
Clamav funcation call flow(AC scan)

ac algorithm adding pattern explained

by eqmcc

Clamav funcation call flow(AC scan)	1
The function cli_ac_addpatt.....	2
Data structures	2
Test case	4
the loading:.....	4
add signature(pre processing for regular expression) - AC.....	5
add pattern to AC tire.....	6

The function cli_ac_addpatt

this function will adding signature into existing ac tire

Data structures

cli_ac_node // store info of each node in a ac tire

```
struct cli_ac_node {  
    struct cli_ac_patt *list;  
    struct cli_ac_node **trans, *fail;  
};
```

list: a list of signature with same prefix, only node which is last letter of signature will have the list. e.g.: for sig "abcd", "abce" and "ab", both node "b" and "c" will have list, furthermore, for node "b"'s list, it's pattern "ab"

trans: an array of 256 – the size of a ASCII table

fail: the fail node chain

cli_ac_patt // store data for each signature

```
struct cli_ac_patt {  
    uint16_t *pattern, *prefix, length, prefix_length;  
    uint32_t mindist, maxdist;  
    uint32_t sigid;  
    uint32_t lsigid[3];  
    uint16_t ch[2];  
    char *virname;  
    void *customdata;  
    uint16_t ch_mindist[2];  
    uint16_t ch_maxdist[2];  
    uint16_t parts, partno, special, special_pattern;  
    struct cli_ac_special **special_table;  
    struct cli_ac_patt *next, *next_same;  
    uint16_t rtype, type;  
    uint32_t offdata[4], offset_min, offset_max;  
    uint32_t boundary;  
    uint8_t depth;  
};
```

mindist, maxdist: the slide between sub sigs

sigid: the mark the parent id of sub sig

parts: number of sub sigs in a signature

partno: current sub sig id in a signature

next: pattern list for signatures have same effective pattern but other signature information is not exactly the same

next_same: pattern list for signatures have same effective pattern and also other signature information is exactly the same

offset_min, offsetmax: the slide of signature matching start

depth: at which level the node residents, corresponding to the char's position inside a signature

the root of a ac tire is cli_matcher->ac_root which is also a cli_ac_node type
and for each sig/subsig adding into ac tire, bellow code in cli_ac_addpatt will make sure the effective length of sig used in this algo should be either 2 or 3:

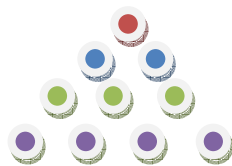
```
uint16_t len = MIN(root->ac_maxdepth, pattern->length);  
if(len < root->ac_mindepth) return CL_EMALFDB;
```

here:

ac_maxdepth is initied as CLI_DEFAULT_AC_MAXDEPTH

ac_mindepth is initied as CLI_DEFAULT_AC_MINDEPTH

As the ac tire only have max 4 levels and min 3 levels with ac_root taken into account



level 0 – ac_root

level 1 – should not be leaf

level 2 – could be leaf

level 3 – could be leaf

So if first 2 or 3 characters of the signature are same, these sigs would be ending in the same leaf node linked via list like cli_ac_patt->next and cli_ac_patt->next_same

Test case

test.txt

```
NWSTARToooTESTkkkMYOtestTEST
```

ndb test

create ndb signature for test.txt

format

MalwareName:TargetType:Offset:HexSignature

File

test1.ndb	test_ndb_sig1:0:13,15:6f6f6f{1-2}6e6e6b6b6b	ooo{1-2}nnkkk
test2.ndb	test_ndb_sig2:0:0:6f6f6f{1-2}6d6d6b6b6b	ooo{1-2}mmkkk
test3.ndb	test_ndb_partsig:0:3,5:6f6f6e{1-2}6b6b6b	oon{1-2}kkk
test.ndb	test_ndb_partsig:0:3,5:6f6f6f{4-6}6b6b6b	ooo{4-6}kkk

Above is a regex(with range) signature with offset info, so it will be loaded into ac pattern structure and will be used in ac scan mode, the signature has a range information and the signature will be split into two sub-sigs.

