Clamav funcation call flow(bytecode scan JIT)

bytecode signature scan JIT mode explained

by eqmcc

Description

this document will talk about bytecode signature scan

Data structures

cl_engine // in this structure, following are used for bytecode

```
/* Used for bytecode */
struct cli_all_bc bcs;
unsigned *hooks[_BC_LAST_HOOK - _BC_START_HOOKS];
unsigned hooks_cnt[_BC_LAST_HOOK - _BC_START_HOOKS];
unsigned hook_lsig_ids;
enum bytecode_security bytecode_security;
uint32_t bytecode_timeout;
enum bytecode_mode bytecode_mode;
```

cli_all_bc // for bytecode config

```
struct cli_all_bc {
    struct cli_bc *all_bcs;
    unsigned count;
    struct cli_bcengine *engine;
    struct cli_environment env;
    int    inited;
};
```

cli_bc //detailed bytecode info

```
struct cli_bc {
  struct bytecode_metadata metadata;
  unsigned id;
  unsigned kind;
  unsigned num_types;
  unsigned num_func;
  struct cli_bc_func *funcs;
  struct cli_bc_type *types;
  uint64_t **globals;
  uint16_t *globaltys;
  size_t num_globals;
  enum bc_state state;
  struct bitset_tag *uses_apis;
  char *lsig;
  char *vnameprefix;
  char **vnames;
  unsigned vnames_cnt;
  uint16_t start_tid;
  struct cli_bc_dbgnode *dbgnodes;
  unsigned dbgnode_cnt;
  unsigned hook_lsig_id;
  unsigned trusted;
  uint32_t numGlobalBytes;
  uint8_t *globalBytes;
};
```

bytecode_metadata

```
struct bytecode_metadata {
    char *compiler;
    char *sigmaker;
    uint64_t timestamp;
    unsigned formatlevel;
    unsigned minfunc, maxfunc;
    unsigned maxresource;/* reserved */
    unsigned targetExclude;
};
```

cli_bcengine // bytecode engine config

```
struct cli_bcengine {
    ExecutionEngine *EE;
    JITEventListener *Listener;
    LLVMContext Context;
    FunctionMapTy compiledFunctions;
    union {
    unsigned char b[16];
    void* align;/* just to align field to ptr */
    } guard;
};
```

cli_environment // execute enviroment

```
struct cli_environment {
    uint32_t platform_id_a;
    uint32_t platform_id_b;
    uint32_t platform_id_c;
    uint32_t c_version;
    uint32_t cpp_version; /* LLVM only */
    /* engine */
    uint32_t functionality_level;
    uint32_t dconf_level;
    int8_t   engine_version[65];
    /* detailed runtime info */
    int8_t    triple[65];/* LLVM only */
    int8_t    cpu[65];/* LLVM only */
    /* uname */
    int8_t    sysname[65];
    int8_t    release[65];
    int8_t    version[65];
    int8_t    machine[65];
    /* build time */
    uint8_t big_endian;
    uint8_t sizeof_ptr;
    uint8_t arch;
    uint8_t os_category;/* from configure */
    uint8_t os;/* from LLVM if available */
    uint8_t compiler;
    uint8_t has_jit_compiled;
    uint8_t os_features;
    uint8_t reserved0;
};
```

cli_bc_ctx

```
struct cli_bc_ctx {
    uint8_t timeout;/* must be first byte in struct! */
    uint16_t funcid;
    unsigned numParams;
    /* id and params of toplevel function called */
    const struct cli_bc *bc;
    const struct cli_bc_func *func;
    uint32_t bytecode_timeout;
    unsigned bytes;
    uint16_t *opsizes;
    char *values;
    operand_t *operands;
    uint32_t file_size;
    int outfd;
    off_t off;
    fmap_t *fmap;
    fmap_t *save_map;
    const char *virname;
    struct cli_bc_hooks hooks;
    struct cli_exe_info exeinfo;
    uint32_t lsigcnt[64];
    uint32_t lsigoff[64];
    uint32_t pdf_nobjs;
    struct pdf_obj *pdf_objs;
```

```
    uint32_t* pdf_flags;
    uint32_t pdf_size;
    uint32_t pdf_startoff;
    unsigned pdf_phase;
    int32_t pdf_dumpedid;
    const struct cli_exe_section *sections;
    uint32_t resaddr;
    char *tempfile;
    void *ctx;
    unsigned written;
    unsigned filewritten;
    unsigned found;
    unsigned ninflates;
    bc_dbg_callback_trace trace;
    bc_dbg_callback_trace_op trace_op;
    bc_dbg_callback_trace_val trace_val;
    bc_dbg_callback_trace_ptr trace_ptr;
    const char *directory;
    const char *file;
    const char *scope;
    unsigned trace_level;
    uint32_t scopeid;
    unsigned line;
    unsigned col;
    mpool_t *mpool;
    struct bc_inflate* inflates;
    struct bc_buffer *buffers;
    unsigned nbuffers;
    unsigned nhashsets;
    unsigned njsnorms;
    unsigned jsnormwritten;
    struct cli_hashset *hashsets;
    struct bc_jsnorm* jsnorms;
    char *jsnormdir;
    struct cli_map *maps;
    unsigned nmaps;
    unsigned containertype;
    unsigned extracted_file_input;
    const struct cli_environment *env;
    unsigned bytecode_disable_status;
    cli_events_t *bc_events;
    int on_jit;
    int no_diff;
};
```

