Killerbeez API

GRIMM

2017.12.22

Contents

1 Overview

This document will cover the specific API of each module, along with a quick high-level summary of what it does.

The APIs are all specified in C, as this provides a consistent language and is explicit about data types which means there's no need for a separate Python specification. The C code is frequently wrapped with Python (via ctypes), but modules are typically written in C code as they run considerably faster when it is all native code.

2 Manager

The manager is what coordinates a fuzz job. It decides which seeds to use, which mutators to run and sends this information to the client, which kicks off one or more Main Fuzzers. The client will then also handle getting the results back to the manager.

3 Main Fuzzer

This will run many iterations of a single seed and a single mutator against a target program. For efficiency, this will be run on the same computer (which means same O/S) as the target binary. This component will be an executable that the manager executes on each of the target systems. The arguments for this function are defined in the usage function of the fuzzer.

4 Mutator

The mutator modules are what actually mutate the seed buffers. These would include things like a bit flipper, byte munger and so forth. They are given an input buffer and optionally some state information. The state information is module-specific and allows the mutator to pick up where it left off. For example, the bit flipper mutator module, which simply flips one bit in the input buffer, would just need to record what bit to flip as their state. On the other hand, more complicated mutators may need to keep track of more information. Additionally, each mutator will have a variety of mutator specific configuration options that can be specified. Both the mutator state and options will be specified as JSON char arrays.

Anything which is mutator specific will only be used within the mutator functions. All other components will treat these items as opaque strings/blobs.

`void init(mutator_t * m)`

This function fills in m with all of the function pointers for this mutator. Note: This function only appears when compiled as a module. When ALL_MUTATORS_IN_ONE is defined, this function will not exist, as there

would be a name collision with all the other init() functions from other modules and there will not be any need for obtaining this struct, as all the functions will just be called directly. It's just the code which uses modules which will want to use this struct. ALL_MUTATORS_IN_ONE being defined will cause all the other functions to have the name of the mutator and an underscore prepended. This means that the create() function will be called bit_flip_create() in the bit flipper mutator. The name of the mutator is defined by MUTATOR_NAME.

- m - a pointer to a mutator_t structure that will be filled in with the function pointers that define this mutator.
- return value - none

`void * create(char * options, char * state, char * input, size_t input_length)`

This function will allocate and initialize the mutator structure. The allocated structure will exist until the cleanup() function is called.

- options - a JSON string that contains the mutator specific string of options.
- state - used to load a previously dumped state (produced by the get_state() function), that defines the current iteration of the mutator. This will be a mutator specific JSON string. Alternatively, NULL can be provided to start a mutator without a previously dumped state.
- input - base input string which will be modified to produce mutated inputs later when the mutate() function is called
- input_length - the size of the input buffer
- return value - a mutator specific structure or NULL on failure. The returned value should not be used for anything other than passing to the various Mutator API functions.

`void cleanup(void * mutator_state)`

This function will release any resources that the mutator has open and free the mutator state structure.

- mutator_state - a mutator specific structure previously created by the create() function. This structure will be freed and should not be referenced afterwards.

`int mutate(void * mutator_state, char * buffer, size_t buffer_length)`

This function will mutate the input given in the create() function and return it in the buffer argument. The size of the buffer will be mutator specific. For example, some mutators may require this buffer to be larger than the original input (passed to the create() function) as it's going to extend the original input in some way. Other mutators will want it to be the same size. Guidance on this will be specified by the mutator specific documentation.

- mutator_state - a mutator specific structure previously created by the create() function.
- buffer - a buffer to which the mutated input will be written
- buffer_length - the size of the passed in buffer argument
- return value - the length of the mutated data on success, 0 when the mutator is out of mutations, or -1 on error

```
int mutate_extended(void * mutator_state, char * buffer, size_t
buffer_length, uint64_t flags);
```

This function is identical to the `mutate` function, with the exception that it accepts a flags parameter that specifies how the mutations should be done.