with loading these signatures, will investigate how these sigs/sub-sigs are added into pattern list like:

cli_ac_node->list

cli_ac_patt->next

cli_ac_patt->next_same

the loading:

e.g.: this signature “test_ndb_regex:0:3,5:6f6f6f{4-6}6b6b6b” has regular expression with floating range involved, so should be loaded into AC scan structure and split into 2 sub signatures. These two sub sigs will be loaded in separately(has no direct connection in ac tire) and will be used in ac_scanbuff via offmetrix which will be bind together with parent sig id.

add signature(pre processing for regular expression) - AC

cli_ac_addsig

```
// 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;
```

```
// dealing case as "[" – "HEXSIG[x-y]aa or aa[x-y]HEXSIG"
```

```
if(strchr(hexsig, '[')) // with "[" – no match for this case
```

```
if(strchr(hexsig, '(')) // with "(" –no match for this case
```

```
// dealing other case
```

```
new->pattern = cli_mpool_hex2ui(root->mempool, hex ? hex : hexsig);
```

```
// new->pattern is uint16_t
```

```
cli_mpool_hex2ui
```

```
cli_realhex2ui // in this function, each byte of the pattern would be  
extended to uint16_t(low byte for the pattern byte and high byte for the matching  
type corresponding to the regular expression type)
```

```
#define CLI_MATCH_WILDCARD 0xff00  
#define CLI_MATCH_CHAR 0x0000  
#define CLI_MATCH_IGNORE 0x0100  
#define CLI_MATCH_SPECIAL 0x0200  
#define CLI_MATCH_NIBBLE_HIGH 0x0300  
#define CLI_MATCH_NIBBLE_LOW 0x0400
```

```
if(hex[i] == '?' && hex[i + 1] == '?') val |= CLI_MATCH_IGNORE;
```

```
if(hex[i + 1] == '?') val |= CLI_MATCH_NIBBLE_HIGH;
```

```
if(hex[i] == '?') val |= CLI_MATCH_NIBBLE_LOW;
```

```
if(hex[i] == '(') val |= CLI_MATCH_SPECIAL;
```

```
filter_add_acpatt /* prefiltering
```

```
// check if there's regex in first letters
```

```
if(new->pattern[i] & CLI_MATCH_WILDCARD)
```

```
cli_caloff /*"test_ndb_regex:0:3,5:6f6f6f{4-6}6b6b6b"
```

```
if((pt = strchr(offcpy, ',')) offdata[2] = atoi(pt + 1); // which is 5
```

```
offdata[0] = CLI_OFF_ABSOLUTE;
```

```
*offset_min = offdata[1] = atoi(offcpy); // which is 3
```

```
*offset_max = *offset_min + offdata[2]; // which is 8
```

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

```
cli_ac_addpatt
```

add pattern to AC tire

===== the example =====

signatures:

test1.ndb	test_ndb_sig1:0:13,15:6f6f6f{1-2}6e6e6b6b6b	ooo{1-2}nnkkk
test2.ndb	test_ndb_sig2:0:0:6f6f6f{1-2}6d6d6b6b6b	ooo{1-2}mmkkk
test3.ndb	test_ndb_partsig:0:3,5:6f6f6e{1-2}6b6b6b	oon{1-2}kkk
test.ndb	test_ndb_partsig:0:3,5:6f6f6f{4-6}6b6b6b	ooo{4-6}kkk

loading sequence:

1. test_ndb_sig2
2. test_ndb_partsig
3. test_ndb_sig3
4. test_ndb_sig1

flow:

1. test_ndb_sig2

- a. for(i = 0; i < len; i++) // loop over effective pattern, length is 3
- b. while(ph) // false, ph = pt->list(null)
- c. if(ph_add_after) // false
- d. else // true
 - a) pattern->next = pt->list; // pattern->next will be null
 - b) pt->list = pattern;

