Bytecode Signature through Example

Here is a sample bytecode signature which uses a simple logical trigger and scans the ClamAV internal preclassification for an embedded (contained) MSEXE object:

```
/* ClamAV.BCC.SandBox.Submit */
/* ClamAV.BCC.SandBox.InActive */
VIRUSNAME_PREFIX("ClamAV.BCC.SandBox")
VIRUSNAMES("Submit", "InActive")

/* Target type is 13, internal JSON properties */
TARGET(13)

/* JSON API call will require FUNC_LEVEL_098_5 = 78 */
FUNCTIONALITY_LEVEL_MIN(FUNC_LEVEL_098_5)

SIGNATURES_DECL_BEGIN
DECLARE_SIGNATURE(sig1)
SIGNATURES_DECL_END

SIGNATURES_DEF_BEGIN
/* search @offset 0 : '{ "Magic": "JSON",' */
/* this can be readjusted for specific filetypes */
DEFINE_SIGNATURE(sig1, "0:7b20224d61676963223a20224a534f4e222c")
SIGNATURES_END

bool logical_trigger(void)
{
    return matches(Signatures.sig1);
}

#define STR_MAXLEN 256

int entrypoint ()
{
    int i;
    int32_t type, obj, objarr, objit, arrlen, strlen;
    char str[STR_MAXLEN];

    /* check is json is available, alerts on inactive (optional) */
    if (!json_is_active())
        foundVirus("InActive");

    /* acquire array of internal contained objects */
    objarr = json_get_object("ContainedObjects", 16, 0);
    type = json_get_type(objarr);
    /* debug print uint (no '\n' or prepended message */
    debug_print_uint(type);

    if (type != JSON_TYPE_ARRAY) {
        return -1;
    }

    /* check array length for iteration over elements */
    arrlen = json_get_array_length(objarr);
    for (i = 0; i < arrlen; ++i) {
        /* acquire json object @ idx i */
```

```
        objit = json_get_array_idx(i, objarr);
        if (objit <= 0) continue;

        /* acquire FileType object of the array element @ idx i */
        obj = json_get_object("FileType", 8, objit);
        if (obj <= 0) continue;

        /* acquire and check type */
        type = json_get_type(obj);
        if (type == JSON_TYPE_STRING) {
            /* acquire string length, note +1 is for the NULL terminator */
            strlen = json_get_string_length(obj)+1;
            /* prevent buffer overflow */
            if (strlen > STR_MAXLEN)
                strlen = STR_MAXLEN;
            /* acquire string data, note strlen includes NULL terminator */
            if (json_get_string(str, strlen, obj)) {
                /* debug print str (with '\n' and prepended message */
                debug_print_str(str,strlen);

                /* check the contained object's type */
                if (strlen == 14 && !memcmp(str, "CL_TYPE_MSEXE", 14)) {
                    /* alert for submission */
                    foundVirus("Submit");
                    return 0;
                }
            }
        }
    }

    return 0;
}
```

Example 1: sample bytecode signature for an embedded (contained) MSEXE object

---

Reported Signatures

The signatures that a bytecode can reported are determined by the strings passed to the VIRUSNAME_PREFIX and VIRUSNAMES macros.

```
/* ClamAV.BCC.SandBox.Submit */
/* ClamAV.BCC.SandBox.InActive */
VIRUSNAME_PREFIX("ClamAV.BCC.SandBox")
VIRUSNAMES("Submit", "InActive")
```

VIRUSNAME_PREFIX: a ***REQUIRED*** macro field. It consists of exactly one string value which may contain alphanumeric characters and periods; periods are used to mark different groups the signature is attributed to.

VIRUSNAMES: an ***OPTIONAL*** macro field. It consists of an array of string values which may only contain alphanumeric characters.

The bytecode signature reports specific detections through the usage of the bytecode API function foundVirus() which takes a single string argument that correlates to the VIRUSNAMES.

```
        foundVirus("InActive");
              ...
                    foundVirus("Submit");
```

Using an empty string (“”) will have the bytecode simple report the VIRUSNAME_PREFIX. Note that the VIRUSNAME_PREFIX string is not part of the foundVirus() call. Bytecode signatures may report one detection; multiple calls to foundVirus() may overwrite the previous detection though this behavior is not guaranteed.