Test case

1. source-code and bytecode

1.1 source code

test_bytecode.c

```
VIRUSNAME_PREFIX("test_bytecode")
VIRUSNAMES("A","B")
TARGET(7)
SIGNATURES_DECL_BEGIN
DECLARE_SIGNATURE(magic)
SIGNATURES_DECL_END

SIGNATURES_DEF_BEGIN
DEFINE_SIGNATURE(magic,"61616262")        // the pattern as "aabb" in hex
SIGNATURES_END
bool logical_trigger (void)
{
        // @ clamav-bytecode-compiler/obj/Release/lib/clang/1.1/include/bytecode_local.h
        return count_match(Signatures.magic) != 1; // if "aabb" match count is '1', it's not a virus
}

int entrypoint (void)
{
        int count = count_match(Signatures.magic);
        if ( count == 3)    foundVirus("B"); // 3 matches of "aabb", find virus B
        else foundVirus ("A"); // other case, find virus A
        return 0;
}
```

1.2 source code

1.2.1 compile

compile the source code to bytecode file *test_bytecode.cbc* via following command:

```
clambc-compiler test_bytecode.c -o test_bytecode.cbc -O2
```

and the *test_bytecode.cbc* looks as bellow:

```
ClamBCafhndcbn`ae|aeegcgefbg```c``a```|bjacflfafmfbfcfmb`cnbicgcnbccafmbecmbgfffecdfdfacdfcc``bcaaap`clamcoincidencejb:4096
```

```
test_bytecode.{A,B};Engine:56-255,Target:7;((0<1)|(0>1));61616262
Teddaaahdabahdacahdadahdaeahdafahdagahebbeebaeebcdebadaacb`bbadb`bdb`aahdb`db`b
Eaeaaaeb`e|amcgefdgfgifbgegcgnfafmfef``
Gd```hai`@`b`aC``a`baeBdgBefBcgBdgBoeBbfBigBdgBefBcfBofBdfBefBnbBad@`baeBdgBefBcgBdgBoeBbfBigBdgBefBcfBofBdfBefBnbBbd@`bcd@D```h`bad@A
b`bad@Ab`bad@Ac`bad@Ac`
A`b`bLadb`b`aa`b`b`b`b`Fagac
Bb`b`gbAd`aaaaeab`b`AcdTaaaaaaab
Bb`bababbaeAh`AodTcab`b@d
Bb`bacabbaeAf`AodTcab`b@dE
Sfeidbeeecendadmdedoe`ebeedfdidhehbbbdgefcgdgoebfigdgefcfofdfefbbibSfeidbeeecendadmdedcehbbbadbblbbbbdbbibSdeadbegdeddehbgcibSceidgdnda
ddeeebeedceoeddedcdldoebdedgdidnd
ddedcdldadbeedoeceidgdndaddeeebeedhbmfafgfifcfibSceidgdndaddeeebeedceoeddedcdldoeednddSSceidgdndaddeeebeedceoeddedfdoebdedgdidndSdde
dfdidndedoeceidgdndaddeeebeedhbmfafgfifcflbbbfcacfcacfcbcfcbcbbib`b`bobob`bdghfef`b`gafdgdgefbgnf`bafcg`bgbgbafafbfbfbb`bifnf`bhfefhg
ceidgdndaddeeebeedceoeedndddSbfofoflf`blfofgfifcfaflfoedgbgifgfgfefbg`bhbfgofifdfibSkgSobob`b`d`bcflfafmfaffgmbbfigdgefcfofdfefmbcfofmf`giflfefbgobofb
fjfobbeeflfefafcgefoblfifbfobcflfafnfgfobacnbacobifnfcflfegdfefobbfigdgefcfofdfefoelfofcfaflfnbhf
bgefdgegbgnf`bcfofegnfdgoemfafdgcfhfhbceifgfnfafdgegbgefcgnbmfafgfifcfib`babmc`backcSmgSSifnfdg`befnfdgbgig`gofifnfdg`bhbfgofifdfibSkgSifnfdg`bcfofeg
nfdg`bmc`bcfofegnfdgoemfafdgcfhfhbceifgfnfafdgegbgefcgnbmfafgfifcfibkc
ifff`bhb`bcfofegnfdg`bmcmc`bccib`b`bffofegnfdffeifbgegcghbbbbdbbibkc`bobob`bcc`bmfafdgcfhfefcg`bofff`bblciafafbfbfbldilb`bffifnfdf`bfgifbgegcg`bbdSeflfcg
ef`bffofegnfdffeifbgegcg`bhbbbadbbibkc`bobob`bofdghfefbg`bcfafcgeflb`bffifnfdf`bfgifbgegcg`bad
bgefdgegbgnf`b`ckcSmgSS
```

