that
a message can be attributed to a specific node helps making these algorithm more secure.

2.2. Classifications of attacks

There are a variety of attacks on networks and can be grouped based on their characteristics.

**External and Internal Attacks:**
- *External attacks* are committed by parties that turnout legally part of the network.
- *Internal attacks* are sourced from inside a particular network. A compromised node (known as malicious node, whose actions compromise the security of the whole ad hoc network) which is accessible to all other nodes within its range poses a high threat to the functional efficiency of the whole network.

**Passive and Active Attacks:**
- *Passive attacks* do not involved any disruption of the service, they are merely intended to steal information and to eves drop on the communication within the network.
- *Active attacks* can actively alter the data, with the intent of overloading the network, obstructing the operation or to cut off certain nodes from their neighbours so that they cannot use the network services effectively anymore.

**Attacks on OSI Layers:**
- *Physical Layer Attacks*: Attacks might target the physical layer of a network, for example by jamming the transmissions of wireless antennas or phones, or by destroying the hardware of a certain node.
• **Network Layer Attacks:** An attacker could also exploit the protocols of the network layer. Intimate knowledge of the routing mechanisms involved can present security risks which are hard to defend against.

• **Application Layer Attacks:** Someone with bad intentions to abuse the loopholes of the application layer. In this case of an information network for example he could inject false or fake information, thus undermining the integrity of the application.

### 2.3. The features of MANET

All signals go through bandwidth-constrained wireless links in a MANET, which makes it more prone to physical security threats than fixed landline networks.

Mobile nodes are roaming independently and are able to move in any direction. Decentralized decision making in the MANET relies on the cooperative participation of all nodes. Nodes may rely on batteries or other exhaustible means for their energy.

**Difficult to provide because**

- Collaborative nature
- Less-robust and shared medium
- Requires solution for internal adversaries
- Transmission range is usually smaller than network span
- Need for *multi-hop* routing
- All nodes can potentially participate in the routing protocol

**Security concerns**

- Must define adversarial model
- Effect on network operation
- Passive attacks.
- Active attacks.
• Attackers are authorized to participate in the network operation
• Outside attacks.
• Inside attacks.

Outside Attacks
• Attackers do not possess credentials.
• Include: packet injection, packet modification, impersonation.
• In general preventable using standard cryptographic mechanisms that ensure authentication and data integrity

Inside (Byzantine) Attacks
• Byzantine behavior.
• Arbitrary action by an authenticated node resulting in disruption of the routing service.
• All nodes participate in routing.
• Authentication and data integrity mechanisms do not provide any guarantees.
• Different than the “selfish node” problem

Attacks against routing
• Black Hole Attack
• Flood Rushing Attack
• Wormhole Attack
• Overlay Network Attack
• Adversaries can act individually or can collude
2.4. Attack Types and their Countermeasures

Impersonation:

- **Definition**
  Impersonation attacks are also called spoofing attacks. The attacker assumes the identity of another node in the network, thus receiving messages directed to the nodes it fakes.

- **Countermeasures**
  - Equip nodes with GPS and calculate whether two nodes could really have a link.
  - Another solution is to include the 2-hop neighbors in the Hello message. This gives every node a 3-hop topology of the network, less expensive than special hardware, but is defeated by spoofing above 3-hops.
  - Using good authentication algorithms, strong data encryption and secure routing protocols.

Sinkhole attacks:

- **Definition**
  By carrying out a sinkhole attack, a compromised node tries to attract the data to it from all neighboring nodes. Since this would give access to all data to this node. The sinkhole attack is the basis for many other attacks like eavesdropping or data alteration. Sinkhole attacks make use of the loopholes in routing algorithms of ad hoc networks and present themselves to adjacent nodes as the most attractive partner in a multihop route.