Target Group and Engine

Bytecode allows for the user to direct the application of a bytecode signature specifically at a particular filetype and for a specific version of the ClamAV engine.

```
/* Target type is 13, internal JSON properties */
TARGET(13)

/* JSON API call will require FUNC_LEVEL_098_5 = 78 */
FUNCTIONALITY_LEVEL_MIN(FUNC_LEVEL_098_5)
```

TARGET: a ***normally OPTIONAL*** macro field. For the case of ClamAV internal preclassification, this is ***REQUIRED*** and **MUST BE set to 13.** It consists of single integer value [1-13 at time of writing] which represents the intended target of the bytecode (bytecode will only run on that target type). Target types are listed in the ClamAV Signature document.

FUNCTIONALITY_LEVEL_MIN: a ***normally OPTIONAL*** macro field. For the case of ClamAV internal preclassification, this is ***REQUIRED*** and **MUST BE set to at least FUNC_LEVEL_098_5.** It consists of an enumeration value that represents the minimum functionality level of ClamAV for this bytecode to run on. ClamAV versions prior to this value will not load this bytecode.

FUNCTIONALITY_LEVEL_MAX: an ***OPTIONAL*** macro field. It consists of an enumeration value that represents the maximum functionality level of ClamAV for this bytecode to run on. ClamAV versions after this value will not load this bytecode.

Patterns

This section contains associated subsignatures and the logical trigger function. These are ***REQUIRED***.

```
SIGNATURES_DECL_BEGIN
DECLARE_SIGNATURE(sig1)
SIGNATURES_DECL_END

SIGNATURES_DEF_BEGIN
/* search @offset 0 : '{ "Magic": "JSON",' */
/* this can be readjusted for specific filetypes */
DEFINE_SIGNATURE(sig1, "0:7b20224d61676963223a20224a534f4e222c")
SIGNATURES_END
```

```
bool logical_trigger(void)
{
    return matches(Signatures.sig1);
}
```

Excerpt from clambc-user.pdf (from ClamAV Bytecode Documentation):
"Logical signatures use .ndb style patterns….
Each pattern has a name (like a variable), and a string that is the hex pattern itself. The declarations are delimited by the macros SIGNATURES_DECL_BEGIN, and SIGNATURES_DECL_END. The definitions are delimited by the macros SIGNATURES_DEF_BEGIN, and SIGNATURES_END. Declarations must always come before definitions, and you can have only one declaration and declaration section! (think of declaration like variable declarations, and definitions as variable assignments, since that what they are under the hood). The order in which you declare the signatures is the order in which they appear in the generated logical signature.
You can use any name for the patterns that is a valid record field name in C, and doesn't conflict with anything else declared.
After using the above macros, the global variable Signatures will have [one field, sig1]. [This] can be used as arguments to the functions count_match(), and matches() anywhere in the program…:

- matches[Signatures.sig1) will return true when the [sig1] signature matches (at least once)
- count_match(Signatures.sig1) will return the number of times the [sig1] signature matched

The condition in [the logical_trigger function can be interpreted as if sig1 is matched at least once]."

The logical trigger for the sample bytecode is set to trigger on all ClamAV preclassifications by triggering on the detection of the associated preclassification "file magic". Note that specifying TARGET(13) will already force the signature to apply strictly to ClamAV preclassification.

Main Program (entrypoint)
The "entrypoint" function in the bytecode signature can be seen as effectively the "main" function within a C program. It is ***REQUIRED*** and must use this prototype.

```
int entrypoint ()
{
        ...
}
```

At this point, the bytecode uses roughly the same syntax as the C programming language to perform operations on the file to determine whether or not to report a detection. There are a key number of limitations on the bytecode language from C however, all of which can be found in section 4 of the clambc-user.pdf document of the ClamAV Bytecode Compiler.