1.2.2 verify

verify the bytecode info via following command:

```
clambc --info test_bytecode.cbc
```

and output as bellow

```
Bytecode format functionality level: 6
Bytecode metadata:
        compiler version: clambc-0.97.3a-5-gf5dd1d3
        compiled on: (1359881038) Sun Feb  3 03:43:58 2013
        compiled by: user
        target exclude: 0
        bytecode type: logical only
        bytecode functionality level: 0 - 0
        bytecode logical signature: test_bytecode.{A,B};Engine:56-255,Target:7;((0<1)|(0>1));61616262
        // ((0<1)|(0>1))means the logic signature will be matched if thesub logic sig's match count is not 1
        virusname prefix: (null)
        virusnames: 0
        bytecode triggered on: files matching logical signature
        number of functions: 1
        number of types: 19
        number of global constants: 9
        number of debug nodes: 0
        bytecode APIs used:
```

```
setvirusname
```

2. run test

2.1 test files

test1.txt

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxaabb
```

test2.txt

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxaabbaabb
```

test3.txt

```
aabbxxxxxxxxxxxxxxxxxxxxaabbxxxxxxxxxxxxaabb
```

2.2 test run

```
clamscan --bytecode=yes --bytecode-unsigned=yes    -d test_bytecode.cbc test[1-3].txt
```

comments:
--bytecode=yes : enable bytecode scan
--bytecode-unsigned=yes : load unofficial bytecode

2.2.2 result

```
user@ubuntu:~/clamav$ clamscan --bytecode=yes --bytecode-unsigned=yes  -d test_bytecode.cbc test[1-3].txt
test1.txt: OK
test2.txt: test_bytecode.A FOUND
test3.txt: test_bytecode.B FOUND

----------- SCAN SUMMARY -----------
Known viruses: 1
Engine version: devel-a6558b5
Scanned directories: 0
Scanned files: 3
Infected files: 2
Data scanned: 0.00 MB
Data read: 0.00 MB (ratio 0.00:1)
Time: 0.031 sec (0 m 0 s)
user@ubuntu:~/clamav$
```

data structure initialization

scanmanager

```
================ scanmanager ================
    //--bytecode=yes : enable bytecode scan
     if(optget(opts,"bytecode")->enabled)
          dboptions |= CL_DB_BYTECODE;

    ret = cl_init(CL_INIT_DEFAULT)
          rc = bytecode_init();// empty function

    // init engine structure
    cl_engine_new
          new->bytecode_security = CL_BYTECODE_TRUST_SIGNED;
          /* 5 seconds timeout */
          new->bytecode_timeout = 60000;
          new->bytecode_mode = CL_BYTECODE_MODE_AUTO;
          // dynamic config
          cli_mpool_dconf_init
               if(!strcmp(modules[i].mname, "BYTECODE")) {
                         if (modules[i].state)
                         dconf->bytecode |= modules[i].bflag;
               }

    // --bytecode-unsigned=yes : load unofficial bytecode
    if(optget(opts, "bytecode-unsigned")->enabled)
    dboptions |= CL_DB_BYTECODE_UNSIGNED;
    // timeout
     if((opt = optget(opts,"bytecode-timeout"))->enabled)
     cl_engine_set_num(engine, CL_ENGINE_BYTECODE_TIMEOUT, opt->numarg);

    // set mode
enum bytecode_mode {
    CL_BYTECODE_MODE_AUTO=0, /* JIT if possible, fallback to interpreter */
    CL_BYTECODE_MODE_JIT, /* force JIT */
    CL_BYTECODE_MODE_INTERPRETER, /* force interpreter */
    CL_BYTECODE_MODE_TEST, /* both JIT and interpreter, compare results,
                  all failures are fatal */
    CL_BYTECODE_MODE_OFF /* for query only, not settable */
};
    if((opt = optget(opts,"bytecode-mode"))->enabled)
          // JIT mode
          if (!strcmp(opt->strarg, "ForceJIT"))
               mode = CL_BYTECODE_MODE_JIT;
          // Interpreter mode
```

```
if(!strcmp(opt->strarg, "ForceInterpreter"))
    mode = CL_BYTECODE_MODE_INTERPRETER;
// test mode
if(!strcmp(opt->strarg, "Test"))
    mode = CL_BYTECODE_MODE_TEST;
// auto mode
else mode = CL_BYTECODE_MODE_AUTO;
// set the mode variable
cl_engine_set_num(engine, CL_ENGINE_BYTECODE_MODE, mode);
```

