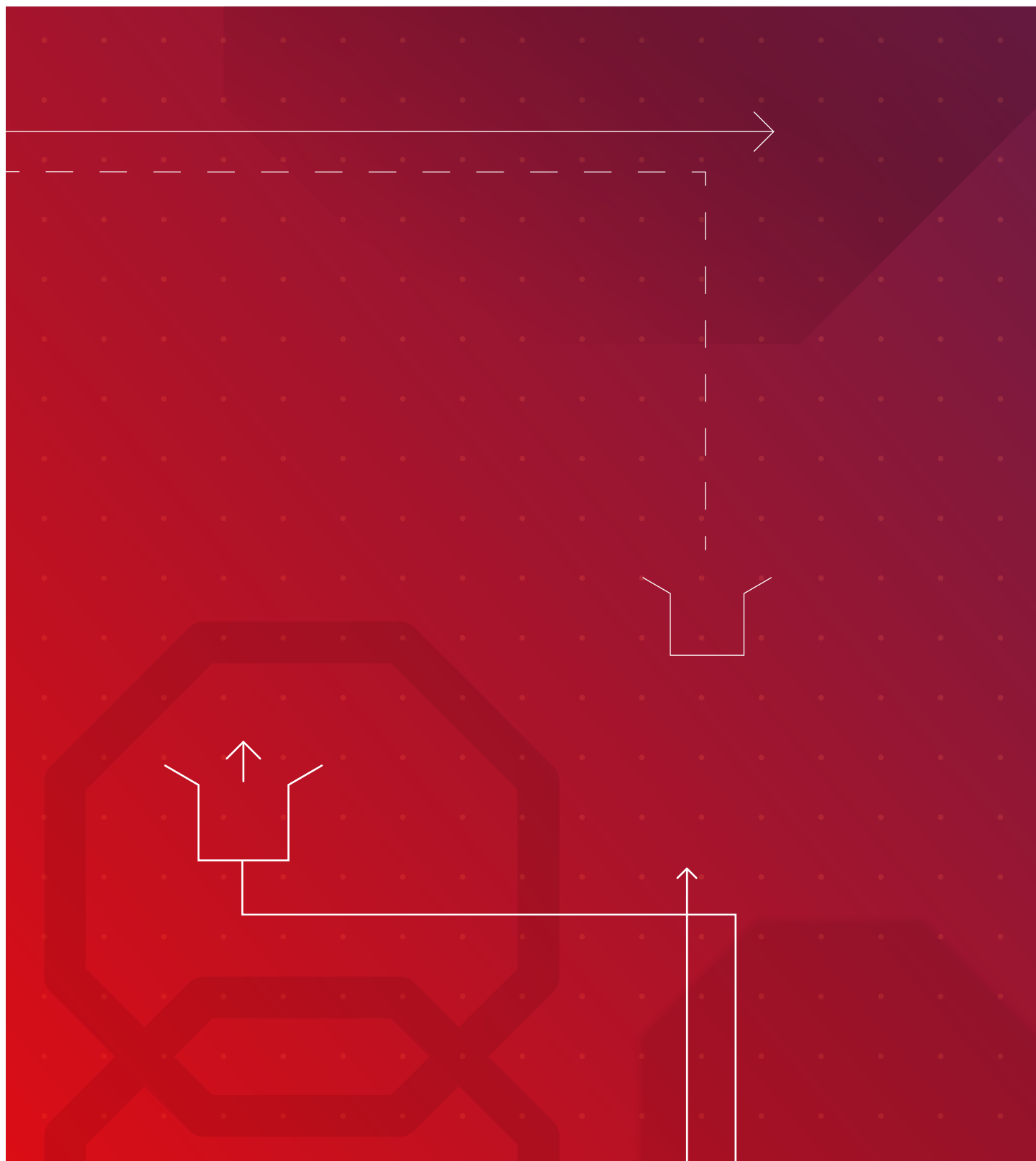


# Anti-malware SDK

## Library Guide (Unix)



# Contents

<b>1 Introduction.....</b>	<b>3</b>
1.1 Disclaimer.....	3
 <b>2 Compiling and running the example.....</b>	 <b>3</b>
2.1 Compiling the examples.....	3
2.2 Running the examples.....	4
 <b>3 SAVAPI Library.....</b>	 <b>4</b>
3.1 SAVAPI initialization.....	4
3.2 SAVAPI instances.....	5
3.3 SAVAPI scanning.....	5
3.4 SAVAPI scan callbacks.....	6
3.5 SAVAPI scan options.....	6
3.6 SAVAPI log callback.....	7
 <b>4 Contact information.....</b>	 <b>8</b>
4.1 Support services.....	8
4.2 Contact.....	8



# 1 Introduction

## 1.1 Disclaimer

The purpose of these simple examples is to show basic functionality of the features as easy as possible. There are no error checks, no log files, and very few console prints in order to preserve the simplicity of the examples. Using code from these basic examples "as is", in a production environment is not recommended. Instead, check the more advanced examples that are safer to use when implementing SAVAPI.



**Note** On UNIX platforms, in order to use native UTF-8 char types, `savapi_unix.h` must be included instead of `savapi.h`.

The SAVAPI Client Library basic examples are split into eight incremental complexity levels:

1. **`clientlib_basic_init_example`** - introduces initialization of SAVAPI Client Library; it doesn't scan files
2. **`clientlib_basic_instance_example`** - introduces the creation of SAVAPI instances; it doesn't scan files
3. **`clientlib_basic_scan_example`** - introduces scanning of a file using default settings and options, without callbacks
4. **`clientlib_basic_scan_status_example`** - introduces callbacks implementation to check scan result
5. **`clientlib_basic_scan_options_example`** - introduces setting of basic scanning options
6. **`clientlib_basic_scan_callbacks_example`** - introduces the most important callbacks implementation
7. **`clientlib_basic_log_callbacks_example`** - introduces logging callback
8. **`clientlib_basic_complete_example`** - introduces checking of return codes; the safest basic example to be used in production code

## 2 Compiling and running the example

### 2.1 Compiling the examples

CMake is used as the build tool for the SAVAPI examples and must be installed before compiling them. It allows to generate build files for various platforms and also build them using an installed toolchain.

