Clamav funcation call flow(logic sig scan)

logic sig scan explained

by eqmcc

Description

this document will talk about logic signature scan

1. about logic signature addition

a logic signature looks like bellow:

```
test_logic_sig;Target:0;(0=1&(1|2)>2&3=3);6f6f6f{4-6}727272;6b6b6b;6d6d6d{3-4}6d6d;6e6e6e
```

6f6f6f{4-6}727272;6b6b6b;6d6d6d{3-4}6d6d;6e6e6e will be breakdown into several sub logic signatures(each logic sub sig is in fact standard Extended signature format) and later each sub logic sig will be further breakdown into sub signatures if necessary e.g.: sub logic sig(6f6f6f{4-6}727272) can be as in normal scan case break down into sub sigs 6f6f6f and 727272, the whole sig addition process is same as normal ac sig addition except should also log the logic sig id for further reference

2. about logic signature matching

during matching phase, in cli_ac_scanbuff:

first all subsigs of a sub logic sig will be matched via ac_findmatch(e.g.: for sub logic sig(6f6f6f{4-6}727272), sub sigs 6f6f6f and 727272 will be matched, then in if(pt->sigid), will checking the offset info to further match the sub logic sig as whole),

secondly, in if(pt->lsigid[0]) inside if(pt->sigid), lsig_sub_matched will be called to log the match count of each logic sub sig, the count will be used in future to evaluate against the logic((0=1&(1|2)>2&3=3)) of the logic signature in cli_lsig_eval.

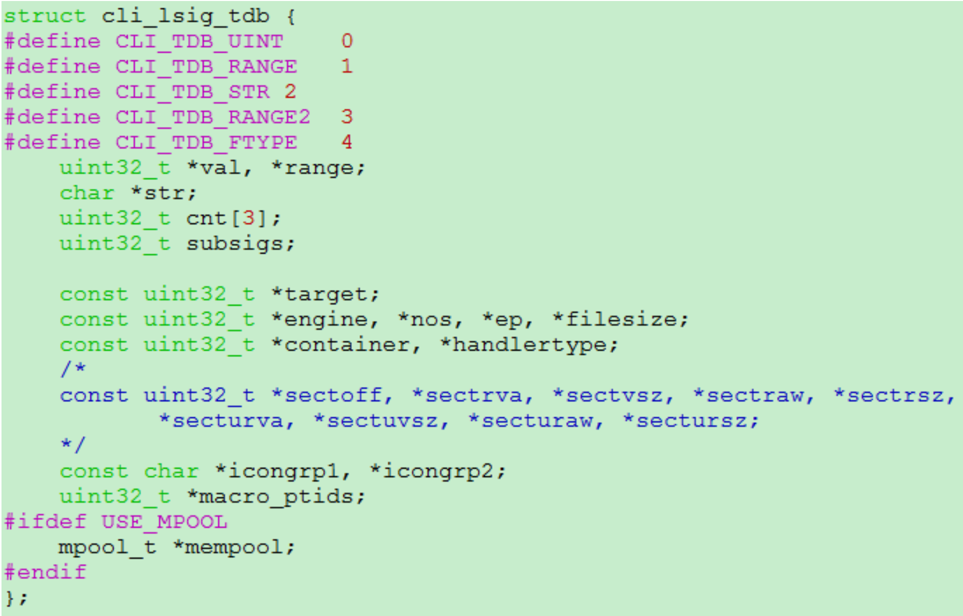
also in cli_ac_scanbuff, the return value will be "0" instead of CLI_VIRUS and after returned, in cli_fmap_scandesc, cli_lsig_eval will be called to decide if there's a match of a logic sig as a whole.