loading sigs

loading logic signature - cli_loadcbc

in cl_load will call cli_bytecode_init
================= cli_bytecode_init =================

```
nt cli_bytecode_init(struct cli_all_bc *allbc)
{
    int ret;
    memset(allbc, 0, sizeof(*allbc));
    ret = cli_bytecode_init_jit(allbc, 0/*XXX*/);
    cli_dbgmsg("Bytecode initialized in %s mode\n",
            allbc->engine ? "JIT" : "interpreter");
    allbc->inited = 1;
    return ret;
}
```

================= cli_loadcbc =================
in this function
bc is cli_bc – data structure for a bytecode
bcs is cli_all_bc – data structure for all bytecodes under the engine structure

```
// allocate memory for array of structure cli_bc
// bytecode entry, should be one entry for a cbc file
bcs->all_bcs = cli_realloc2(bcs->all_bcs, sizeof(*bcs->all_bcs)*(bcs->count+1));
bcs->count++;
// get address of current bytecode entry
bc = &bcs->all_bcs[bcs->count-1];
```

```
// parse the cbc file and bytecode information
rc = cli_bytecode_load(bc, fs, dbio, security_trust);
/*
      in cli_bytecode_load
      // to parser the header of the cbc file
      rc = parseHeader(bc, (unsigned char*)firstbuf, &linelength);

      state = PARSE_BC_LSIG; // init stat

parse_state
enum parse_state {
     PARSE_BC_TYPES=0,
     PARSE_BC_APIS,
     PARSE_BC_GLOBALS,
     PARSE_BC_LSIG,
     PARSE_MD_OPT_HEADER,
     PARSE_FUNC_HEADER,
     PARSE_BB,
     PARSE_SKIP
};


*/
// loading the final part of cbc file
while (cli_dbgets(buf, sizeof(buf), fs, dbio)) {}

enum BytecodeKind {
    /** generic bytecode, not tied a specific hook */
    BC_GENERIC=0,
    BC_STARTUP=1,
    _BC_START_HOOKS=256,
    /** triggered by a logical signature */
    BC_LOGICAL=256,
    /** a PE unpacker */
    BC_PE_UNPACKER,
    /* PDF hook */
    BC_PDF,
    BC_PE_ALL,/* both packed and unpacked files */
    _BC_LAST_HOOK
};
/** Bytecode trigger kind */
if (bc->kind == BC_LOGICAL || bc->lsig) // yes for this case as kind=256, means this
bytecode will be triggered upon a logical signature match
      // load the logic sig as normal ones
      rc = load_oneldb(bc->lsig, 0, engine, options, dbname, 0, &sigs, bcs->count,
NULL, &skip);
// one more sig
sigs++;

/** other kind */
if (bc->kind != BC_LOGICAL)
      if (bc->lsig)
```

```
            // log current cbc sig's logic sig id of hook??
            bc->hook_lsig_id = ++engine->hook_lsig_ids;
        // kind _BC_START_HOOKS/BC_LOGICAL/
        // BC_PE_UNPACKER/BC_PDF/BC_PE_ALL
        if (bc->kind >= _BC_START_HOOKS && bc->kind < _BC_LAST_HOOK)
            // get hook type
            unsigned hook = bc->kind - _BC_START_HOOKS;
            // one more sig pending on this hook
            unsigned cnt = ++engine->hooks_cnt[hook];
            engine->hooks[hook][cnt-1] = bcs->count-1;
        else switch (bc->kind)
            //kind BC_STARTUP
            case BC_STARTUP:
                // loop over all cbc sigs loaded before this sig
                // and make sure no BC_STARTUP is loaded before this one
                for (i=0;i<bcs->count-1;i++)
                    if (bcs->all_bcs[i].kind == BC_STARTUP)
                    // error case as only allow to load on BC_STARTUP bytecode
            default: // error case

// how many sigs are loaded in this cbc file
if (signo)
    *signo += sigs;
```