To build the examples, navigate to an extracted SAVAPI SDK folder and run the commands below. CMake will detect an available toolchain and a build tool, and then build the examples directly into the SAVAPI SDK's `/bin` folder.

Note: It is mandatory to pass to CMake `<SAVAPI SDK>/examples` folder as a source folder (i.e. `"-S ../examples"` in this case).

```
# mkdir examples_build
# cd examples_build
# cmake -S../examples -DPLATFORM_ARCH=${arch} -DCMAKE_BUILD_TYPE=${type}
```

Then to build all the examples run:

```
# cmake --build .
```

Or to build a specific example pass its name to CMake like this:

```
# cmake --build . -t lib_file_scan_example
```

Where:

```
- arch=[x64|x86|arm64]
- type=[Release|Debug]
```

Please refer to CMake documentation for more information.



Build files generated by CMake can be used separately with an IDE (for ex., Visual Studio), `nmake`, `make` etc.

## 2.2 Running the examples

Because by default, the examples search for some modules that are located in the *bin* directory, they must be executed from the *bin* directory.

Each example can run without arguments (the very basic ones) or will require one or two arguments, depending on its complexity. By executing an example with no arguments, it will print its usage to the console.

The possible arguments for basic examples are:

`license product id`: License product ID, provided by Avira

`file to scan`: The file to be scanned

```
# cd /opt/savapi/bin

# ./lib_basic_scan_status_example
SAVAPI Library usage in a basic scan with status answer example program
(c) Avira Operations GmbH & Co. KG 2016

Usage: <./lib_basic_scan_status_example> <license_product_id> <file_to_scan>

# ./lib_basic_scan_status_example 1 /tmp/eicar.com
SAVAPI_initialize succeeded
SAVAPI_create_instance succeeded
    File '/tmp/eicar.com' is infected!
    malware name: Eicar-Test-Signature
    malware info: Contains code of the Eicar-Test-Signature virus
SAVAPI_scan succeeded
```

## 3 SAVAPI Library

### 3.1 SAVAPI initialization

Before most of the SAVAPI functions can be used, SAVAPI must be initialized using `SAVAPI_initialize()`. This function takes a single parameter of type `SAVAPI_GLOBAL_INIT` as argument.