- mutator_state - a mutator specific structure previously created by the create() function.
- buffer - a buffer to which the mutated input will be written
- buffer_length - the size of the passed in buffer argument
- flags - this parameter is a bitfield that specifies how the mutations should be done. Available flags are:
  - `MUTATE_THREAD_SAFE` - The mutator should ensure that the mutations are done in a thread safe way. If the mutator will be accessed via multiple concurrent threads, this flag should be set.
  - `MUTATE_MULTIPLE_INPUTS` - If the mutator will be handling individual input parts, this flag should be used. For some of the fuzzer applications, it may be necessary to split the input up into separate pieces that are mutated independently. In these cases, the mutator can be given multiple inputs and asked for mutations of the input parts individually. One example user of this API is the network driver, where each input is a separate network packet sent to the target process. When this flag is set, the index of the input part to mutate should be included in the lowest 16-bits of the flags parameter. For instance, to mutate the fifth input buffer, set flags to `(MUTATE_MULTIPLE_INPUTS | 5)`.
- return value - the length of the mutated data on success, 0 when the mutator is out of mutations, or -1 on error

`char * get_state(void * mutator_state)`

This function will return the state of the mutator. The returned value can be used to restart the mutator at a later time, by passing it to the create() or set_state() function. It is the caller's responsibility to free the memory allocated here using the free_state() function.

- mutator_state - a mutator specific structure previously created by the create() function.
- return value - a buffer that defines the current state of the mutator. This will be a mutator specific JSON string.

`void free_state(char * state)`

This function will free a previously dumped state (via the get_state() function) of the mutator.

- state - a previously dumped state buffer obtained by the get_state() function.

`int set_state(void * mutator_state, char * state)`

This function will set the current state of the mutator. This can be used to restart a mutator once from a previous run.

- mutator_state - a mutator specific structure previously created by the create() function.
- state - a previously dumped state buffer obtained by the get_state() function. This will be a mutator specific JSON string.
- return value - 0 on success or non-zero on failure

`int get_current_iteration(void * mutator_state)`

This function will return the current iteration count of the mutator, i.e. how many mutations have been generated with it.

- mutator_state - a mutator specific structure previously created by the create() function.
- return value - the number of previously generated mutations

`int get_total_iteration_count(void * mutator_state)`

This function will return the total possible number of mutations with this mutator. For some mutators, this value won't be possible to predict or the mutator will be capable of an infinite number of mutations.

- mutator_state - a mutator specific structure previously created by the create() function.
- return value - the number of possible mutations with this mutator. If this number can't be predicted or is infinite, -1 will be returned.

`void get_input_info(void * mutator_state, int * num_inputs, size_t **input_sizes)`

This function will retrieve the number of inputs and the size of each input that is managed by a mutator. For most of the simple mutators, they will only be given a single input. However, some of the more complicated mutators, such as the manager mutator, will manage several input buffers and mutate them independently with the `mutate_extended` function. This function will return the number of inputs that a mutator is mutating and the sizes of each of those inputs.

- mutator_state - a mutator specific structure previously created by the create() function.
- num_inputs - a pointer to an integer which will be used to return the number of inputs that a mutator has.
- input_sizes - a pointer to a size_t array that will be used to return the size of each of the inputs.

`int set_input(void * mutator_state, char * new_input, size_t input_length)`

This function will set the input (saved in the mutator's state) to something new. This can be used to reinitialize a mutator with new data, without reallocating the entire state struct.

- mutator_state - a mutator specific structure previously created by the create() function.
- new_input - The new input used to produce new mutated inputs later when the mutate() function is called
- input_length - the size in bytes of the input buffer.
- return value - 0 on success and -1 on failure

`int help(char ** help_str)`

This function sets a help message for the mutator. This is useful if the mutator takes a JSON options string in the create() function.

- help_str - A double pointer that will be updated to point to the new help string.
- return value - 0 on success and -1 on failure

5 Driver

The driver will be the component that runs the program being fuzzed. The driver should start the program, feed in the input, and determine when the program is done processing the input. This component may need to be customized per target application.

Anything which is driver specific will only be used within the driver functions. All other components will treat these items as opaque strings/blobs.

```
void * create(char * options, instrumentation_t * instrumentation,
void * instrumentation_state, mutator_t * mutator, void * mutator_state)
```

This function will allocate and initialize the driver structures. If the driver is going to be testing a long-running process, this function should start that process. Anything that needs to be done before a fuzzing run can start should be done here.