- **Countermeasures**
  Probabilistic protocols measure the trustworthiness of a message based on the probability of the packet arriving from a certain source, which can help detecting sinkholes within the network (if many packets arrive from a rather improbable source).
Wormholes:

- **Definition**
  In wormhole attack, a malicious node uses a path outside the network to route message to another compromised node at some other location in the network.

- **Countermeasures**
  The idea of outfitting each packet with timestamps and location stamps in order to detect wormhole intrusions in a system.
  Avoiding any race conditions, making the attack close to pointless.
  Packet leashes, temporal and geographical.
  These prevent a packet from moving too far too fast.

**Sleep deprivation torture:**

- **Definition**
  The idea behind this attack is to request the services a certain node offers, over and over again, so it cannot go into an idle or power preserving state, this depriving it of its sleep (hence the name).

- **Countermeasures**
  Measures to prevent such attacks are hard to take, but the effects can be minimized by prioritizing between the functions of the target node, so that constant requests of low-priority services do not block other high-priority requests.

**The Sybil Attack:**

- **Definition**
  Malicious nodes in a network may not only impersonate one node, they could assume the identity of several nodes. Thus undermine the redundancy of many routing protocols.

- **Countermeasures**
  Using unique symmetric keys, by which each node can verify its neighbors identity and limiting the number of neighbours, a node can achieve the partial isolation of
compromised nodes, as these nodes can only communicate with their verified neighbors.

**Rushing Attack:**

- **Definition**
  This type of attack is mostly directed against on-demand routing protocols based on the Dynamic Source Routing protocol. A malicious node will attempt to tamper with ROUTE REQUEST packets, modifying the node list, and hurrying this packet to the next node.

- **Countermeasures**
  Rushing attacks can be detected by evaluating the Route Discovery.

**Denial-of-Service and Flooding:**

- **Definition**
  Malicious nodes can attempt to impersonate one or more nodes and control all data paths to a certain destination, thereby reducing its availability.
  Exhaust network resources, overall bandwidth, and individual nodes resources of computational and battery power.
  In AODV, an attacking node sends out a large number of RREQs for a route to a non-existent node.

- **Countermeasures**
  By calculating and blocking the rate of neighbours RREQs, when it exceeds the threshold value.
  By blocking valid node if it is spoofing real nodes, but cannot stop flooding below threshold.
  By using statistical analysis to detect varying rates of flooding.
Pollution Attacks:

- **Definition**
  This is an attack where attackers inject polluted coded packets into the network.
  Attacker joins an ongoing video channel.
  Attacker advertises it has a large numbers of chunks.
  When neighbours request chunks, attackers send bogus chunks.
  Receivers play back bogus chunks.
  Each receiver may further forward the polluted chunks.

- **Countermeasures**
  Using Blacklisting procedure.
  By Traffic Encryption.
  By Chunk Signing which is done by
  Using PKI (Public Key Infrastructure).
  By using the public-private key pair of the video source.
  Sources use the private key to sign the chunks.
  Receiver uses the public key of the source to verify integrity of chunk.

**2.5. Security Requirement**

Lack of physical security makes devices susceptible to theft.
All nodes participate in routing, must rely on untrusted nodes.
Lack of security leads to degradation of service because medium is shared.

MANET is particularly vulnerable due to its fundamental characteristics, such as open medium, dynamic topology, distributed cooperation, and constrained capability. Each mobile node operates not only as a host but also as a router. Routing plays an important role in the security of the entire network. MANET can be established extremely flexibly without any fixed base station in battlefields, military applications, and other emergency and disaster situation.
2.6. Simulator Used

NS-2 is a discrete event simulator written in C++, with an OTcl interpreter shell as the user interface that allows the input model files (Tcl scripts) to be executed. Most network elements in NS-2 are developed as classes, in object-oriented fashion. The simulator supports a class hierarchy in C++, and a very similar class hierarchy in OTcl. The root of this class hierarchy is the TclObject in OTcl. Users create new simulator objects through the OTcl interpreter, and then these objects are mirrored by corresponding objects in the class hierarchy in C++. NS2 provides substantial support for simulation of TCP, routing algorithms, queuing algorithms, and multicast protocols over wired and wireless (local and satellite) networks, etc. It is freely distributed, and all source code is available.

