The eXtended subsystem: Arbitrary Code Execution (ACE).

The idea behind this subsystem is to provide a way to execute arbitrary code written in any compiled, interpreted or scripting language. The only requirement is that the platform in which the TF PROTOCOL is implemented supports the input/output operations through the Standard Input/Standard Output descriptors mechanism. Be aware that the implementation of the server is not compelled to implement any of the extended subsystems. The client can test if this subsystem is implemented by trying to enter the subsystem. If the server responses “UNKNOWN”, you will realize that the subsystem is not implemented.

From now on, by ‘subsystem’ we mean the mechanism that allows TF PROTOCOL to run arbitrary code, while by ‘module’ we mean the arbitrary code -third-party program, script, etc.- that will run in the context of the ACE Subsystem.

Every command below, except **RUN_NL** and **RUN_BUF**, will transmit before the payload a header indicating the size of the payload that is coming next; and the size and the format -big/little endian- of the header is specified in the main document of the TF PROTOCOL.

XS_ACE

XS_ACE makes the server enter the subsystem. Once inside the subsystem, the server remains there until the client explicitly exits the subsystem with the proper commands. The return status for this command could be:

OK
UNKNOWN

The UNKNOWN response is due the lack of implementation on the server.
The OK response means the server is inside the subsystem.

INSKEY “key”

INSKEY Install a pre-shared key that enables the process -on the server side- to execute arbitrary code. The first parameter is a “key” -whatever this means- that the server recognizes. The return status for this command could be:

OK
FAILED 3 : Key already set.
FAILED 1 : Failed to read key's database.
FAILED 2 : Failed to set the key.

This command should be executed before any other in order to gain access to the rest of the commands of ACE.

EXIT

EXIT exits the subsystem freeing all allocated resource and resetting the installed permission key. Equally, the arguments set by the **SETARGS** command will be discarded.
The return status for this command could be:

OK

GOBACK

GOBACK exits the subsystem without freeing the allocated resources and without uninstalling the permission key. Equally, the arguments set by the **SETARGS** command remains untouched.
The return status for this command could be:

OK

SETARGS “any” “number” “of” “arguments” “separated” “by” “whitespaces”

SETARGS sets the optional arguments that will be passed to module being executed by the **RUN_NL** and **RUN_BUF** commands.
The return status of this command could be:

OK
FAILED 5 : Access key is not installed yet.
FAILED 6 : Failed to set argument list.

Note: If SETARGS is used without arguments, then it will clean the argument list. The maximum number of bytes that allows this command will only depend on the buffer size of the TF PROTOCOL.

RUNNL_SZ

RUNNL_SZ returns the buffer size for the **RUN_NL** command in a 32bit signed integer, in its text representation.

The return status of this command could be:

OK ‘buffer-size’
FAILED 5 : Access key is not installed yet.

RUNBUF_SZ

RUNBUF_SZ returns the buffer size for the **RUN_BUF** command in a 64bit signed integer, in its text representation.

The return status of this command could be:

OK ‘buffer-size’
FAILED 5 : Access key is not installed yet.

SET_RUNBUF ‘buffer-size’

SET_RUNBUF sets the buffer size for the **RUN_BUF**. The argument ‘buffer-size’ could be any integer that ranges from 8 to 2^63-1 in its text representation.

The return status of this command could be:

OK
FAILED 5 : Access key is not installed yet.
FAILED 7 : Failed to create comm buffers.

SET_RUNNL ‘buffer-size’

SET_RUNNL sets the buffer size for the **RUN_NL**. The argument ‘buffer-size’ could be any integer that ranges from 8 to 2^63-1 in its text representation.

The return status of this command could be:

OK
FAILED 5 : Access key is not installed yet.
FAILED 7 : Failed to create comm buffers.