- options - a JSON string that contains the driver specific string of options.
- instrumentation - a pointer to an instrumentation instance that the driver will use to instrument the requested program. The caller should initialize this instrumentation instance before the create call to the driver, and then free it after cleaning up the driver. This parameter is optional and can be set to NULL if the caller does not wish to use an instrumentation with the driver.
- instrumentation_state - a pointer to the instrumentation state for the passed in instrumentation. This parameter is optional and can be set to NULL if the caller does not wish to use an instrumentation with the driver.
- mutator - a pointer to a mutator instance that the driver can use to obtain the next input (for use in the `test_next_input` function). This parameter is optional and can be set to NULL if the caller does not wish to use a mutator with the driver. Without this parameter, the `test_next_input` and `get_last_input` functions will be unavailable.
- mutator_state - a pointer to the mutator state for the passed in mutator. This parameter is optional and can be set to NULL if the caller does not wish to use a mutator with the driver.
- return value - a driver specific structure or NULL on failure. The returned value should not be used for anything other than passing to the various Driver API functions.

```
void cleanup(void * driver_state)
```

This function will kill any processes created by the driver and clean up anything else that was created to help fuzzing. It will also free the driver state.

- driver_state - a driver specific structure previously created by the create function. This structure will be freed and should not be referenced afterwards.

```
int test_input(void * driver_state, char * buffer, size_t length)
```

This function will cause the program being fuzzed to be tested against the given input. This function should block execution until the program being fuzzed has finished processing the given input.

- driver_state - a driver specific structure previously created by the create function.
- buffer - the input that should be tested
- length - the length of the buffer argument
- return value - 0 on success, -1 on failure

```
int test_next_input(void * driver_state);
```

This function uses the mutator given during the driver creation to retrieve the next mutated input and test it against the target program. This function blocks execution until the program being fuzzed has finished processing the mutated input. This function is only available if a mutator was given to the driver in the `create` function.

- driver_state - a driver specific structure previously created by the create function.
- return value - 0 on success, -1 on failure, or -2 if the mutator has run out of inputs to to mutate.

```
void * get_last_input(void * driver_state, int * length);
```

This function retrieves the most recent mutated input that was tested with the `test_next_input` function. This function is only available if a mutator was given to the driver in the `create` function.

- driver_state - a driver specific structure previously created by the create function.
- length - a pointer to an integer that will be set to the length of the returned buffer.
- return value - on success this function will return a buffer containing the last input that was tested, or NULL on failure. This pointer should not be freed by the caller, and is only valid until the next call to `test_next_input`.

6 Instrumentation

The instrumentation modules are what track the state of a process and determine if a path through the process is new. This will include things such as QEMU (for Linux), LLVM (for source), PIN, Dynamo-RIO, Dyninst, and Intel PT. They are optionally given some state information. The state information is module-specific and is used to tell the instrumentation module which paths have been previously hit. Additionally, each instrumentation module will have a variety of configuration options that can be specified that will be specific to that instrumentation module. These options will be specified as a JSON char array.

Anything which is instrumentation specific will only be used within the instrumentation functions. All other components will treat these items as opaque strings/blobs.

`void * create(char * options, char * state)`

This function will create and return an instrumentation struct that defines the instrumentation's state. The state argument will be used to load the previously executed paths through the fuzzed program.

- options - a JSON string that contains the instrumentation specific options
- state - used to load a previously dumped state (produced by the get_state() function), that defines the current paths seen by the instrumentation. Alternatively, NULL can be provided to start an instrumentation without a previously dumped state
- return value - an instrumentation specific structure or NULL on failure. The returned value should not be used for anything other than passing to the various Instrumentation API functions

`void cleanup(void * instrumentation_state)`

This function will release any resources that the instrumentation has open and free the instrumentation state.

- instrumentation_state - an instrumentation specific structure previously created by the create function. This structure will be freed and should not be referenced afterwards

`char * get_state(void * instrumentation_state, int *out_length)`

This function will return the state information holding the previous execution path info. The returned value can later be passed to the instrumentation create() function to load the state back into an instrumentation struct. It is the caller's responsibility to free the memory allocated and returned here using the free_state() function.