loading logic signature - load_oneldb

```
    // same as normal logic signature
```

compile the engine and bytecode testing run

call stack

```
==================== call stack ====================
cl_engine_compile
    cli_bytecode_prepare2
        run_builtin_or_loaded
            cli_bytecode_prepare_interpreter
            cli_bytecode_context_setfuncid
            cli_bytecode_run
        cli_bytecode_context_getresult_int
        selfcheck
            cli_bytecode_prepare_jit //libclamav/c++/bytecode2llvm.cpp
            cli_bytecode_prepare_interpreter
            run_selfcheck
        cli_bytecode_prepare_jit //libclamav/c++/bytecode2llvm.cpp
        cli_bytecode_done_jit
        cli_bytecode_prepare_interpreter
```

compile the bytecodes - cl_engine_compile

```
==================== run_builtin_or_loaded ====================
/* runs the first bytecode of the specified kind, or the builtin one if no
  * bytecode of that kind is loaded */
static int run_builtin_or_loaded(struct cli_all_bc *bcs, uint8_t kind, const char*
builtin_cbc, struct cli_bc_ctx *ctx, const char *desc)

for (i=0;i<bcs->count;i++)
    bc = &bcs->all_bcs[i];
    if (bc->kind == kind) break;
// if no sig in loaded bytecode is the kind as specified in incoming parameter
// cli_bc will be null
if (i == bcs->count) bc = NULL;

/* no loaded bytecode found, load the builtin one! */
if (!bc)
    bc = cli_calloc(1, sizeof(*bc));
    builtin = 1;
    rc = cli_bytecode_load(bc, NULL, &dbio, 1);
```

// prepare interpreter
rc = cli_bytecode_prepare_interpreter(bc);
/* after return, bc->stat should be bc_interp
bc_state

```
enum bc_state {
    bc_skip,
    bc_loaded,
    bc_jit,
    bc_interp,
    bc_disabled
};
```

*/
// prepare interpreter successfully
if (!rc)
    // set functions loaded from bytecode to context
    cli_bytecode_context_setfuncid(ctx, bc, 0);
    // run bytecode
    rc = cli_bytecode_run(bcs, bc, ctx);

========== cli_bytecode_prepare2 ==========
/* Compile bytecode */
// set to auto mode for testing
engine->bytecode_mode = CL_BYTECODE_MODE_AUTO;

// detect host enviroment
cli_detect_environment(&bcs->env);

switch (bcs->env.arch)
    case arch_i386:
    case arch_x86_64:
        // will not allow JIT x86 mode
        if (!(dconfmask & BYTECODE_JIT_X86)) // no for this case
            // set to CL_BYTECODE_MODE_INTERPRETER mode
            set_mode(engine, CL_BYTECODE_MODE_INTERPRETER)
// allocate memory for cli_bc_ctx
ctx = cli_bytecode_context_alloc();

//run the bytecode with bytecode kind marked as "BC_STARTUP"
rc = run_builtin_or_loaded(bcs, BC_STARTUP, builtin_bc_startup, ctx, "BC_STARTUP");

if (rc != CL_SUCCESS) // issue running bytecode
    ctx->bytecode_disable_status = 2;
else // bytecode running successfully

```
        // get running results
        rc = cli_bytecode_context_getresult_int(ctx);
        // for test case, should return 0xda7aba5e
        if (rc != 0xda7aba5e) // issue happened during bytecode run
            if (engine->bytecode_mode == CL_BYTECODE_MODE_TEST)
                return CL_EBYTECODE_TESTFAIL;
// test failed
switch (ctx->bytecode_disable_status)
        case 1:   return CL_EBYTECODE_TESTFAIL;
        case 2:   return CL_EBYTECODE_TESTFAIL;

if (engine->bytecode_mode != CL_BYTECODE_MODE_INTERPRETER &&
        engine->bytecode_mode != CL_BYTECODE_MODE_OFF)
        // not in interpreter mode or bytecode off mode
        //could be in JIT mode or auto mode
        // do self check
        selfcheck(1, bcs->engine);
        //and testing JIT mode
        rc = cli_bytecode_prepare_jit(bcs);
            if (rc == CL_SUCCESS)
                jitok = 1; // JIT mode test successfully
                    if (engine->bytecode_mode != CL_BYTECODE_MODE_TEST)
                        return CL_SUCCESS;
else
        cli_bytecode_done_jit(bcs, 0);

if (engine->bytecode_mode == CL_BYTECODE_MODE_OFF)
        for (i=0;i<bcs->count;i++)
            // disable all bytecodes as in CL_BYTECODE_MODE_OFF mode
            bcs->all_bcs[i].state = bc_disabled;
            return CL_SUCCESS;
for (i=0;i<bcs->count;i++)
        // current sig can be run in jit mode
        if (bc->state == bc_jit)      jitcount++;
        // current sig can be run in interpreter mode
        if (bc->state == bc_interp)       interp++;
        // test in interpreter mode
        rc = cli_bytecode_prepare_interpreter(bc);
        // one more interpreter sig
        interp++;
```