User’s Perspectives of NS

From the user’s perspective, NS-2 is an OTcl interpreter that takes an OTcl script as input and produces a trace file as output. NS uses two languages. Detailed simulations of protocols require a systems programming language which can efficiently manipulate bytes, packet headers, and implement algorithms that run over large data sets. For these tasks run-time speed is important and turn-around time (run simulation, find bug, fix bug, recompile, re-run) is less important.
A large part of the network research involves slightly varying parameters or configurations and quickly exploring a number of scenarios. Hence, iteration time (change the model and re-run) is more important. As configuration runs once (at the beginning of the simulation) run time becomes less important.

NS meets both the above requirements with two languages, C++ and OTcl. C++ is fast to run but slower to change, making it suitable for detailed protocol implementation. OTcl runs much slower but can be changed very quickly (and interactively), making it ideal for simulation configuration, ns (via tcl) provides glue to make objects and variables appear in both languages.

OTcl is used for the following purposes:

- For configuration, setup, and on-time stuff.
- Manipulating existing C++ objects.

C++ is used for the following purposes:

- Processing each of a flow.
- To change the behaviour of an existing C++ class.

**TCL Programs:** TCL codes are written to setup the wireless simulation components used by the Mobile Nodes:

<table>
<thead>
<tr>
<th>Network Component Type</th>
<th>Type of Antenna</th>
<th>Radio Propagation Model</th>
<th>Adhoc Routing Protocol</th>
<th>Traffic Models</th>
<th>Node Movement Models</th>
</tr>
</thead>
</table>
| **Network Animator (NAM):**| NAM is a Tcl/Tk based animation tool for viewing network simulation traces and real world packet trace data. The design theory behind nam was to create an animator that is able to read large animation data sets and be...
extensible enough so that it could be used indifferent network visualisation situations. Usually, the trace file is generated by ns. During an ns simulation, user can produce topology configurations, layout information, and packet traces using tracing events in ns. However any application can generate a nam. Upon start up, nam will read the trace file, create topology, pop up a window, do layout if necessary, and then pause at time 0. Through its user interface, nam provides control over many aspects of animation.

**Nam Trace:** Nam is a Tcl/Tk based animation tool that is used to visualise the ns simulation and real world packet trace data. The first step to use nam is to produce a nam trace file. The nam trace file should contain topology information like nodes, links, queues, node connectivity etc. as well as packet trace information.

**Creating Topologies in NS2**

![Figure 2.2: A Network Topology](image)

**Animation Objects:** NAM does animation using the following building blocks.

**Node:** Node represents a source, host, or router. Node can have three shapes, circle square and hexagon, but once created, it cannot change its shape. Node can change colour during animation and can be labelled.

**Link:** Links are created between nodes to form a network topology. Links can be simplex, duplex, wireless, or satellite. Links can be labelled and can change colour during animation.
**Queue:** Queues need to be constructed between two nodes. Queues are visualised as stacked packets.

**Packet:** Packets are visualised as a block with an arrow. Direction of the arrows shows the flow direction of the packets. Queued packets are shown as small squares. A packet may be dropped from a queue or a link. Dropped packets are shown as falling rotating squares.

**Agent:** Agents are used to separate protocol states from nodes. Agents are always associated with nodes. An agent has a name, which are unique identifiers of the agent. It is shown as a square with its name inside, and is drawn next to its associated node.