Data structures

cli_ac_lsig // the logic signature

```
struct cli_ac_lsig {
    uint32_t id;
    unsigned bc_idx;
    char *logic;
    const char *virname;
    struct cli_lsig_tdb tdb;
};
```

cli_lsig_tdb //the logic sig's detailed info



```
struct cli_lsig_tdb {
#define CLI_TDB_UINT    0
#define CLI_TDB_RANGE   1
#define CLI_TDB_STR 2
#define CLI_TDB_RANGE2  3
#define CLI_TDB_FTYPE   4
    uint32_t *val, *range;
    char *str;
    uint32_t cnt[3];
    uint32_t subsigs;

    const uint32_t *target;
    const uint32_t *engine, *nos, *ep, *filesize;
    const uint32_t *container, *handlertype;
    /*
    const uint32_t *sectoff, *sectrva, *sectvsz, *sectraw, *sectrsz,
           *secturva, *sectuvsz, *secturaw, *sectursz;
    */
    const char *icongrp1, *icongrp2;
    uint32_t *macro_ptids;
#ifdef USE_MPOOL
    mpool_t *mempool;
#endif
};
```

Test case

test.txt

testoooeqmccrrrkekekkkOmmmMMMmmOkkkKeKennNnnnNnnnNnnntest

ldb test
create ldb signature for test.txt

format

SignatureName;TargetDescriptionBlock;LogicalExpression;Subsig0;Subsig1;Subsig2;...
where:

- TargetDescriptionBlock provides information about the engine and target file with comma separated Arg:Val pairs, currently (as of 0.95.1) only Target:X and Engine:X-Y are supported.
- LogicalExpression specifies the logical expression describing the relationship between Subsig0...SubsigN.

Basis clause: 0,1,...,N decimal indexes are SUB-EXPRESSIONS representing Subsig0, Subsig1,...,SubsigN respectively.

Inductive clause: if A and B are SUB-EXPRESSIONS and X, Y are decimal numbers then (A&B), (A|B), A=X, A=X,Y, A>X, A>X,Y, A<X and A<X,Y are SUB-EXPRESSIONS

- SubsigN is n-th subsignature in extended format possibly preceded with an offset. There can be specified up to 64 subsigs.

Keywords used in TargetDescriptionBlock:

- Target:X: Target file type
- Engine:X-Y: Required engine functionality (range; 0.96)

- FileSize:X-Y: Required file size (range in bytes; 0.96)
- EntryPoint: Entry point offset (range in bytes; 0.96)
- NumberOfSections: Required number of sections in executable (range; 0.96)
- Container:CL_TYPE_*: File type of the container which stores the scanned file

Modifiers for subexpressions:

- A=X: If the SUB-EXPRESSION A refers to a single signature then this signature must get matched exactly X times; if it refers to a (logical) block of signatures then this block must generate exactly X matches (with any of its sigs).
- A=0 specifies negation (signature or block of signatures cannot be matched)
- A=X,Y: If the SUB-EXPRESSION A refers to a single signature then this signature must be matched exactly X times; if it refers to a (logical) block of signatures then this block must generate X matches and at least Y different signatures must get matched.
- A>X: If the SUB-EXPRESSION A refers to a single signature then this signature must get matched more than X times; if it refers to a (logical) block of signatures then this block must generate more than X matches (with any of its sigs).
- A>X,Y: If the SUB-EXPRESSION A refers to a single signature then this signature must get matched more than X times; if it refers to a (logical) block of signatures then this block must generate more than X matches and at least Y different signatures must be matched.
- A<X and A<X,Y as above with the change of ”more” to ”less”.

Examples:

```
Sig1;Target:0;(0&1&2&3)&(4|1);6b6f74656b;616c61;7a6f6c77;7374656661 6e;deadbeef
```

```
Sig1;Target:0;(0&1&2&3)&(4|1);6b6f74656b;616c61;7a6f6c77;73746566616e;deadbeef
Sig2;Target:0;((0|1|2)>5,2)&(3|1);6b6f74656b;616c61;7a6f6c77;73746566616e
Sig3;Target:0;((0|1|2|3)=2)&(4|1);6b6f74656b;616c61;7a6f6c77;73746566616e;deadbeef
Sig4;Target:1,Engine:18-20;((0|1)&(2|3))&4;EP+123:33c06834f04100f2aef7d14951684cf04100e8
110a00;S2+78:22??232c2d252229{-15}6e6573(63|64)61706528;S+50:68efa311c3b9963cb1ee8e
586d32aeb9043e;f9c58dcf43987e4f519d629b103375;SL+550:6300680065005c0046006900
```

signature

```
test_logic_sig;Target:0;(0=1&(1|2)>2&3=3);6f6f6f{4-6}727272;6b6b6b;6d6d6d{3-4}6d6d;6e6e6e
test_logic_sig;Target:0;(0=1&(1|2)>2&3=3);ooo{4-6}rrr;kkk;mmm{3-4}mm;nnnn
```