functions in libclamav/c++/bytecode2llvm.cpp

following functions will be used while doing scan in JIT mode

cli_bytecode_prepare_jit

cli_bytecode_done_jit

cli_vm_execute_jit

bytecode_execute

Scan

call stack

```
==================== the scan call stack ============================
scanfile
   cl_scandesc_callback
      scan_common
         cli_magic_scandesc
            magic_scandesc
               CL_TYPE_TEXT_ASCII //yes
                     cli_scan_structured // no
               cli_scanraw //yes
                     cli_fmap_scandesc //yes
                           matcher_run //yes
                                 cli_bm_scanbuff
                                 cli_ac_scanbuff
                           if(groot) cli_lsig_eval // yes but return clean
               cli_scanscript
                     cli_scanbuff
                           matcher_run
                                 cli_ac_scanbuff
                     cli_lsig_eval
                           cli_bytecode_runlsig
                                 cli_bytecode_context_setfuncid
                                 cli_bytecode_context_setctx
                                 cli_bytecode_context_setfile
                                 cli_bytecode_run
                                       cli_event_time_start
                                       cli_vm_execute_jit //libclamav/c++/bytecode2llvm.cpp
                                       cli_event_time_stop
                                 cli_bytecode_context_getresult_int
```

scan_common

```
==================== scan_common ============================
// bitset
ctx.hook_lsig_matches = cli_bitset_init()
#define BITSET_DEFAULT_SIZE (1024)
```

```
bitset_t *cli_bitset_init(void)
{
    bitset_t *bs;

    bs = cli_malloc(sizeof(bitset_t));
    if (!bs) {
        return NULL;
    }
    bs->length = BITSET_DEFAULT_SIZE;
    bs->bitset = cli_calloc(BITSET_DEFAULT_SIZE, 1);
    if (!bs->bitset) {
        free(bs);
        return NULL;
    }
    return bs;
}
```

magic_scandesc

========== magic_scandesc ==========

```
bitset_t *old_hook_lsig_matches;

// easy to restore
old_hook_lsig_matches = ctx->hook_lsig_matches;
ctx->hook_lsig_matches = NULL;

// init new space
ctx->hook_lsig_matches = cli_bitset_init();

// check types for action
switch(type)
    case CL_TYPE_TEXT_ASCII: // yes for this case
        // no for this case
        if(SCAN_STRUCTURED && (DCONF_OTHER & OTHER_CONF_DLP))

if(type != CL_TYPE_IGNORED && (type != CL_TYPE_HTML || !(DCONF_DOC & DOC_CONF_HTML_SKIPRAW)) && !ctx->engine->sdb) {
    // raw scan is clean in this case
    res = cli_scanraw(ctx, type, typercg, &dettype, hash);
    if(res != CL_CLEAN) // no for this case

ctx->recursion++;

switch(type) // type=500 in this case
    /* bytecode hooks triggered by a lsig must be a hook
    * called from one of the functions here */
    case CL_TYPE_TEXT_ASCII:
    case CL_TYPE_TEXT_UTF16BE:
```

```
    case CL_TYPE_TEXT_UTF16LE:
    case CL_TYPE_TEXT_UTF8:
        if((DCONF_DOC & DOC_CONF_SCRIPT) && dettype != CL_TYPE_HTML &&
    ret != CL_VIRUS) // yes for this case
            ret = cli_scanscript(ctx); // will return 1 in this case
        if(SCAN_MAIL && (DCONF_MAIL & MAIL_CONF_MBOX) && ret != CL_VIRUS
&& (ctx->container_type == CL_TYPE_MAIL || dettype == CL_TYPE_MAIL)) // no
            ret = cli_fmap_scandesc(ctx, CL_TYPE_MAIL, 0, NULL, AC_SCAN_VIR,
NULL, NULL);

// return
ret_from_magicscan(ret);
```