**Perl Programs:** PERL stands for ‘Practical Extraction and Report Language’. Perl allows easy filtering and processing of ASCII data files in UNIX. This Language was created by Larry Wall with the main idea of simplifying the task of system administration. Perl has evolved a lot and nowadays is a general purpose language and one of the most used tools for web and internet, data managing. Perl is an interpreted language who has many uses, but is mainly addressed to the search extraction and report. Some advantages of Perl are:

- Easily implementation of small programs to be used filters, for extracting information from text files.
- It can be used in many OSs without changing code.
- Maintaining and debugging of Perl Scripts are simpler than programs in other specific language.
- Perl is very popular, so there exists a lot of gnu scripts on the web.

**GNU Plots:** GNU plot is widely available free software both for unix as well as
windows operating systems. GNU plot has help command that can be used to learn details of its operation. The simplest way to use GNU plot is to type “plot<fn>”, where the file (whose name is fn) has two columns representing x and y values of points. Points can be joined by a line of different styles.

**Writing Basic Program in NS**

NS is an Object-oriented, discrete event-driven network simulator written in C++ and OTcl.

![Figure 2.3: Basic NS Structure](image)

### Simple Program In Interactive Mode:

bash-shell$ ns

% set ns [new Simulator]

% $ns at 1 "puts \"Welcome to NS!\""

1

% $ns at 1.5 “exit”

2

% $ns run

Welcome to NS!

bash-shell$
Simple Program In Batch Mode:

```tcl
set ns [new Simulator]
$ns at 1 "puts \"Welcome to NS!\""
$ns at 1.5 "exit"
$ns run
bash-shell$ ns simple.tcl
Welcome to NS!
bash-shell$
```

Basic Tc1 Program: example.tcl

```tcl
# Writing a procedure called "test"
proc test () {
    set a 43
    set b 27
    set c [expr $a + $b]
    set d [expr [expr $a - $b] * $c]
    for {set k 0} {$k < 10} {incr k} {
        if {$k < 5} {
            puts "$k < 5, pow = [expr pow($d, $k)"
        } else {
            puts "$k >= 5, mod = [expr $d % $k]"
        }
    }
}

# Calling the "test" procedure created above
test
```

Basic OTc1 Program:

```tcl
Class Father

Father instproc Hello {} {
    $self instvar age_
    puts "$age_ years old father says:
How are you?"
}

Class Son superclass Father

Son instproc Hello {} {
    $self instvar age_
```
puts "$age_ years old son says:
I am fine Dad"
}

set father [new Father]
$father set age_ 55
set son [new Son]
$son set age_ 20

$father Hello
$son Hello

55 years old father says:
How are you?
20 years old son says:
I am fine Dad

**NS-2 Generic Script Structure:**

*Create Simulator object*

Create event scheduler

set ns [new Simulator]

```plaintext

<table>
<thead>
<tr>
<th>event</th>
<th>time</th>
<th>from node</th>
<th>to node</th>
<th>pkt type</th>
<th>pkt size</th>
<th>flags</th>
<th>rid</th>
<th>src addr</th>
<th>dst addr</th>
<th>ssq</th>
<th>pkt id</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>1.3555</td>
<td>3 2</td>
<td>ack</td>
<td>40</td>
<td>1 3 0 0 1 1 2 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>1.3555</td>
<td>2 0</td>
<td>ack</td>
<td>40</td>
<td>1 3 0 0 1 1 2 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>1.3555</td>
<td>2 0</td>
<td>ack</td>
<td>40</td>
<td>1 3 0 0 1 1 2 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r</td>
<td>1.3557</td>
<td>2 3</td>
<td>tcp</td>
<td>1000</td>
<td>1 0 0 3 2 0 3 9 199</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>1.3557</td>
<td>2 3</td>
<td>tcp</td>
<td>1000</td>
<td>1 0 0 3 2 0 3 9 199</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>1.3557</td>
<td>2 3</td>
<td>tcp</td>
<td>1000</td>
<td>1 0 0 3 2 0 3 9 199</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>1.3556</td>
<td>1 2</td>
<td>obr</td>
<td>1000</td>
<td>2 1 0 3 1 1 5 7 2 0 7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>1.356</td>
<td>1 2</td>
<td>obr</td>
<td>1000</td>
<td>2 1 0 3 1 1 5 7 2 0 7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Figure 2.4:** A Typical Trace File
Tracing

- Inserted immediately after scheduler
- Trace packets on all links