Some notable limitations:

- at most 64-bit integers and must be fixed-size
- no floating point numbers

- globals must be read-only constant and compile-time computable
- no struct or pointer returns
- no variable-length arrays
- no inline assembly
- pointers cannot be casted to integers and vice-versa
- function calls using pointer argument must contain the size of the object the pointer points to as the immediately following argument
- attempting to access out-of-bound locations will result in bytecode termination with an abort
- no external include files (such as the cstdlib), the bytecode api is preincluded
- all code must be contain in a single source file
- while not strictly a limitation, it is a good practice to declare all variables at the start of the function

---

Bytecode API

The bytecode API is a series of functions that are provided to users automatically in all bytecode signatures. Note that all external function calls can only come from the bytecode API. For a full list of all bytecode API, consult the clambc-api document in the bytecode compiler's documentation.

The normal JSON API uses pointer values, however, the bytecode API uses object ID values to reference specific JSON nodes. Object IDs are retrieved through calls of json_get_object with the topmost node predefined to object ID 0. Note that this predefinition allows for certain functions to return the value 0 as an error value even though 0 is "valid".

Here is a list of JSON-specific bytecode API functions:

```
Determines if libclamav is configured with JSON.
    @return 0 - json is disabled or option not specified
    @return 1 - json is active and properties are available
int32_t json_is_active(void);
```

```
Retrieves the ID value of the specified named object within the specified parent object
ID (object ID 0 is guaranteed to be defined as the topmost object).
    @return objid of json object with specified name
    @return 0 if json object of specified name cannot be found
    @return -1 if an error has occurred
    @param[in] name - name of object in ASCII
    @param[in] name_len - length of specified name (not including terminating NULL),
                          must be >= 0
    @param[in] objid - id value of json object to query
int32_t json_get_object(const int8_t* name, int32_t name_len, int32_t objid);
```

Determines the type of the specified JSON object, value is of the enumeration bc_json_type {JSON_TYPE_NULL=0, JSON_TYPE_BOOLEAN, JSON_TYPE_DOUBLE, JSON_TYPE_INT, JSON_TYPE_OBJECT, JSON_TYPE_ARRAY, JSON_TYPE_STRING}.

```
    @return type (json_type) of json object specified
    @return -1 if type unknown or invalid id
    @param[in] objid - id value of json object to query
int32_t json_get_type(int32_t objid);
```

Determines the length of the JSON array object; objid must be of type JSON_TYPE_ARRAY or an error will be returned.

```
    @return number of elements in the json array of objid
    @return -1 if an error has occurred
    @return -2 if object is not JSON_TYPE_ARRAY
    @param[in] objid - id value of json object (should be JSON_TYPE_ARRAY) to query
int32_t json_get_array_length(int32_t objid);
```

Retrieves the ID value for the object located at a specific index of a JSON array object; objid must be of type JSON_TYPE_ARRAY or an error will be returned.

```
    @return objid of json object at idx of json array of objid
    @return 0 if invalid idx
    @return -1 if an error has occurred
    @return -2 if object is not JSON_TYPE_ARRAY
    @param[in] idx - index of array to query, must be >= 0 and less than array length
    @param[in] objid - id value of json object (should be JSON_TYPE_ARRAY) to query
int32_t json_get_array_idx(int32_t idx, int32_t objid);
```

Determines the length of a JSON string object; objid must be of type JSON_TYPE_STRING or an error will be returned. Note: this value DOES NOT include the terminating null.

```
    @return length of json string of objid, not including terminating null-character
    @return -1 if an error has occurred
    @return -2 if object is not JSON_TYPE_STRING
    @param[in] objid - id value of json object (should be JSON_TYPE_STRING) to query
int32_t json_get_string_length(int32_t objid);
```

Retrieves the string contents of the JSON string object and stores it to a user location; objid must be of type JSON_TYPE_STRING or an error will be returned. Note: the specified length MUST include the terminating NULL.

```
    @return number of characters transferred (capped by str_len),
            including terminating null-character
    @return -1 if an error has occurred
    @return -2 if object is not JSON_TYPE_STRING
    @param[out] str - user location to store string data; will be null-terminated
    @param[in] str_len - length of str or limit of string data to read,
                         including terminating null-character
    @param[in] objid - id value of json object (should be JSON_TYPE_STRING) to query
int32_t json_get_string(int8_t* str, int32_t str_len, int32_t objid);
```

Retrieves the boolean value of a JSON object.

```
    @return boolean value of queried objid; will force other types to boolean
    @param[in] objid - id value of json object to query
int32_t json_get_boolean(int32_t objid);
```

Retrieves the integer value of a JSON object.

```
    @return integer value of queried objid; will force other types to integer
    @param[in] objid - id value of json object to query
int32_t json_get_int(int32_t objid);
```

Note that since bytecode does not support double type, it is not possible to retrieve double values from JSON objects.