Above is a logic signature, so it will be loaded via cli_loadldb and later in load_oneld

loading logic signature - load_oneldb

load_oneldb

```
/*
    in this function, logic sig like
```

```
        SignatureName;TargetDescriptionBlock;LogicalExpression;Subsig0;Subsig1;Subsig2;...
        will be parsed to get infor into a structure called cli_lsig_tdb
        especially, the number of sub logic signatures will be retrieved from
LogicalExpression
        here sub logic signature and sub signature are different, a sub logic signature
may contain several sub sigs, e.g.: this logic sub sig "6f6f6f{4-6}727272" actually is a
standard extend format signature with two sub sigs.
        each of the logic sub sig will be added into ac tire using cli_parse_add
*/
struct cli_lsig_tdb tdb;
uint32_t lsigid[2];

// tokenize the buffer of logic sig
test_logic_sig;Target:0;(0=1&(1|2)>2&3=3);6f6f6f{4-6}727272;6b6b6b;6d6d6d{3-4}6
d6d;6e6e6e
tokens_count = cli_strtokenize(buffer, ';', LDB_TOKENS + 1, (const char **) tokens);
virname = tokens[0]; // test_logic_sig
logic = tokens[2]; // (0=1&(1|2)>2&3=3)

// callback function
// the function to set this callback function is defined in libclamav/others.c
```

```
void cl_engine_set_clcb_sigload(struct cl_engine *engine, clcb_sigload callback, void *context) {
    engine->cb_sigload = callback;
    engine->cb_sigload_ctx = callback ? context : NULL;
}
```

```
// and called in
// win32/clamav-for-windows/clamav-for-windows/interface.c
engine->cb_sigload("ldb", virname, ~options &
CL_DB_OFFICIAL,engine->cb_sigload_ctx)

/* get last sub logic sig count id from the logic expression (0=1&(1|2)>2&3=3)
    here sub logic signature and sub signature are different, a sub logic sig may contain
several sub sigs
    keys:
        parenthesis: ()
        compare: >=<
        bool operation: &|
*/
// check in parse_only mode
// In parse_only mode this function returns -1 on error or the max subsig id
subsigs = cli_ac_chklsig(logic, logic + strlen(logic), NULL, NULL, NULL, 1);
subsigs++; // how many sigs do we have

//SignatureName;TargetDescriptionBlock;LogicalExpression;Subsig0;Subsig1;Subsig2;...
// get attributes from TargetDescriptionBlock to tdb(cli_lsig_tdb) structure
```

```
ret = lsigattribs(tokens[1], &tdb)
// verify some info loaded into tdb
tdb.engine
tdb.target
tdb.icongrp1 || tdb.icongrp2
tdb.ep || tdb.nos

// get engine root for signature target
root = engine->root[tdb.target[0]];

// allocate memory for logic sig
lsig->logic = cli_mpool_strdup(engine->mempool, logic);

// get the logic sig id
// lsigid[0] for parent logic sig id
// lsigid[1] for sub logic sig id
lsigid[0] = lsig->id = root->ac_lsigs;

/* for bytecodfe: 0 marks no bc */
lsig->bc_idx = bc_idx;
// store the lsig pointer to global table
newtable[root->ac_lsigs-1]=lsig
// global table
root->ac_lsigtable = newtable;
//get number of sub logic sigs
tdb.subsigs = subsigs

// adding all the sub logic sigs
for(i = 0; i < subsigs; i++)

    lsigid[1] = i; // sub logic id inside a logic sig
    sig = tokens[3 + i]; //go to start of current sub logic sig

    // get offset info and make it to adhere standard extend signature format
    // sub logic sig with offset info would using following format:
    //subsig=[offset:]pattern
    if((pt = strchr(tokens[3 + i], ':'))) // have offset info
        *pt = 0;
        sig = ++pt;
        offset = tokens[3 + i]; // get offset
    else // no offset info specified, log as '*'
        offset = "*";
        sig = tokens[3 + i];
    // now add the sig
    cli_parse_add(root, virname, sig, 0, 0, offset, target, lsigid, options)

// copy over tdb info to cli_ac_lsig structure
memcpy(&lsig->tdb, &tdb, sizeof(tdb));
```

