

TAG Based Data Exchange

Advanced Operative Systems and System Security Project

Ezio Emanuele Ditella

0286766

Università Di Roma Tor Vergata

I. INTRODUCTION

THE The TAG Based data exchange kernel module is intended to be an inter process communication system which can be seen as a variant of a Publish-Subscriber message oriented middleware. A full description of the project requirements can be found here. In this report we are going to discuss about the *choices* made during the Module's development. The most *critical parts* which we are going to **focus on** are the **design and implementation decisions** of:

- The Architecture
- The IPC_PRIVATE key management
- Creation and Remotion of a TAG Service
- Sending & Receiving mechanisms
- The Synchronization methodologies
- Driver's memory optimization technique
- Syscall Injection module

II. ARCHITECTURE

The architecture is composed by the following data structures:

- A *fixed* size table, **tag_table**, holding the pointers to each entries, aligned in such a way to avoid false cache sharing
- The entry of the *tag_table*, namely the **tag_service**, holding information such as the creator pid, permission, pointer to the tag_levels, ecc
- A double linked list held in the *tag_service*, the **tag_levels_list**, which maintains the informations (on the levels opened in the corresponding *tag_service*) in the data structure called **tag_level**
- A linked list, **receiving_threads**, held in the *tag_level*, which contains the metadata associated to the threads receiving on that level

The combination between fixed size data structures and dynamic ones has been made to obtain a good trade-off between ease of management and service malleability.

III. IPC_PRIVATE KEY MANAGEMENT

One of the most challenging decision of this project was to define *how* to store and handle the data of an IPC_PRIVATE tag **along with** the '*normal*' tags, in such a way to **prevent** a generic process from accessing a private tag **unless** it was an ancestor of the creator of that private tag.

One of the following three approaches could have been used:

- 1) **Ideal approach**: a private *tag_service* is kept in the address space of the creator (not in the kernel space) and the pointer to its address is kept in the *tag_table*. In this way only the son's of the creator of that tag would be able to access that data.
- 2) **Ancestor discovery**: if a thread is trying to access a private *tag_service*, somehow it is examined whether that thread is a son of the creator.
- 3) **Password based**: upon the creation of a private *tag_service* it will be returned **both** the tag descriptor and a password. In this way only the owners of that password will be able to access to that private tag.

It has been decided to follow the **password based** approach because it is a good trade off between:

- 1) **Memory accesses**: the number of memory accesses needed to handle a private tag will be the same memory accesses needed for a normal tag. The latter affirmation would have not state true for the *ancestor discovery* approach. In the *ideal approach* we would need to switch between kernel and user address space, and with PTI enabled system performances would have been affected.
- 2) **Latency**: between all the previous approaches, the password based require the lowest number of operations.
- 3) **Ease of implementation**: the *password based* is more difficult to implement and handle rather than the *ancestor discovery*, but way more easier than the *ideal approach*.

The *password based* approach can be summarized in the following steps:

- 1) Upon the creation of a private tag, the following 32-bit integer descriptor is returned:

0RRRRRRR	RRRRRRRR	RRRRRRRR	DDDDDDDD
----------	----------	----------	----------

(R: password's bit; D: descriptor's bit)

Note: in this example the maximum number of tag services would be $256 = 2^8$:

- The most significant $32-8=24$ bits will contain the password. But the most significant bit of the password will always be set to zero to avoid negative int return (to distinguish error from non error returns).
 - The least significant 8 bits will contain the descriptor: the real offset to be used to access the *tag_table*.
- 2) The random password will be placed in the corresponding *tag_service* data structure
 - 3) Upon accessing the private *tag_service*,
 - a) The password and the descriptor will be extracted.
 - b) There will be executed validity checks on the descriptor.
 - c) The access will be granted only if the password provided will match the memorized one on the tag service.
 - d) If the passwords will not match, an **anti-bruteforce** mechanism will be executed: the (eventual) malicious thread will be put to sleep for 3 seconds, and a warning message will appear on his console.