- instrumentation_state - an instrumentation specific structure previously created by the create function
- out_length - this pointer will be filled with the length of the returned state buffer
- return value - a buffer that holds information about the previous execution paths as a JSON char array.

`void free_state(char * state)`

This function will free a previously dumped state (via the get_state() function) of the instrumentation.

- state - a previously dumped state buffer obtained by the get_state() function

`int set_state(void * instrumentation_state, char * state)`

This function will set the previous execution paths of the instrumentation. This can be used to restart an instrumentation once it has been created.

- instrumentation_state - an instrumentation specific structure previously created by the create() function
- state - a previously dumped state buffer obtained by the get_state() function
- return value - 0 on success or non-zero on failure

`void * merge(void * instrumentation_state, void * other_instrumentation_state)`

This function will merge two sets of instrumentation coverage data. The resulting instrumentation state will include the tracked coverage from both instrumentation states. Both instrumentation states must have the same instrumentation options (what to track coverage of, which modules, etc.) specified, and generally need to be produced by the same instrumentation module in order for the merge to work correctly. It's possible that two different instrumentation modules may produce state information in the same format, however this is up to them and not something guaranteed by this specification. Neither argument will be modified nor freed. It is the caller's responsibility to free the memory allocated and returned here using the free_state() function.

- instrumentation_state - an instrumentation specific structure previously created by the create() function
- other_instrumentation_state - a second instrumentation specific structure previously created by the create() function that should be merged with the first

- return value - an instrumentation specific structure that combines the coverage information from both of the instrumentation states or NULL on failure

`int enable(void * instrumentation_state, HANDLE * process, char * cmd_line, char * input, size_t input_length)`

This function will enable the instrumentation module for a specific process and runs that process. If the process needs to be restarted, it will be.

- instrumentation_state - an instrumentation specific structure previously created by the create() function
- process - a pointer to a handle for the process on which the instrumentation was enabled
- cmd_line - the command line of the fuzzed process on which to enable instrumentation
- input - pointer to the buffer containing the input data that should be sent to the fuzzed process
- input_length - the length of the input parameter
- return value - 0 on success, non-zero on failure

`int is_new_path(void * instrumentation_state, int * process_status)`

This function will determine whether the process being instrumented has taken a new path. It should be called after the process has finished processing the tested input.

- instrumentation_state - an instrumentation specific structure previously created by the create() function
- process_status - pointer that will be filled with a value representing whether the fuzzed process crashed or hung, or neither
- return value - 1 if the previously setup process (via the enable() function) took a new path, 0 if it did not, or -1 on failure

`int get_module_info(void * instrumentation_state, int index, int * is_new, char ** module_name, char **info, int size)`

This function is optional and not required for the fuzzer to work. It can be used to obtain coverage information for each executable/library separately. This function returns information about each of the separate modules (shared libraries such as .dll, .so, .dynlib).

- instrumentation_state - an instrumentation specific structure previously created by the create() function
- index - an index into the module list for the module about which information should be retrieved. The return value will indicate if a module exists for this index. Indices start at 0 and increase from there
- is_new - This parameter returns whether or not the last run of the instrumentation returned a new path for the module with the specified index. In order for the information returned in this parameter to be accurate, the is_new_path method should be called first. This parameter is optional and can be set to NULL, if you do not want this information
- module_name - This parameter returns the filename of the module at the specified index. This parameter is optional and can be set to NULL, if you do not want this information. This parameter should not be freed by the caller
- info - This parameter returns the per-instrumentation path info for the module with the specified index. For example, for the DynamoRIO module, the returned info is an AFL style bitmap of the edges. This parameter is optional and can be set to NULL, if you do not want this information. This parameter should not be freed by the caller
- size - This parameter returns the size of the per-instrumentation path info in the returned info parameter. This parameter is optional and can be set to NULL, if you do not want this information
- return value - non-zero if module with the specified index cannot be found, or 0 if it is found

`instrumentation_edges_t * get_edges(void * instrumentation_state, int index)`

This function is optional and not required for the fuzzer to work. It is used by the tracer. This function returns an array of basic block edges that occurred in the most recent run of the instrumentation.