```plaintext
set nf [open out.nam w]
$ns trace-all $nf
$ns namtrace-all $nf
```

Create topology

Two nodes, One link

Nodes

```plaintext
set n0 [$ns node]
set n1 [$ns node]
```

Links and queuing

```plaintext
$ns duplex-link $n0 $n1 1Mb 10ms RED
$ns duplex-link $n0 $n1 <bandwidth> <delay> <queue_type>
<queue_type>: DropTail, RED, etc
```

Creating a larger topology

```plaintext
for {set i 0} {$i < 7} {incr i} {
set n($i) [$ns node]
}
for {set i 0} {$i < 7} {incr i} {
$ns duplex-link $n($i) $n([expr ($i+1)%7]) 1Mb 10ms RED
}
```

![Diagram of two nodes connected](image)

**Figure 2.5:** Typical links amongst the nodes
Network Dynamics

Setup packet loss, link dynamics

Link failures

Hooks in routing module to reflect routing changes

\$ns\ rtmodel\-at <time> up|down $n0 $n1

For example:

\$ns\ rtmodel\-at 1.0 down $n0 $n1
\$ns\ rtmodel\-at 2.0 up $n0 $n1

Create routing agents

Creating UDP connection

set udp [new Agent/UDP]
set null [new Agent/Null]
\$ns\ attach\-agent $n0 $udp
\$ns\ attach\-agent $n1 $null
\$ns\ connect $udp $null

▶ Creating Traffic (On Top of UDP)
▶ CBR

**Figure 2.6:** Creating UDP connections

set cbr [new Application/Traffic/CBR]
$cbr$ set packetSize_ 500
$cbr$ set interval_ 0.005
$cbr$ attach-agent $udp

Creating TCP connection

set tcp [new Agent/TCP]
set tcpsink [new Agent/TCPSink]
$ns$ attach-agent $n0$ tcp
$ns$ attach-agent $n1$ tcpsink
$ns$ connect tcp tcpsink

Create application and/or traffic sources

Creating Traffic

(On Top of TCP)

FTP

set ftp [new Application/FTP]
$ftp$ attach-agent tcp

Telnet

set telnet [new Application/Telnet]
		$telnet$ attach-agent tcp

Figure 2.7: Creating TCP connections
**Post-processing procedures (i.e. nam)**

Add a 'finish' procedure that closes the trace file and starts nam.

```tcl
proc finish {} {
    global ns nf
    $ns flush-trace
    close $nf
    exec nam out.nam &
    exit 0
}
```

**Start simulation**

Schedule Events

```tcl
$ns at <time> <event>

- <event>: any legitimate ns/tcl commands

$ns at 0.5 "$cbr start"
$ns at 4.5 "$cbr stop"
```

Call ‘finish’

```tcl
$ns at 5.0 "finish"
```

Run the simulation

```tcl
$ns run
```

**Visualization Tools**

nam-1 (Network AniMator Version 1) is a Packet-level animation and is well

- supported by ns
- xgraph

Simulation results

**nam Interface: Nodes**

Color