IV. TAG SERVICE CREATION & REMOTION

The creation and remotion of a new TAG Service is done by using the *tag_get()* and *tag_ctl()* systemcalls, respectively.

In order to increase the possible use cases that this Kernel Module could satisfy, it has been decided to implement the creation and remotion procedure in such a way that it is **not mandatory** to explicitly remove the tag after its creation.

With this feature:

- A tag service could be created by a thread and deleted by another one (if tag permissions allow it).
- A tag service could be created and 'forgotten', allowing more flexibility to unused tags.
- A **dynamic** and sophisticated IPC system could be set up.

However, the obvious **side effect** of this feature is that unremoved tags could potentially fill the whole *tag_table*. Consequently the TAG Service module would be unusable.

In order to avoid this issue, a new component in the system was introduced: the **TAG Cleaner**.

The **TAG Cleaner** is a Kernel Thread which periodically checks whether a tag has to be removed or not. In particular he does it using an integer array, say **A**, of as many entries as the *tag_table* ones. The generic entry of **A** can be interpreted as the TTL (Time To Live) for that tag service. This means that whenever a thread receives and/or send data in a particular level of a tag service, the corresponding entry in **A** has to be reset.

In particular the **TAG Cleaner** sleeps for **X** seconds and decreases the entries of the array **A** by **X** units. If one of these entries is less or equal than zero, the corresponding tag service will be deleted.

V. SENDING & RECEIVING MECHANISMS

Receiving

The receiving operation take place when calling the *tag_receive()* system call. The latter can be divided in **4 phases**:

1) *Input Validation*: In this phase it is verified if the calling thread is allowed to receive in the tag service provided (through the descriptor) according to the tag permissions and its state. The latter means that if the tag service has been 'acquired' by the cleaner and it is being deleted, the syscall will fail.

Furthermore it has been decided that a thread could specify **any** integer value for the level with the aim to let a receiver choose a **dynamic** number of levels.

2) *Level Lookup*: A receiving thread will have either to find the level on which he would like to receive, or create it. After that he has to put the information needed to be reachable by a sender: its PID, the address of the receiving buffer and its size. In this phase the synchronization has to take in place only between the receivers of the same tag level (more on that in the synchronization section).

3) *Receiving op*: In this phase the receiver will put himself in an **interruptible wait event queue**. To increase the ease of queue handling, it has been decided to maintain a global queue. Each receiver will then recognize if he has been awoken or not by checking some values edited by the sender in a *tag_level*.

Whenever a signal arrives or data is received or an **AWAKE_ALL** command is invoked, the thread will wake up and recognize the reason.

4) *Cleanup op*: Independently of the awakening reason, in this phase the receiver has to clean up the data structures allocated.

Sending

The sending operation is executed through the *tag_send()* syscall. After the user input validation phase, done in the same way of the receiving op, the sender thread has to copy the content of a buffer from its (user) address space to the receiver(s) (user) address space.

In order to do this, **firstly** the buffer to be sent is copied into the kernel address space, then, the sender **move** its address space to the receiver(s) one. This is made **by changing** its *vm_area_struct* (mm field of a thread TCB) and the virtual address of the page table (pgd field of mm struct) from the sender's one to the receiver's. After that the sender performs the actual copy and then he moves back to its address space.

VI. SYNCHRONIZATION METODOLOGIES

In this section we are going to explain **how** the system components coordinate between each others. We will refer to them by using the following notation:

- Tag Cleaner: *C*
- Deleting tag service operation with *tag_ctl(cmd=DELETE)*: *D*
- Creation of a New tag service with *tag_get(cmd=CREATE)*: *N*
- Receiver: *R*
- Sender: *S*

Since we are in kernel mode, we would like to be as fast as possible. Therefore it has been decided to use synchronization primitives that **do not** rely on wait queues (apart from the driver operations). The first thing to clarify is that **to avoid deadlock** it has been fixed an order of acquiring the synchronization primitives. The latters are:

- *L*: Spinlock of a given level in a given tag service
- *S*: Spinlock of a tag service
- *T*: Spinlock of the tag table
- *E*: Semaphore array (entries initialized to one)
- *M*: Mutex used by the driver

Lets now clarify how these primitives are used and by whom.