Example Entrypoint Walkthrough

```
    int i;
    int32_t type, obj, objarr, objit, arrlen, strlen;
    char str[STR_MAXLEN];
```

These are the declarations of the variables will use in the bytecode. While it is not strictly enforced, it is generally good practice to state all variables at the start of each function. Note that only basic C types (excluding floats and doubles) and fixed-sized ints can be used. `STR_MAXLEN` is a macro equal to 256.

```
    /* check is json is available, alerts on inactive (optional) */
    if (!json_is_active())
        foundVirus("InActive");
```

The bytecode API is used to query if JSON is enabled in the libclamav instance. Note that the target type requirement of 13 will generally force the returned value to be true; this statement is here for extra safety. This segment also reports a virus “InActive” in the case that the JSON is inactive. A call to “foundVirus()” does not terminate the run of the program, so this sample signature will actually continue running even though JSON is not available. This is alright as most JSON parsing API functions check if JSON is enabled and return an error value.

```
    /* acquire array of internal contained objects */
    objarr = json_get_object("ContainedObjects", 16, 0);
    type = json_get_type(objarr);
    /* debug print uint (no '\n' or prepended message */
    debug_print_uint(type);

    if (type != JSON_TYPE_ARRAY) {
        return -1;
    }
```

This segment acquires an object ID for the “ContainedObjects” object and checks to see if the object is typed JSON_TYPE_ARRAY. The other call “debug_print_uint()” prints a debug message with only the uint value (no “LibClamAV Debug” or newline”).

```
    /* check array length for iteration over elements */
    arrlen = json_get_array_length(objarr);
    for (i = 0; i < arrlen; ++i) {
        /* acquire json object @ idx i */
        objit = json_get_array_idx(i, objarr);
        if (objit <= 0) continue;
```

This segment setups an iteration across all the members of the “ContainedObjects” array retrieved earlier. Note the check for the objit ID to be a valid value.

```
        /* acquire FileType object of the array element @ idx i */
        obj = json_get_object("FileType", 8, objit);
        if (obj <= 0) continue;

        /* acquire and check type */
        type = json_get_type(obj);
        if (type == JSON_TYPE_STRING) {
            /* acquire string length, note +1 is for the NULL terminator */
            strlen = json_get_string_length(obj)+1;
            /* prevent buffer overflow */
            if (strlen > STR_MAXLEN)
                strlen = STR_MAXLEN;
            /* acquire string data, note strlen includes NULL terminator */
            if (json_get_string(str, strlen, obj)) {
                /* debug print str (with '\n' and prepended message */
                debug_print_str(str,strlen);

                /* check the contained object's type */
                if (strlen == 14 && !memcmp(str, "CL_TYPE_MSEXE", 14)) {
                    /* alert for submission */
                    foundVirus("Submit");
                    return 0;
                }
            }
        }
    }
```

This segment retrieves the objit’s type and, if it is a string, retrieves the string value and stores it in the str user string. The returned user string is a copy and modifications to the string do not change the internal object’s value. Next the string is a comparison of the returned string against “CL_TYPE_MSEXE” to determine whether to “Submit”. Effectively, this signature returns a “Submit” whenever it detects a PE file embedded within the parent file.

Note the checks on the received strlen value to prevent a buffer overflow vulnerability. Bytecode signatures are always compiled with various runtime checks and thus a case that would cause the vulnerability would trigger a runtime error and bytecode termination (clamav continues to run). Regardless, great care should be exercised in regards to user variable boundaries.

```
    return 0;
}
```

Entrypoint function needs to return an integer; returning a 0 reports no issues with the bytecode execution however, all return values are ignored by libclamav by default.