- instrumentation_state - an instrumentation specific structure previously created by the create() function.
- index - If per-module instrumentation information is enabled, this parameter is an index into the module list for the module about which edges should be retrieved. The return value will indicate if a module exists for this index. Indices start at 0 and increase from there. If per-module instrumentation information is NOT enabled, then this parameter is ignored and the general edges array will be returned.

- return value - NULL if an array of basic block edges was not tracked for the most recent instrumentation run or per-module instrumentation is enabled and the requested index was not found. Otherwise, an instrumentation_edges_t pointer that contains an array of basic block edges that were hit in the most recent instrumentation run. The returned pointer should be freed by the caller.

7 Structures

This section describes the structures used throughout the API. For each of the top level components, there is a structure which defines the available functions in that component. This allows for a common interface among all of the available implementations of a component.

```
typedef struct mutator
{
  void * (*create)(char * options, char * state, char * input,
    size_t input_length);
  void(*cleanup)(void * mutator_state);

  int(*mutate)(void * mutator_state, char * buffer,
    size_t buffer_length);
  int(*mutate_extended)(void * mutator_state, char * buffer,
    size_t buffer_length, uint64_t flags);

  char * (*get_state)(void * mutator_state);
  void(*free_state)(char * state);
  int(*set_state)(void * mutator_state, char * state);

  int(*get_current_iteration)(void * mutator_state);
  int(*get_total_iteration_count)(void * mutator_state);
  void(*get_input_info)(void * mutator_state, int * num_inputs,
    size_t **input_sizes);

  int(*set_input)(void * mutator_state, char * new_input,
    size_t input_length);
  int(*help)(char **help_str);
} mutator_t;
```

Listing 1: `mutator_t` struct definition

The `mutator_t` structure, shown in Listing 1, defines all of the common interfaces for each mutator. The definitions of each of the function pointers in the `mutator_t` structure is described in Section 4.

```
struct driver
{
  void (*cleanup)(void * driver_state);
  int (*test_input)(void * driver_state, char * buffer, size_t length);
  int(*test_next_input)(void * driver_state);
  void *(*get_last_input)(void * driver_state, int * length);
  void * state;
};
typedef struct driver driver_t;
```

Listing 2: `driver_t` struct definition

The `driver_t` structure, shown in Listing 2, defines all of the common interfaces for each driver. The definitions of each of the function pointers in the `driver_t` structure is described in Section 5. The last field in the struct, `state` holds a pointer to the implementation specific driver state structure. This field should only be used when calling each of the driver's functions.

```
struct instrumentation
{
  void *(*create)(char * options, char * state);
  void(*cleanup)(void * instrumentation_state);
  void *(*merge)(void * instrumentation_state,
    void * other_instrumentation_state);
  char * (*get_state)(void * instrumentation_state);
  void(*free_state)(char * state);
  int(*set_state)(void * instrumentation_state, char * state);
  int(*enable)(void * instrumentation_state, HANDLE * process,
  char * cmd_line, char * input, size_t input_length);
  int(*is_new_path)(void * instrumentation_state, int * process_status);

  //Optional
  int (*get_module_info)(void * instrumentation_state, int index,
    int * is_new, char ** module_name, char ** info, int * size);
  instrumentation_edges_t * (*get_edges)(void * instrumentation_state,
    int index);
};
typedef struct instrumentation instrumentation_t;
```

Listing 3: `instrumentation_t` struct definition

The `instrumentation_t` structure, shown in Listing 3, defines all of the common interfaces for each instrumentation. The definitions of each of the function pointers in the `instrumentation_t` structure is described in Section 6.

```
struct instrumentation_edge
{
#ifdef _M_X64
  uint64_t from;
  uint64_t to;
#else
  uint32_t from;
  uint32_t to;
#endif
};
typedef struct instrumentation_edge instrumentation_edge_t;
```

Listing 4: `instrumentation_edge_t` struct definition

The `instrumentation_edge_t` structure, shown in Listing 4, is used to return a list of basic block addresses which make up each edge in the fuzzed program's path. This list is returned from the `get_edges` method as described in Section 6.