The entry E_i of *E* is accessed by Every component but *N*. In particular *C* and *D* will **try** (with a *try_down*) to decrease E_i . If the read value of E_i is zero, **no** other component is using that tag service, and it can be deleted. *C* and *D* will fail if, after decreasing E_i , its value is not 0. This means that whenever *R* or *S* wants to perform some operation on a tag service they have to increase the corresponding E_i , after making sure that has a value greather than 0 (still, with a *try_down*). As we can see **no thread** is put into some wait queue by using these semaphores.

The spinlock *T* is acquired by *N*, *C*, *D* in order to update the tag table upon deletion or creation of a new tag.

The spinlock *S* is held during the lookup phase of *R* and *S*. Once the level has been found, firstly is locked *L* then is unlocked *S*. This would let senders and receivers operating in different levels to have the lowest possible interference.

The TAG Driver uses the mutex *M* whenever he's called and he has to compose the message status to display to the user. If a read operations is blocked by the mutex, another thread has been called at the same time to print the tag service status. On mutex release the waiting threads **will reuse** the buffered message composed by the driver thread which was the first to acquire the mutex.

VII. TAG DRIVER

The read operation of the tag driver developed simply consist of acquiring information on the receiving threads (their PID and number) of each tag service level.

The most relevant operation that is performed to enhance system performance is an **auto adaptation** mechanism on memory allocation. In particular on module installation it is reserved a fixed number of pages (via *vmalloc* primitive), and every time that the tag driver read operation is called, it's evaluated the space used to store the status message. If after 10 subsequent call the space needed is greater (lower) than the initial reserved pages, the buffer is increased (decreased) in its size.

This mechanism has been implemented in order to allocate a relatively low number of non-contiguous pages on system start up and, if needed, to increase this number by adapting to system usage.

The current version of the module creates a node, called *TAG-service-stat*, that will use the driver operations. An example of the output upon a read on this node is shown in Figure 1

VIII. SYSCALLADDER

The TAG Based Data Exchange module rely on the *SyscallAdderV2* module to install the system calls. This module will use the following functions

- 1) *syscall_adder(...)*: will check if there is a free entry on the syscall table and if so, the syscall will be inserted. There will be inserted a MACRO in a file (located at */home/'user'/custom_syscall_macros.h*). An example this file contents is shown in Figure 2. This macro, when **included** in the user c file where the custom syscall is used, will make possible calling the new syscall like: *custom_syscall(...)*. An exemplificative image is shown in Figure 3.
- 2) *syscall_remover(...)*: This function simply delete a custom system call inserted previously

The **reason** for the implementation of the *SyscallAdderV2* module was to grant the greatest **transparency of use**. For more detail on how this module work and its implementation check this github repository.

IX. SYSTEM TESTS

To let the user **test** in an **interactive way** the module, a bash script was implemented, located at *TAGBased-DE/test/multi-console-test*. An example of what the script can do is shown in Figure 4.

The script simply let the user execute some compiled user C programs in different terminals than the multi-console-test. In this way it is possible to see more clearly the behaviour of the TAG Service module (and the return values of each system call).

For instance we can check the status of the TAG Service, by launching the *monitor* option, while using a custom number of senders and receivers (a view of this test is shown in Figure 5).