cli_scanscript

```
==================== cli_scanscript ============================
groot = ctx->engine->root[0]; // generic root
troot = ctx->engine->root[7];
// normalize the data
text_normalize_init(&state, normalized, SCANBUFF + maxpatlen);
// init
ret = cli_ac_initdata(&gmdata, groot->ac_partsigs, groot->ac_lsigs,
groot->ac_reloff_num, CLI_DEFAULT_AC_TRACKLEN)

mdata[0] = &tmdata;
mdata[1] = &gmdata;

while(1)
    buff = fmap_need_off_once(map, at, len);
    if(cli_scanbuff(state.out, state.out_pos, offset, ctx, CL_TYPE_TEXT_ASCII, mdata)
== CL_VIRUS) // return 0, no for this case
    // out of the loop
    if(!len) break;

// logical sig re-evaluate
if(ret != CL_VIRUS || SCAN_ALL)
    if ((ret = cli_lsig_eval(ctx, troot, &tmdata, NULL, NULL)) == CL_VIRUS)
        viruses_found++;
    if(ret != CL_VIRUS || SCAN_ALL)
        if ((ret = cli_lsig_eval(ctx, groot, &gmdata, NULL, NULL)) == CL_VIRUS)
            viruses_found++;

// final return
```

return ret;

cli_lsig_eval

```
==================== cli_lsig_eval ====================
// loop over each logic sig
for(i = 0; i < root->ac_lsigs; i++)
    //will cal cli_ac_chklsig in parse_only=0 mode
    // parse_only=0 mode will check if match the logic "(0=1&(1|2)>2&3=3)" after cli_ac_scanbuff match
    // acdata->lsigcnt[i] points to an array stores for match of each logic sub sig
    if(cli_ac_chklsig(root->ac_lsigtable[i]->logic, root->ac_lsigtable[i]->logic + strlen(root->ac_lsigtable[i]->logic), acdata->lsigcnt[i], &evalcnt, &evalids, 0) == 1)
        // check tdb.container against ctx->container_type
        // check tdb.filesize against map->len
        // check tdb.ep against target_info->exeinfo.ep
        // check tdb.nos against target_info->exeinfo.nsections
        // check tdb.handlertype
            cli_magic_scandesc_type // no
        // check tdb.icongrp1 || tdb.icongrp2
            if(matchicon(ctx, &target_info->exeinfo, root->ac_lsigtable[i]->tdb.icongrp1, root->ac_lsigtable[i]->tdb.icongrp2) == CL_VIRUS) // no
                // none bytecode mode
                if(!root->ac_lsigtable[i]->bc_idx)
                    return CL_VIRUS;
                // bytecode mode, run bytecode
                else if(cli_bytecode_runlsig(ctx, target_info, &ctx->engine->bcs, root->ac_lsigtable[i]->bc_idx, acdata->lsigcnt[i], acdata->lsigsuboff_first[i], map) == CL_VIRUS)
                    return CL_VIRUS;

        // none bytecode mode
        if(!root->ac_lsigtable[i]->bc_idx)
            return CL_VIRUS;
        // bytecode mode, run bytecode
        if(cli_bytecode_runlsig(ctx, target_info, &ctx->engine->bcs, root->ac_lsigtable[i]->bc_idx, acdata->lsigcnt[i], acdata->lsigsuboff_first[i], map) == CL_VIRUS)
            return CL_VIRUS;
```