The needed important members of the `SAVAPI_GLOBAL_INIT` structure are:

`api_major_version`: The minimum major API version expected

`api_minor_version`: The minimum minor API version expected

`program_type`: License product ID, provided by Avira

`engine_dirpath`: Directory containing the SAVAPI engine files

`vdfs_dirpath`: Directory containing Avira Virus Definition (VDF) files

`key_file_name`: Path to the license key file (HBEDV.KEY) provided by Avira

Create and initialize the structure:

```
/* Declare a SAVAPI global object and set the required fields
 * (license product id is only needed when scanning) */
SAVAPI_GLOBAL_INIT global_init = {0};
global_init.api_major_version = SAVAPI_API_MAJOR_VERSION;
global_init.api_minor_version = SAVAPI_API_MINOR_VERSION;
global_init.program_type = atoi(argv[1]);
```

In the basic examples, the license product ID is received as a program argument.



Also, the `engine_dirpath`, `vdfs_dirpath` and `key_file_name` are not set, so SAVAPI will search them in the current directory. If they are not located in the current directory, they must be explicitly set before initializing SAVAPI.

```
/* Example code */
global_init.engine_dirpath = "/opt/savapi/engine_dir";
global_init.vdfs_dirpath = "/opt/savapi/vdf_dir";
global_init.key_file_name = "/opt/savapi/key_dir/HBEDV.KEY";
```

After creating the structure, call `SAVAPI_initialize()`:

```
/* Initialize SAVAPI library */
ret = SAVAPI_initialize(&global_init);
```

SAVAPI is initialized now, more code can be added after the initialization.

A proxy server can be set for the SAVAPI FPC module. Setting it is optional. It must be done after initializing SAVAPI, but before creating any SAVAPI instances.

```
/* Example code */
/* If needed, set a proxy server for the FPC connection */
/* SAVAPI_global_set(SAVAPI_OPTION_G_PROXY, "<your-proxy-here>"); */
```

Before exiting the program, SAVAPI must be uninitialized.

```
ret = SAVAPI_uninitialize();
```

## 3.2 SAVAPI instances

In order to use SAVAPI scanning, one needs to create at least one SAVAPI instance. A program can have many instances running at the same time, each instance having its own unique set of options. Each instance usually has to be attached to a run time “worker thread” in the context of the running program, and the user is responsible for all thread creation and management.

To create an instance, the user needs to call `SAVAPI_create_instance()` and retrieve an instance handle of type `SAVAPI_FD`. The instance handle is then used in subsequent calls to set options, scan files, memory and hashes.

```
/* Declare objects to prepare the instance creation */
SAVAPI_FD instance_handle = NULL;
SAVAPI_INSTANCE_INIT instance_init = {0};

/* Create the instance and return an instance_handle that will be used to set
 * different scanning options and scan files, memory or hashes */
ret = SAVAPI_create_instance(&instance_init, &instance_handle);
```

All created instances should be destroyed by calling `SAVAPI_release_instance()`.

```
ret = SAVAPI_release_instance(&instance_handle);
```

## 3.3 SAVAPI scanning

Once the instance has been created, it can be used to scan files, memory or hashes by calling `SAVAPI_scan()`.

```
/* File to be scanned must be received as second argument */
char *file_to_scan = argv[2];
ret = SAVAPI_scan(instance_handle, file_to_scan);
```

It is important to note that `SAVAPI_scan()` does not return until the scan operation is complete. In order to scan multiple objects at the same time, it is therefore necessary to use threading, along with multiple SAVAPI instances. Each instance must run in a different thread context. The developer implementing SAVAPI in his application is responsible for maintaining the multithreaded code that makes `SAVAPI_scan()` calls to the SAVAPI instances.