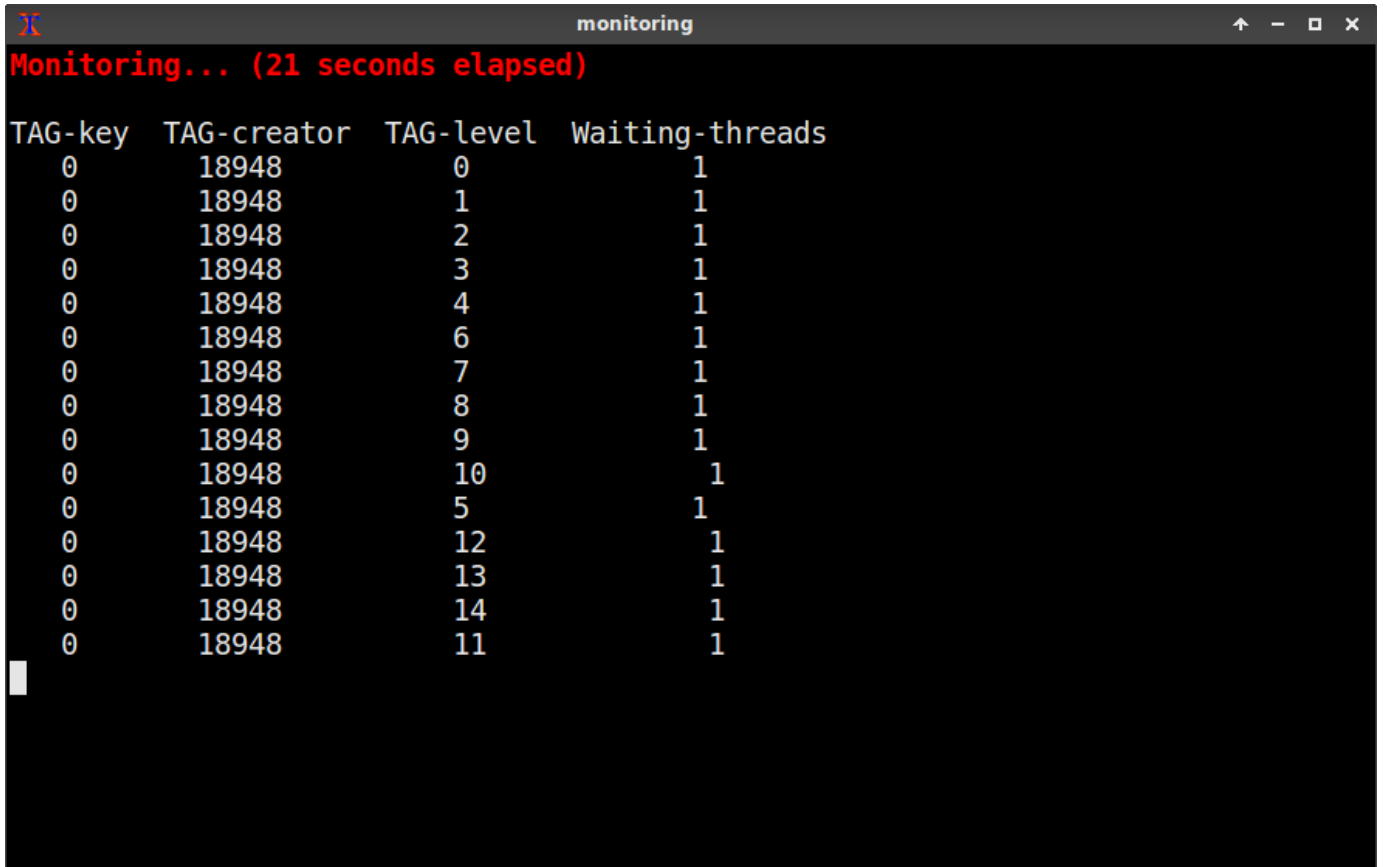
Attention: The current (released) version of the module has implemented the *tag_receive()* syscall in such a way that if after 5 seconds no data is being received, it returns. **In this way** the *Automatic Send-Receive Test* option of the multi console test can be used. The latter is a C program that spawn X receiver threads on the same tag service. Firstly the receivers are all on the same level and then all on different levels (this test can be done to check the TAG Service performance on the underlying system).