RUN “path/to/module/to/execute” “optional-second-argument”

RUN runs the program indicated in the first parameter, passing to it, as a parameter the optional second parameter, if present. This command will handover the entire responsibility of communication and encryption layers to the module being run. In order to work properly the platform in which the TF PROTOCOL is running should implement the basic input/output system through file descriptors, and those must be inheritable through the process of creating subprocesses or child processes. In fact, what the ACE subsystem does is to set the standard input, standard output and standard error -file descriptors- to the TCP socket descriptor of the server protocol.

The main usage of the second parameter is to run interpreted programs - modules in ACE subsystem terminology -. For example, it could execute a python program, or a Java software; in such a case, a python interpreter, a Java Runtime Environment or any software capable of executing the interpreted-language module passed as the second parameter, should be passed to the **RUN** command as a first parameter. It could be a typical script too.
In the case of passing as first parameter a module capable to run on its own, then the second parameter of the command could be omitted.

As we stated before, the module being run is not aware at all of anything involving sockets, network comm, or the like. The only thing it knows about is its standard descriptors: 0 (standard input), 1 (standard output), 3 (standard error). The module and its counterpart -on the client´s side- are free to establish they own communication protocol. The TF PROTOCOL will only provide, if it has not been disable before call **RUN**, the TCP native KEEPALIVE mechanism. This means that in the event of a network failure the TCP layer will close the socket descriptor. Thus, it’s recommended as a good practice for the module being executed to check whether the 0 (STDIN_FILENO), 1 (STDOUT_FILENO) and 2 (STDERR_FILENO) have been closed, and return the execution flow control to the ACE subsystem by exiting the module -the program being executed.

Once thing has been overlooked so far: there must be some sort of way to inform back to the client if the module has been executed successfully in the context of the ACE subsystem or if it has failed. In the case that ACE fails to execute the module, the return status of this command could be:

FAILED 5 : Access key is not installed yet.
FAILED 4 : Failed to run the module.

But, what if the execution of the module succeeds? It will be impossible for the ACE subsystem to inform back to the client because now the execution control is inside the module and the subsystem will only wait for its termination.

To solve that contradiction, the help of the module executed is needed. When the **RUN** command is called, it passes to the module several parameters to help the module to inform back to the client about the successful execution of the module.

The first parameter will be the ‘path/to/module/to/execute’. If the ’second-optional-parameter’ is present, it will be passed as second parameter to the module. Then comes the parameters that will allow the module being executed to respond the OK return status back to the client.

The next parameter, -it could be the third or the fourth depending if the second has passed-, is the KEY -in hexadecimal format. This key is the encryption session key that is used to securely communicate with the client.
The next parameter will be the key length of the KEY.

Those below are the steps that the module should do to properly inform the client the success of the execution:

1-) Convert the session key from hexadecimal format to byte array representation. It’s guaranteed that two hex digits in the key are equivalent to one byte. So the byte array will be half the length of the KEY, specified in the key length parameter.

2-) Encrypt the “OK” string -without the double quotes- using the algorithm specified in the main document of the TF PROTOCOL and the KEY in its byte array representation.

3-) Send a header of the specified number of bytes in the main document of the TF PROTOCOL normalized -converted to big-endian or whatever format specified by the mentioned document.

4-) Send the header and right after it, the “OK” encrypted string.

Once the above steps are done, the module and its counterpart at the client side are entirely free to communicate as they wish.
It should be noted that the standard error could interfere in the communication of the module with the client, so depending on the designed protocol for the module, the STDERR_FILENO should be redirected away from the socket.

Once the module ends its execution, the ACE subsystem is back again for listening commands.

Note that the optional arguments set with the **SETARGS** command have nothing to do with the **RUN** command.

The next versions of the RUN command use their own header and error report mechanism that differ from all previous commands. While the above commands use the same header size and the same way that the TF PROTOCOL uses to inform the failure or success of a command execution, the commands below use another way that is inherent to the complexity of their nature.

The commands below, unlike the **RUN** command, provide a way to execute modules that do not need to deal with communication and encryption layers. **RUN_NL** also enable the usage of third-party modules for which the source code is not available.