loading logic signature - cli_parse_add

cli_parse_add

for logic sig case, all sigs should be added into ac data structure and never goes to bm data structure

```
if (hexsig[0] == '$') //MACRO
    //macro signatures only valid inside logical signatures
    if (!lsigid) return CL_EMALFDB;
    ret = cli_ac_addpatt(root, patt)
if((wild = strchr(hexsig, '{')))
    // same as normal ac sig addition
    //
test_logic_sig;Target:0;(0=1&(1|2)>2&3=3);6f6f6f{4-6}727272;6b6b6b;6d6d6d{3-4}6d6d;6e6e6e
    // split sub logic into sub sigs if necessary, e.g.: logic sub sig(6f6f6f{4-6}727272) will
be split into sub sigs 6f6f6f and 727272
if(strchr(hexsig, '*'))
    // same as normal ac sig addition
if(root->ac_only || type || lsigid || strpbrk(hexsig, "?([") || (root->bm_offmode &&
(!strcmp(offset, "*") || strchr(offset, ','))) || strstr(offset, "VI") || strchr(offset, '$'))
    // same as normal ac sig addition
```

add signature(pre processing for regular expression) - AC

cli_ac_addsig

```
    struct cli_ac_patt *new;
    // in cli_ac_patt, uint32_t lsigid[3];

    // get parent-sig id, number of parts, part index, mindist and maxdist
    new->sigid = sigid;
    new->parts = parts;
    new->partno = partno;
    new->mindist = mindist;
    new->maxdist = maxdist;
    new->ch[0] |= CLI_MATCH_IGNORE;
```

```
new->ch[1] |= CLI_MATCH_IGNORE;

// special treatment for logic sig
if(lsigid)
      // in cli_ac_patt, uint32_t lsigid[3];
      new->lsigid[0] = 1; // indicates we got a logic sig
      // copy over lsigid id passed in as variable
      // so here:
      // new->lsigid[1]: the parent logic sig id
      // new->lsigid[2]: sub logic sig id
      memcpy(&new->lsigid[1], lsigid, 2 * sizeof(uint32_t));

… …
// others are same as normal ac sig addition
… …

if(new->lsigid[0])
      // get virus name for each logic signature
      root->ac_lsigtable[new->lsigid[1]]->virname = new->virname;

… …
// others are same as normal ac sig addition
… …

  // add the pattern into ac tire
   cli_ac_addpatt
```

add pattern to AC tire - cli_ac_addpatt

nothing special than normal ac sig addition

compile the tire - ac_maketrans

nothing special than normal ac sig addition

Scan - cli_ac_scanbuff

```
===================== the scan call stack =============================
scanfile
   cl_scandesc_callback
      scan_common
         cli_magic_scandesc
            magic_scandesc
               cli_scanraw
                  cli_fmap_scandesc

cli_fmap_scandesc
   matcher_run
      cli_ac_scanbuff
   if(groot)
      if(ret != CL_VIRUS || SCAN_ALL)
           // further check the match results of each sub logic sig
           // against logic expression
         ret = cli_lsig_eval(ctx, groot, &gdata, &info, refhash);
```

===================== some pre-config ============================

in cli_scanbuff, cli_ac_initdata will be called to init some data structure used for final scan