X. INSTALLATION

To install this Kernel Module you can simply run the following commands:

- `git clone https://github.com/x-Ultra/TAG-Based-DE`
- `cd TAG-Based-DE/`
- `./install`

SNAPSHOTS

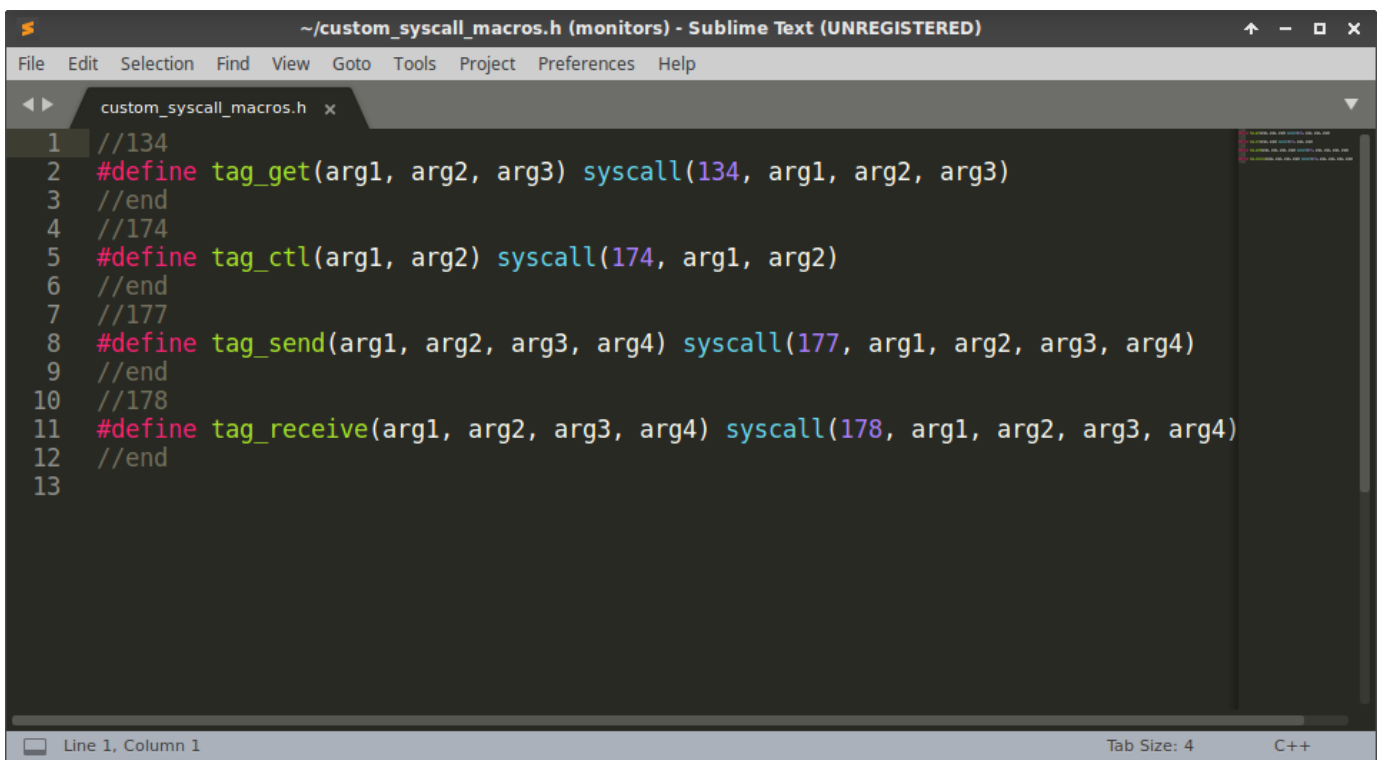


monitoring

Monitoring... (21 seconds elapsed)