**Note** `SAVAPI_scan()` does not return the status of the scan (clean or infected) but only a success or failure code. In order for the user to get more granular results, it is necessary to implement scan callbacks.

### 3.4 SAVAPI scan callbacks

SAVAPI uses callbacks during scanning to send information to the user. The callbacks can be registered by calling `SAVAPI_register_callback()` and are unique for each instance. All relevant callbacks must be set on a SAVAPI instance before calling `SAVAPI_scan()` for that instance.

The most important callback is `SAVAPI_CALLBACK_REPORT_FILE_STATUS`, which contains information about the file scanned. If the file is infected, there is additional information available about the malware.

```
ret = SAVAPI_register_callback(instance_handle,
    SAVAPI_CALLBACK_REPORT_FILE_STATUS, file_status_callback);
```

The basic implementation of `SAVAPI_CALLBACK_REPORT_FILE_STATUS` function is:

```
/* Triggered after a file is scanned, contains information
 * about the file status (clean, infected etc.) */
static int file_status_callback(SAVAPI_CALLBACK_DATA *data)
{
    SAVAPI_FILE_STATUS_DATA *file_status_data =
        data->callback_data.file_status_data;

    if (file_status_data->scan_answer == SAVAPI_SCAN_STATUS_INFECTED)
    {
        printf("\tFile '%s' is infected!\n", file_status_data->file_info.name);
        printf("\tmalware name: %s\n", file_status_data->malware_info.name);
        printf("\tmalware info: %s\n", file_status_data->malware_info.message);
    }

    return 0;
}
```

Once a callback has been registered, it must be unregistered by calling `SAVAPI_unregister_callback()`.

```
ret = SAVAPI_unregister_callback(instance_handle,
    SAVAPI_CALLBACK_REPORT_FILE_STATUS, file_status_callback);
```

Other important callbacks are:

- `SAVAPI_CALLBACK_PRE_SCAN` - Triggered before the scanning begins. Can be used to create filters. For example, if the user wishes to scan only .exe files, the user installs a `PRE_SCAN` callback. Before any file is scanned, the `PRE_SCAN` callback is called. If the returned code is success (`SAVAPI_S_OK`), the file will be scanned, otherwise it will be skipped.
- `SAVAPI_CALLBACK_ARCHIVE_OPEN` - Triggered before an archive (e.g. ZIP, RAR, MIME encoded files etc) is opened. If the returned code is success (`SAVAPI_S_OK`), the archive will be opened, otherwise it will be skipped.
- `SAVAPI_CALLBACK_PROGRESS_REPORT` - Triggered multiple times during a scan operation, when messages related to the scan progress are available. Useful in a lengthy scan operation on a large archive or file.
- `SAVAPI_CALLBACK_REPORT_ERROR` - Triggered on errors encountered during a scan, i.e. damaged files or archives etc. Also triggered on warnings encountered during a scan. Can be triggered at any time during the scan.

### 3.5 SAVAPI scan options

SAVAPI has default scanning options for basic malware but in many cases additional options are needed for optimal performance. For example, by default SAVAPI will not scan in archives, so `ARCHIVE_SCAN` must be enabled. The options are set per instance, so different SAVAPI instances can have different options set for them.



The options can be set by calling `SAVAPI_set()`.

```
/* Enable false positive control */
ret = SAVAPI_set(instance_handle, SAVAPI_OPTION_FPC, "1");

/* Enable archive scanning */
ret = SAVAPI_set(instance_handle, SAVAPI_OPTION_ARCHIVE_SCAN, "1");

/* Set maximum allowed size (in bytes) for any file within an archive */
ret = SAVAPI_set(instance_handle, SAVAPI_OPTION_ARCHIVE_MAX_SIZE, "0");

/* Enable detection for all categories */
ret = SAVAPI_set(instance_handle, SAVAPI_OPTION_DETECT_ALLTYPES, "1");
```



**Note** In order to have the best detection, it is highly recommended to enable the False Positive Control module (set `SAVAPI_OPTION_FPC` to "1"). An Internet connection is needed for the FPC module to work.