```tcl
$node color red
```
- Shape (can’t be changed after sim starts)
  $\text{node shape box (circle, box, hexagon)}$

- Label (single string)
  $\text{ns at 1.1 \"n0 label \"web cache 0\"\"}$

**Figure 2.8: Network AniMator**

**nam Interface: Links**

**Color**

$\text{ns duplex-link-op n0 n1 color "green"}$

**Label**

$\text{ns duplex-link-op n0 n1 label "backbone"}$

**nam Interface: Topology Layout**

“Manual” layout: specify everything

$\text{ns duplex-link-op n(0) n(1) orient right}$

$\text{ns duplex-link-op n(1) n(2) orient right}$

$\text{ns duplex-link-op n(2) n(3) orient right}$
Performance Study of An Ad hoc Network under Malicious Node Attack

$\textit{ns duplex-link-op n(3) n(4) orient 60deg}$

If anything missing $\rightarrow$ automatic layout

**Simulation Example**

![Simulation Diagram]

Simulation Example

Extending ns

**OTcl and C++: The Duality**

![OTcl and C++ Diagram]

Figure 2.9: Simulation Example

Figure 2.10: OTcl and C++
**TclObject: Hierarchy and Shadowing**

![Diagram of TclObject hierarchy and shadowing](image)

**Figure 2.11: Tcl Object**

**Figure 2.12: Typical NS-allinone**

**Extending ns in OTcl**

- If compiling is not required
- We source changes in simulation scripts
- Modifying existing code
- Recompiling
- Adding new files
- Changing Makefile (NS_TCL_LIB),
• Updating tcl/lib/ns-lib.tcl
• Recompiling

Adding Changes into NS

```
class MyAgent : public Agent {
public:
    MyAgent();
protected:
    int command(int argc, const char*const* argv);
private:
    int my_var1;
    double my_var2;
    void MyPrivFunc(void);
};
```

```
static class MyAgentClass : public TclClass {
public:
    MyAgentClass() : TclClass("Agent/MyAgent0tcl") {}
    TclObject* create(int, const char*const*) {
        return (new MyAgent());
    }
} class_my_agent;
```

Extending NS in C++
• Modifying code
• Make depend
• Recompile
• Adding code in new files
• Change Makefile
• Make depend ` Recompile

**OTcl Linkage**

• Creating a new agent “MyAgent”
• Dummy agent
• Derived from the “Agent” class

**Step 1: Export C++ class to OTcl**

**Step 2: Export C++ class variables to OTcl**

```cpp
MyAgent::MyAgent() : Agent(PT_UDP) {
    bind("my_var1_otcl", &my_var1);
    bind("my_var2_otcl", &my_var2);
}
```

Set the default value for the variables in the "ns-2/tcl/lib/ns-lib.tcl" file

**Step 3: Export C++ Object Control Commands to OTcl**

```cpp
int MyAgent::command(int argc, const char* const* argv) {
    if(argc == 2) {
        if(strcmp(argv[1], "call-my-priv-func") == 0) {
            MyPrivFunc();
            return(TCL_OK);
        }
    }
    return(Agent::command(argc, argv));
}
```

**Step 4: Execute an OTcl command from C++**

```cpp
void MyAgent::MyPrivFunc(void) {
    Tcl& tcl = Tcl::instance();
    tcl.eval("puts \"Message From MyPrivFunc\"\";
    tcl.evalf("puts \" my_var1 = %d\"\", my_var1);
    tcl.evalf("puts \" my_var2 = %f\"\", my_var2);
}
```
Step 5: Compile

- Save above code as “ex-linkage.cc”
- Open "Makefile", add "ex-linkage.o" at the end of object file list.
- Re-compile NS using the "make" command.

Step 6: Run and Test “MyAgent”

```tcl
ex-linkage.tcl

# Create MyAgent (This will give two warning messages that
# no default values exist for my_var1_otcl and my_var2_otcl)
set myagent [new Agent/MyAgentOtcl]

# Set configurable parameters of MyAgent
$myagent set my_var1_otcl 2
$myagent set my_var2_otcl 3.14

# Give a command to Myagent
$myagent call-my-priv-func
```

**result**

```
warning: no class variable Agent/MyAgentOtcl::my_var1_otcl
see tcl-object.tcl in tclcl for info about this warning.

warning: no class variable Agent/MyAgentOtcl::my_var2_otcl

Message From MyPrivFunc
my_var1 = 2
my_var2 = 3.140000
```