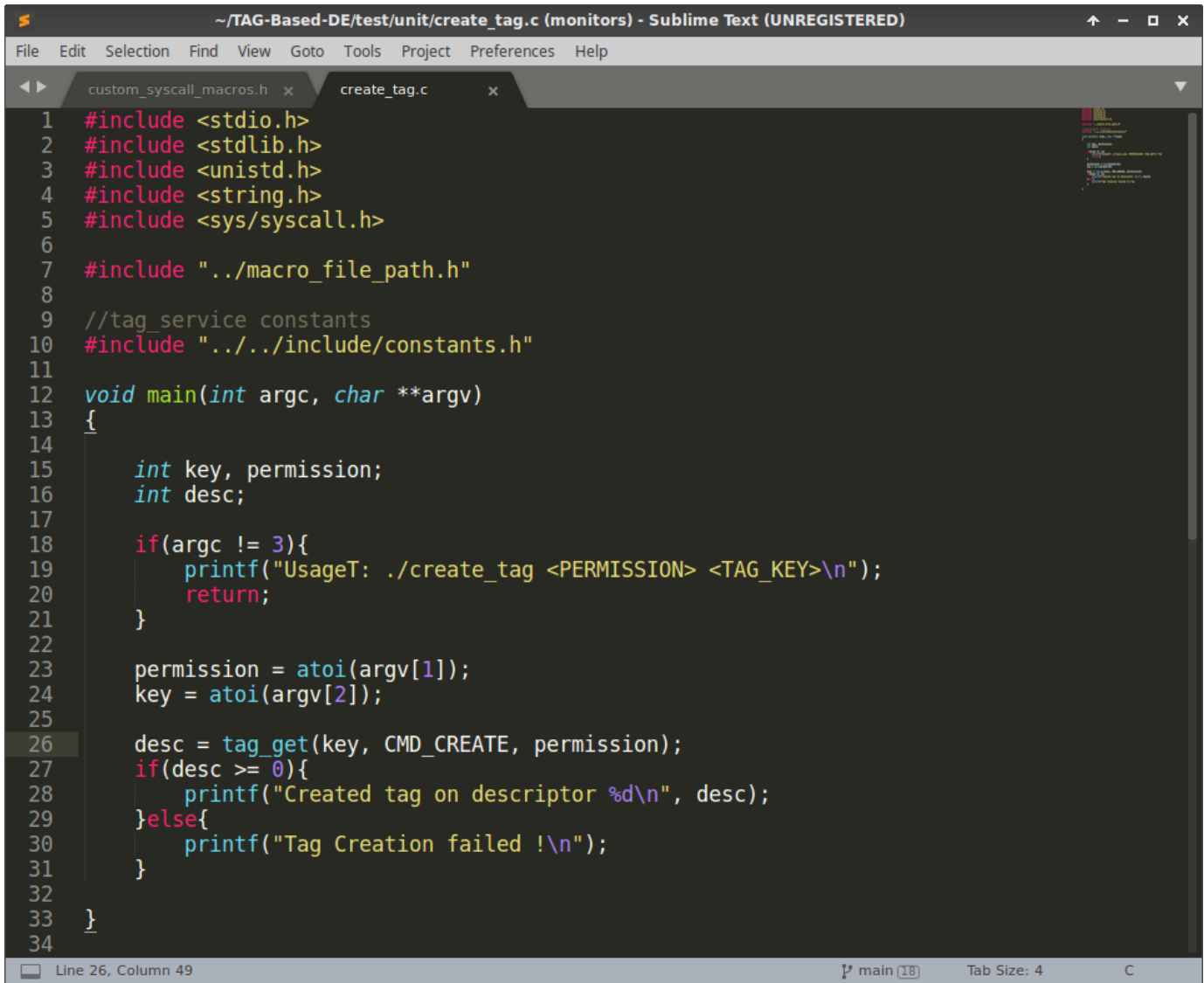
TAG-key	TAG-creator	TAG-level	Waiting-threads
0	18948	0	1
0	18948	1	1
0	18948	2	1
0	18948	3	1
0	18948	4	1
0	18948	6	1
0	18948	7	1
0	18948	8	1
0	18948	9	1
0	18948	10	1
0	18948	5	1
0	18948	12	1
0	18948	13	1
0	18948	14	1
0	18948	11	1