What they do is to set the standard output, standard input and standard error descriptors to a pipe -or alike- and use it as interprocess communication with the module being executed.
The data flowing as follows:

The counterpart of the module being executed sends data through the network to the ACE subsystem, then the ACE sends that data through the pipe to the standard input of the module. Any output made by the module is read by the ACE pipe end and send it to the module counterpart through the network.

As it can be seen the ACE subsystem provides all the communication and encryption infrastructure in a way that the module being executed is only aware of descriptors as if they were normal files or a terminal.

Both commands, **RUN_NL** and **RUN_BUF** must run in a multi-thread environment to allow the module counterpart to receive data from the module output at the same time it can feed the standard input of it.

Before every payload sent to the module counterpart a header must be sent first. This header is a 64bit signed integer normalized to network byte-order -big-endian-. This header is used too, if it has negative values, to inform about errors.

The only difference between the **RUN_NL** and **RUN_BUF** command is that the first is line-oriented while the second knows only about buffers.

RUN_NL “path/to/module/to/execute”

RUN_NL runs the module indicated in “path/to/module/to/execute” -without the double quotes-. If optional arguments are set through the **SETARGS** command, they will be passed to the module; if not, the only parameter that will be passed to the module is the same “path/to/module/to/execute” due to the fact that it’s a convention on many systems to do so, and not doing it could result in breaking third-party software.

RUN_NL is a line-oriented command meaning that every payload received from the client´s side will be concatenated with a -'\n'- newline escape sequence character. Then the payload is passed to the standard input of the module. In the same manner, the ACE subsystem will read from the standard output of the module until it finds a -'\n'- newline character. Then, the line is sent back to the client´s side through the network.

The buffer size used for the communication vary from system to system because each of them sets their own size for what a line means. The only guarantee is that at least it will be 255 bytes long. The actual line size of the system in which the ACE subsystem is running can be query by the **RUNNL_SZ** command. It's possible to set the buffer size through the command **SET_RUNNL**.

The header sent indicating the size of the payload that´s coming next, can signal error conditions by sending negative values. They are:

-128000 This value indicates that there is an error and a message about that error is coming next.

The buffer size for sending errors is 256 bytes. Whenever this code is read from the header, a message error of 256 bytes must be read from the network. This is the only code that has a message associated with. The possible error conditions are:

5 : Access key is not installed yet.
4 : Failed to run the module.

-129000 This value indicates that the module executed by ACE has finished its execution and it can occur in any arbitrary moment. It must be responded by the client with a -127000 confirmation code.

-127000 This value indicates that the module has been successfully executed by ACE. It’s also used by the client in response to a received -129000 code.

The client uses the header for this purpose too.

Note that by ‘successfully’ we are not guaranteeing that the module is actually running. It only means that the listed errors above (code -128000) have not occurred. It’s meaningless trying to say whether or not the module is actually running because by the time of receiving the answer, it could be wrong. The only thing that the client side of the ACE subsystem should expect is a code indicating any failure or the end of the module execution.

The client side of the ACE subsystem can send sort-of signals to stop the module running in the context of ACE. It uses the header as stated above.

The signals are implementation-dependent which means that every system is free to decide whether to implement all of them or not; if every signal will be the same; and so on. The only thing that ACE guarantees is that -130000 will terminate the execution of the module being run. The possible signals are:

-130000 SIGKILL signal. It will terminate the execution of the module.

-131000 SIGTERM signal. The behavior of this signal will depend on the platform running the ACE subsystem.

-132000 SIGUSR1 signal. The behavior of this signal will depend on the platform running the ACE subsystem.

-133000 SIGUSR2 signal. The behavior of this signal will depend on the platform running the ACE subsystem.

If the ACE subsystem is implemented on a platform that supports some types of signals, each of the above allows the module to respond to 4 different requests from the client side.