cli_ac_initdata

```
struct cli_ac_data *data
data->partsigs = partsigs;
if(partsigs)
     // allocate space of 4bytes*partsigs and inited as 0
     data->offmatrix = (int32_t ***) cli_calloc(partsigs, sizeof(int32_t **));

data->lsigs = lsigs;
if(lsigs)
```

```
    // allocate memory for 2 dimension array
    data->lsigcnt = (uint32_t **) cli_malloc(lsigs * sizeof(uint32_t *));
    // max 64 sub logic sig in a logic sig
    data->lsigcnt[0] = (uint32_t *) cli_calloc(lsigs * 64, sizeof(uint32_t));
    for(i = 1; i < lsigs; i++)
        // locate entry of each logic sig
        data->lsigcnt[i] = data->lsigcnt[0] + 64 * i;

    /*  subsig offsets
        allocate memory and locate entries
      */
    data->lsigsuboff_last = (uint32_t **) cli_malloc(lsigs * sizeof(uint32_t *));
    data->lsigsuboff_first = (uint32_t **) cli_malloc(lsigs * sizeof(uint32_t *));
    data->lsigsuboff_last[0] = (uint32_t *) cli_calloc(lsigs * 64, sizeof(uint32_t));
    data->lsigsuboff_first[0] = (uint32_t *) cli_calloc(lsigs * 64, sizeof(uint32_t));
    for(j = 0; j < 64; j++)
          data->lsigsuboff_last[0][j] = CLI_OFF_NONE;
          data->lsigsuboff_first[0][j] = CLI_OFF_NONE;
    for(i = 1; i < lsigs; i++)
          data->lsigsuboff_last[i] = data->lsigsuboff_last[0] + 64 * i;
          data->lsigsuboff_first[i] = data->lsigsuboff_first[0] + 64 * i
          for(j = 0; j < 64; j++)
                    data->lsigsuboff_last[i][j] = CLI_OFF_NONE;
                    data->lsigsuboff_first[i][j] = CLI_OFF_NONE;
```

==================== the scan ============================

```
// the scan procedure is same as normal ac sig scan except having following logic sig match function
/*
    the call stack
    if(ac_findmatch(buffer, bp, offset + bp - patt->prefix_length, length, patt, &matchend))
      while(pt)
        if(pt->sigid) // partial sig
            if(pt->partno == 1 || (found && (pt->partno != pt->parts)))
            else if(found && pt->partno == pt->parts)
                if(pt->type)
                else { /* !pt->type, general sig type */
                    if(pt->lsigid[0])
        else // old type sig
            if(pt->type)
            else
                if(pt->lsigid[0])
```

```
*/
// got a logic sig
if(pt->lsigid[0])
    // match it
    lsig_sub_matched(root, mdata, pt->lsigid[1], pt->lsigid[2], offmatrix[pt->parts - 1][1], 1);
    pt = pt->next_same;
    /*
        after matching logic sig via lsig_sub_matched, it will never return CL_VIRUS but instead will always continue trying next sig and finally will do:
        return (mode & AC_SCAN_FT) ? type : CL_CLEAN;
        "0" in this case
        and the logic sig scan results will be further investigated via
        cli_lsig_eval
    */
    continue; // always continue

return (mode & AC_SCAN_FT) ? type : CL_CLEAN;


===================== lsig_sub_matched ============================
/*
    this function will log the match offset of the sub logic sig
    log the first and last match offset
    log the match count
    while will be used as metric to compare against the logic expression
    (0=1&(1|2)>2&3=3)
*/

void lsig_sub_matched(const struct cli_matcher *root, struct cli_ac_data *mdata, uint32_t lsigid1, uint32_t lsigid2, uint32_t realoff, int partial)

// get parent logic info from global table
const struct cli_lsig_tdb *tdb = &root->ac_lsigtable[lsigid1]->tdb;
// always true in scan mode as will always be a offset to matched at ac_findmatch
if(realoff != CLI_OFF_NONE)
    // first time scan this sub logic sig
    if(mdata->lsigsuboff_first[lsigid1][lsigid2] == CLI_OFF_NONE)
        mdata->lsigsuboff_first[lsigid1][lsigid2] = realoff; //get current match offset
    // mdata->lsigsuboff_last[lsigid1][lsigid2] != CLI_OFF_NONE: not first scan of this sub logic sig
    // (!partial && realoff <= mdata->lsigsuboff_last[lsigid1][lsigid2]): not partial mode and current match offset are smaller than last time
    //(partial && realoff < mdata->lsigsuboff_last[lsigid1][lsigid2]): partial mode and
```