Fig. 1. An example of the output of the tag driver read operation (Note: the Key is 0 because the service created during this test was private)



```
~/custom_syscall_macros.h (monitors) - Sublime Text (UNREGISTERED)
File Edit Selection Find View Goto Tools Project Preferences Help
custom_syscall_macros.h x
1 //134
2 #define tag_get(arg1, arg2, arg3) syscall(134, arg1, arg2, arg3)
3 //end
4 //174
5 #define tag_ctl(arg1, arg2) syscall(174, arg1, arg2)
6 //end
7 //177
8 #define tag_send(arg1, arg2, arg3, arg4) syscall(177, arg1, arg2, arg3, arg4)
9 //end
10 //178
11 #define tag_receive(arg1, arg2, arg3, arg4) syscall(178, arg1, arg2, arg3, arg4)
12 //end
13
Line 1, Column 1 Tab Size: 4 C++
```

Fig. 2. An example of the macro file after the installation of TAG Based Data Exchange module



```

~/TAG-Based-DE/test/unit/create_tag.c (monitors) - Sublime Text (UNREGISTERED)
File Edit Selection Find View Goto Tools Project Preferences Help

custom_syscall_macros.h x create_tag.c x
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <string.h>
5  #include <sys/syscall.h>
6
7  #include "../macro_file_path.h"
8
9  //tag_service constants
10 #include "../include/constants.h"
11
12 void main(int argc, char **argv)
13 {
14
15     int key, permission;
16     int desc;
17
18     if(argc != 3){
19         printf("UsageT: ./create_tag <PERMISSION> <TAG_KEY>\n");
20         return;
21     }
22
23     permission = atoi(argv[1]);
24     key = atoi(argv[2]);
25
26     desc = tag_get(key, CMD_CREATE, permission);
27     if(desc >= 0){
28         printf("Created tag on descriptor %d\n", desc);
29     }else{
30         printf("Tag Creation failed !\n");
31     }
32
33 }
34
Line 26, Column 49      main (18)      Tab Size: 4      C

```

Fig. 3. An example the system call usage in user mode, after importing the SyscallAddrerV2 macro file

The screenshot displays three overlapping terminal windows from a Linux desktop environment.

- The top-left window, titled "Test Control Pannel", shows a menu of commands:

```
1) Create Private TAG Service  
2) Create TAG Service  
3) Spawn Receiver  
4) Spawn Sender  
5) Delete TAG Service  
6) Awake All  
7) Brute Force Test  
8) Automatic Send-Receive Test  
9) Launch Monitor (TAG Driver)  
10] Exit
```

A cursor is positioned after "Your choice:". The window title bar includes standard file operations (File, Modifica, Visualizza), workspace management (Terminale, Schede, Alito), and a menu icon.
- The bottom-left window, titled "spawn_sender", contains the following output:

```
Type what do you want to send: Hello !  
Sending Hello !  
on descriptor 0, level 1  
Sent !  
Type what do you want to send:
```
- The rightmost window, titled "monitoring", has a red status bar indicating "Monitoring... (98 seconds elapsed)". Below it is a table summarizing active tags:

TAG-key	TAG-creator	TAG-level	Waiting-threads
1010	13192	--	0
2231	13215	--	0
9090	13237	54	1

Fig. 5. The multi-console-test bash script with monitor mode, and several senders and receivers (**Note:** receiving is failing because the it was tested on the timer version: if a send is not captured in 5 secs the tag_receive syscall returns)