Also, by calling `SAVAPI_get()`, values of some specific options can be obtained.

```
char buf[BUF_SIZE] = {0};
SAVAPI_SIZE_T buf_size = BUF_SIZE;

/* Get some instance options */
ret = SAVAPI_get(instance_handle, SAVAPI_OPTION_FPC, buf, &buf_size);
printf("False positive control: %s\n", buf);

ret = SAVAPI_get(instance_handle, SAVAPI_OPTION_AVE_VERSION, buf, &buf_size);
printf("Engine version: %s\n", buf);

ret = SAVAPI_get(instance_handle, SAVAPI_OPTION_VDF_VERSION, buf, &buf_size);
printf("VDF version: %s\n", buf);

ret = SAVAPI_get(instance_handle, SAVAPI_OPTION_ARCHIVE_SCAN, buf, &buf_size);
printf("Scan in archives: %s\n", buf);
```

## 3.6 SAVAPI log callback

The logging callback is one of the few SAVAPI functions that can be called before the SAVAPI Client Library is initialized.

The user must implement his own logging callback function, that will be called anytime SAVAPI logs something. In the example, a log file is created and filled with SAVAPI logging information.

```
/* Define a structure to be passed as user data to the log callback */
log_user_data_t log_user_data = {0};
log_user_data.log_file = fopen("savapi.log", "w");

/* Set the logging callback function (can be done before SAVAPI_initialize) */
ret = SAVAPI_set_log_callback(&log_callback, SAVAPI_LOG_DEBUG, &log_user_data);
```



The logging function example below will append the messages received from SAVAPI to the *savapi.log* file. SAVAPI will provide the message string, but time-stamping and output formatting are up to the developer.

```
/* Triggered when SAVAPI logs a message */
static void log_callback(SAVAPI_LOG_LEVEL log_level,
                        const SAVAPI_TCHAR *message, void *user_data)
{
    /* get the log file handle and write the message
     * received from SAVAPI to the file */
    log_user_data_t *log_user_data = (log_user_data_t*)user_data;

    /* retrieve the local time to be written in the log file */
    time_t local_time = time(NULL);
    struct tm *local_tm = localtime(&local_time);

    fprintf(log_user_data->log_file, "%02d:%02d:%02d %s: %s\n",
            local_tm->tm_hour, local_tm->tm_min, local_tm->tm_sec,
            log_level_to_string(log_level), message);
}
```

Close the file handle before exiting the program.

```
/* Close the log file handle */
fclose(log_user_data.log_file);
```

## 4 Contact information

### 4.1 Support services

#### During evaluation and integration

If you are evaluating or starting to integrate Avira's Technology into your solution, the System Integration Team will answer your technical questions, from planning the architecture of the integration, to detailed code-related routines.

To contact the SI Team for technical issues, <mailto:si-support@avira.com>

#### After the integration is finalized

As soon as you finalize the integration of Avira's Technology and you release your solution to your customers, the OEM Integration Support manages all support issues.

To contact the OEM Support Team for technical issues, <mailto:oemsupport@avira.com>

#### Partner Portal

For our OEM customers we also provide a login to our Partner Portal with all the latest news and information about Avira's technology, SDK downloads, and documentation: <https://oem-portal.avira.com>

### 4.2 Contact

#### Address

Avira Operations GmbH  
Kaplaneiweg 1  
D-88069 Tettnang  
Germany

#### Internet

You can find further information about us and our products on the website: [www.avira.com](http://www.avira.com)



Europe Middle East, Africa	Americas	Asia/Pacific and China	Japan
<b>Avira</b> Kaplaneiweg 1 88069 Tettngang, Germany Tel: +49 7542 5000	<b>Avira, inc</b> c/o WeWork, 75 E Santa Clara Street Suite 600, 6th floor San José CA 95113 United States	<b>Avira Pte Ltd</b> 50 Raffles Place 32-01 Singapore Land Tower Singapore 048623	<b>Avira GK</b> 8F Shin-Kokusai Bldg 3-4-1, Marunouchi Chiyoda-ku Tokyo 100-0005, Japan