current match offset are smaller than last time

```
    if(mdata->lsigsuboff_last[lsigid1][lsigid2] != CLI_OFF_NONE && ((!partial && realoff <= mdata->lsigsuboff_last[lsigid1][lsigid2]) || (partial && realoff < mdata->lsigsuboff_last[lsigid1][lsigid2])))
        return; // match before last time’s match, ignore
    mdata->lsigcnt[lsigid1][lsigid2]++; // otherwise, one more sub logic sigs match
    // no more than one match of a sub logic sig
    if(mdata->lsigcnt[lsigid1][lsigid2] <= 1 || !tdb->macro_ptids || !tdb->macro_ptids[lsigid2])
        // log the offset info
        mdata->lsigsuboff_last[lsigid1][lsigid2] = realoff;

// bellow will handle macro case, not suitable for this case
if (mdata->lsigcnt[lsigid1][lsigid2] > 1)
    if (!tdb->macro_ptids) return; // no macro sig match, just return
```

==================== cli_ac_chklsig ====================

this function is called at

load_oneldb - parse_only=1 mode

in parse_only=1 mode, will return last sub logic sig index inside the logic sig

cli_lsig_eval - parse_only=0 mode

in parse_only=0 mode, after comparing the match count of each sub logic sig, will return 1 if achieves full match of a logic sig;

```
/*
    (0=1&(1|2)>2&3=3)
    in cli_ac_chklsig checking "(0=1&(1|2)>2&3=3)"
    in cli_ac_chklsig checking "0=1&(1|2)>2&3=3)"
    in cli_ac_chklsig checking "0=1&(1|2)>2&3=3)"
    in cli_ac_chklsig checking "(1|2)>2&3=3)"
    in cli_ac_chklsig checking "1|2)>2&3=3)"
    in cli_ac_chklsig checking "2)>2&3=3)"
    in cli_ac_chklsig checking "3=3)"
*/
/*
  keys:
    parenthesis: () - pth
    compare: >=< - mod/modoff
    bool operation: &|- op/opoff or op1/op1off - op1 means op inside parenthesis
*/

// 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
/*
```

```
test_logic_sig;Target:0;(0=1&(1|2)>2&3=3);6f6f6f{4-6}727272;6b6b6b;6d6d6d{3-4}6d6d;6e6e6e
```

```
  in cli_lsig_eval acdata->lsigcnt[49][0]=1    // 6f6f6f{4-6}727272
  in cli_lsig_eval acdata->lsigcnt[49][1]=2    // 6b6b6b
  in cli_lsig_eval acdata->lsigcnt[49][2]=1    // 6d6d6d{3-4}6d6d
  in cli_lsig_eval acdata->lsigcnt[49][3]=3    // 6e6e6e
*/
int cli_ac_chklsig(const char *expr, const char *end, uint32_t *lsigcnt, unsigned int *cnt, uint64_t *ids, unsigned int parse_only)
for(i = 0; i < len; i++)
    // get parenthesis/compare/operation
    switch(expr[i])
        case '(':
        case ')':
        case '>':
        case '<':
        case '=':
        default:
            if(strchr("&|", expr[i]))

if(!op && !op1)
    if(expr[0] == '(')
        // recursively checking
        return cli_ac_chklsig(++expr, --end, lsigcnt, cnt, ids, parse_only);

    // get sub logic id from the logic
    // i.e.: 0/1/2/3 from (0=1&(1|2)>2&3=3)
    ret = sscanf(expr, "%u", &id);
    if(parse_only)    val = id;
    else        val = lsigcnt[id]; // get match count of a sub logic sig

    if(mod) // >/=/<
        // get the mode
        ret = sscanf(pt, "%u", &modval1);
        if(!parse_only) // parse_only=0 mode
            switch(mod)
                // 0=1 and 3=3
                case '=': if(val != modval1) return 0;
                // <
                case '<': if(val >= modval1) return 0;
                // (1|2)>2
                case '>': if(val <= modval1) return 0;
// recursively checking left and right part of a “&” or “|”
```

```
lval = cli_ac_chklsig(lstart, lend, lsigcnt, &lcnt, &lids, parse_only);
rval = cli_ac_chklsig(rstart, rend, lsigcnt, &rcnt, &rids, parse_only);

if(parse_only)
    switch(op)
        case '&':
        case '|':
else
    switch(op)
        // merge left and right expression value
        case '&': ret = lval && rval;
        case '|': ret = lval || rval;


===================== 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
        // 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)
                    // 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;
```