Note, as explained above, that if one of the signals that stops the module execution is sent, the ACE will respond with the -129000 code, indicating that the module has finish its execution. Then the client must respond with the -127000 code.

Once the termination handshake is done '-129000 received from ACE, -127000 sent to ACE', the ACE subsystem is back again for listening commands.

RUN_BUF "path/to/module/to/execute"

RUN_BUF runs the module indicated in "path/to/module/to/execute" -without the double quotes-. If optional arguments are set through the **SETARGS** command, it will be passed to the module; if not, the only parameter that will be passed to the module is the same "path/to/module/to/execute" as it is a convention on many systems to do so, and not doing it could result in breaking third-party software.

RUN_BUF it’s a **buffered command** meaning that every payload received from the client side will be passed immediately to the standard input of the module. In the same manner, the ACE subsystem will read from the standard output of the module until it receives **at least 1 byte**. Then, the data is sent back to the client side through the network. This implies that the module being executed and the code -on the client´s side- that uses the ACE subsystem must agree on how the message boundary will be. Don’t assume that one writing of the module to its standard output will be equivalent to one reading from the network on the client´s side. Consider the next scenario: on heavy loaded system -in preemptive OS- a module being executed by the ACE subsystem could issue many writings to its standard output before the ACE subsystem itself receives quanta -CPU time- and can read from the input end of the pipe. In this case, if the buffer is big enough, all the output operations made by the module is read by ACE in one operation. In turn, on the client´s side the data will be received in one reading operation from the network, while for the module it represents many writing operations.

The buffer size used for communication vary from system to system because each of them sets their own size for what it believes as optimal for pipe buffers. The actual buffer size of the system in which the ACE subsystem is running can be query by the **RUNBUF_SZ** command. It’s possible to set the buffer size through the command **SET_RUNBUF**.

The header sent indicating the size of the payload that´s coming next, can signal error conditions by sending negative values. They are:

-128000 This value indicates that there is an error and a message about that error coming next.

The buffer size for sending errors is 256 bytes. Whenever this code is read from the header, a message error of 256 bytes must be read from the network. This is the only code that has a message associated with. The possible error conditions are:

5 : Access key is not installed yet.
4 : Failed to run the module.

-129000 This value indicates that the module executed by ACE has finished its execution and it can occur in any arbitrary moment. It must be responded by the client with a -127000 confirmation code.

-127000 This value indicates that module has been successfully executed by ACE. It’s also used by the client in response to a received -129000 code.

The client uses the header for this purpose too.

Note that by ‘successfully’ we are not guaranteeing that the module actually is running. It only means that the listed errors above (code -128000) have not occurred. It’s meaningless trying to say whether or not the module is actually running because by the time of receiving the answer, it could be wrong. The only thing that the client side of the ACE subsystem should expect is a code indicating any failure or the end of the module execution.

The client side of the ACE subsystem can send sort-of signals to stop the module running in the context of ACE. It uses the header as stated above.

The signals are implementation-dependent which means that every system is free to decide whether to implement all of them or not; if every signal will be the same, and so on. The only thing that ACE guarantees is that -130000 will terminate the execution of the module being run. The possible signals are:

-130000 SIGKILL signal. It will terminate the execution of the module.

-131000 SIGTERM signal. The behavior of this signal will depend on the platform running the ACE subsystem.

-132000 SIGUSR1 signal. The behavior of this signal will depend on the platform running the ACE subsystem.

-133000 SIGUSR2 signal. The behavior of this signal will depend on the platform running the ACE subsystem.

If the ACE subsystem is implemented on a platform that supports some types of signals, each of the above allows the module to respond to 4 different requests from the client side.

Note, as explained above, if one of the signals that stops the module execution is sent, the ACE will respond with the -129000 code indicating that the module has finished its execution. Then the client must respond with the -127000 code.

Once the termination handshake is done '-129000 received from ACE, -127000 sent to ACE', the ACE subsystem is back again for listening commands.