cli_bytecode_runlsig

```
==================== cli_bytecode_runlsig ====================
// set functions defined in bytecode
cli_bytecode_context_setfuncid(&ctx, bc, 0);
    func = ctx->func = &bc->funcs[funcid]; // set functions defined by cbc code
    ctx->bc = bc; // set cli_bc of context
    ctx->numParams = func->numArgs; // set number of args of func, 0 in this case
    ctx->funcid = funcid; // set current function id
    if (func->numArgs) // set arg if necessary, no for this case
    s += 8;/* return value */
    ctx->bytes = s; // how many bytes are used for this func

/*
cli_bc_hooks
struct cli_bc_hooks {
    const uint32_t* match_offsets;
    const uint16_t* kind;
    const uint32_t* match_counts;
    const uint32_t* filesize;
    const struct cli_pe_hook_data* pedata;
};
*/

//Bytecode test_bytecode.cbc(1) has logical signature:
// test_bytecode.{A,B};Engine:56-255,Target:7;((0<1)|(0>1));61616262
// lsigcnt points to an array stores match count of each logic sub sig
// so the match count would be retrieved by lsigcnt[id] where id is the sub logic sig id
// in this case, for logic sig's first pattern "61616262", the match count is 2
// so array lsigcnt will only have one element that is lsigcnt[0]=2
ctx.hooks.match_counts = lsigcnt;
// inited via lsigsuboff_first
// match position in the buffer
// for "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxaabbaabb"
// it's 36, i.e.: "a" in red
ctx.hooks.match_offsets = lsigsuboff;

// for pe info
if (tinfo && tinfo->status == 1) // no for this case


if (bc->hook_lsig_id) // no for this case
    /* this is a bytecode for a hook, defer running it until hook is
     * executed, so that it has all the info for the hook */
    if (cctx->hook_lsig_matches)
```

```
            cli_bitset_set(cctx->hook_lsig_matches, bc->hook_lsig_id-1);
        /* save match counts */
        memcpy(&ctx.lsigcnt, lsigcnt, 64*4);
        memcpy(&ctx.lsigoff, lsigsuboff, 64*4);


/* Running bytecode for logical signature match*/
ret = cli_bytecode_run(bcs, bc, &ctx);

// find a virus
if (ctx.virname)
      // doing PUA scan if we matching heuristics sig
      if (!strncmp(ctx.virname, "BC.Heuristics", 13))
            rc = cli_found_possibly_unwanted(cctx);
      else // report a virus
            rc = CL_VIRUS;

// no virus found
// bytecode return an result code and get it
ret = cli_bytecode_context_getresult_int(&ctx);
```

cli_bytecode_run

========== cli_bytecode_run ==========

cli_bc_inst

```
struct cli_bc_inst {
    enum bc_opcode opcode;
    uint16_t type;
    operand_t dest;
    interp_op_t interp_op;/* opcode for interpreter */
    union {
    operand_t unaryop;
    struct cli_bc_cast cast;
    operand_t binop[2];
    operand_t three[3];
    struct cli_bc_callop ops;
    struct branch branch;
    bbid_t jump;
    } u;
};
```

cli_bc_func

```
struct cli_bc_func {
    uint8_t numArgs;
    uint16_t numLocals;
    uint32_t numInsts;
    uint32_t numValues;/* without constants */
    uint32_t numConstants;
    uint32_t numBytes;/* stack size */
    uint16_t numBB;
    uint16_t returnType;
    uint16_t *types;
    uint32_t insn_idx;
    struct cli_bc_bb *BB;
    struct cli_bc_inst *allinsts;
    uint64_t *constants;
    unsigned *dbgnodes;
};
```

```
struct cli_events;
typedef struct cli_events cli_events_t;
struct cli_events {
    struct cli_event *events;
    struct cli_event errors;
    uint64_t oom_total;
    unsigned max;
    unsigned oom_count;
};
```

```
// some local vars before vm execute
struct cli_bc_inst inst;
struct cli_bc_func func;
cli_events_t *jit_ev = NULL, *interp_ev = NULL;

// get running env
ctx->env = &bcs->env;
context_safe(ctx); /* make sure some vars in ctx are never NULL */

if (test_mode) // not in test mode for this case
if (bc->state == bc_interp || test_mode) // no for this case
      cli_event_time_start(interp_ev, BCEV_EXEC_TIME);
      ret = cli_vm_execute(ctx->bc, ctx, &func, &inst);
      cli_event_time_stop(interp_ev, BCEV_EXEC_TIME);
      cli_event_int(interp_ev, BCEV_EXEC_RETURNVALUE, ret);
      cli_event_string(interp_ev, BCEV_VIRUSNAME, ctx->virname);


// deal with JIT mode as for this case
// bc->state == bc_jit
//bc_state
```

```
enum bc_state {
    bc_skip,
    bc_loaded,
    bc_jit,
    bc_interp,
    bc_disabled
};
```

```
if (bc->state == bc_jit || test_mode)
      ctx->bc_events = jit_ev;
      ctx->on_jit = 1;
      // execute
       cli_event_time_start(jit_ev, BCEV_EXEC_TIME);
       ret = cli_vm_execute_jit(bcs, ctx, &bc->funcs[ctx->funcid]);
            // called from libclamav/c++/bytecode2llvm.cpp
       cli_event_time_stop(jit_ev, BCEV_EXEC_TIME);
      // post execute
       cli_event_int(jit_ev, BCEV_EXEC_RETURNVALUE, ret);
       // get the virus name identified according bytecode executed by JIT vm
       // ctx->virname will be used after return as a judgment of if there's a match or
not
       cli_event_string(jit_ev, BCEV_VIRUSNAME, ctx->virname);
```