2. test_ndb_partsig

- a. for(i = 0; i < len; i++) // loop over effective pattern, length is 3
- b. while(ph) // yes, ph = pt->list(has test_ndb_sig2)
 - a) // !ph_add_after is true
// ph->partno <= pattern->partno is true, both are 1
// !ph->next is true
if(!ph_add_after && ph->partno <= pattern->partno && (!ph->next || ph->next->partno > pattern->partno)) // true
ph_add_after = ph;
 - b) // ph->length == pattern->length is true, both are 3
// ph->prefix_length == pattern->prefix_length is true, both is 0
// no special char, so ch[0] and ch[1] are not changed after initialized
if((ph->length == pattern->length) && (ph->prefix_length == pattern->prefix_length) && (ph->ch[0] == pattern->ch[0]) && (ph->ch[1] == pattern->ch[1])) // true
 - i. if(!memcmp(ph->pattern, pattern->pattern, ph->length * sizeof(uint16_t)) && !memcmp(ph->prefix, pattern->prefix, ph->prefix_length * sizeof(uint16_t))) // true, double compare each char in effective pattern and prefix if there is

- ii. // match=1 and do
 iii. and since partno are same, so will do 'else' block in bellow code.

```
if(match) {
    if(pattern->partno < ph->partno) {
        pattern->next_same = ph;
        if(ph_prev)
            ph_prev->next = ph->next;
        else
            pt->list = ph->next;
        ph->next = NULL;
        break;
    } else {
        while(ph->next_same && ph->next_same->partno < pattern->partno)
            ph = ph->next_same;
        pattern->next_same = ph->next_same;
        ph->next_same = pattern;
        return CL_SUCCESS;
    }
}
```

in 'else' block, insert according the partno with inc sequence
 and **return CL_SUCCESS** means will not add into list
 also

if match!=1 do:

ph_prev = ph; ph = ph->next; // try next pattern in list, ph_prev
 points to the pattern right before current pattern in list of in tire
 node

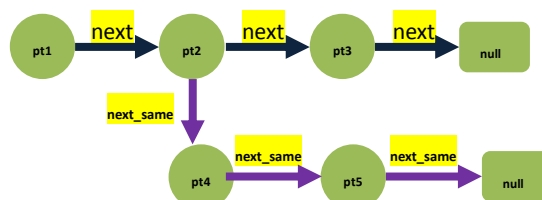
if match=1 but the partno is not current parrtten's number of part
 sigs is less than current tire one's, current pattern of tire will be as
 current pattern's next_same and previous node's next pattern will
 be current pattern

the graph to illustrate:

for **nodeX**'s list:

current

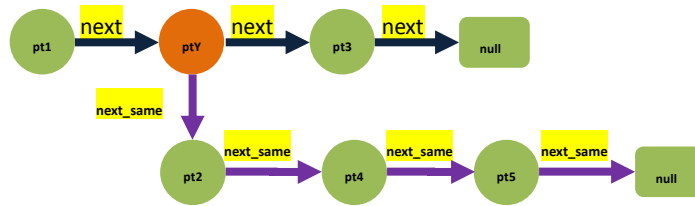
pattern	partno
pt1	n/a
pt2	2
pt3	n/a
pt4	3
pt5	4



after inserting ptY

for **nodeX**'s list:

ptY: partno=1 and other info are same as pt2



- c. if(ph_add_after) // will not go to here, returned at **return CL_SUCCESS**
 - a) pattern->next = ph_add_after->next;
 - b) ph_add_after->next = pattern;

3. test_ndb_sig3

node related to signature added before

4. test_ndb_sig1

- a. for(i = 0; i < len; i++) // loop over effective pattern, length is 3
- b. while(ph) // yes, ph = pt->list(has test_ndb_sig2->test_ndb_partsig)
 - a) // !ph_add_after is true
 - // ph->partno <= pattern->partno is true, both are 1
 - // !ph->next is true
 - if(!ph_add_after && ph->partno <= pattern->partno && (!ph->next || ph->next->partno > pattern->partno)) // true
 - ph_add_after = ph;
 - b) // ph->length == pattern->length is true, both are 3
 - // ph->prefix_length == pattern->prefix_length is true, both is 0
 - // no special char, so ch[0] and ch[1] are not changed after initialized
 - if((ph->length == pattern->length) && (ph->prefix_length == pattern->prefix_length) && (ph->ch[0] == pattern->ch[0]) && (ph->ch[1] == pattern->ch[1])) // true
 - i. if(!memcmp(ph->pattern, pattern->pattern, ph->length * sizeof(uint16_t)) && !memcmp(ph->prefix, pattern->prefix, ph->prefix_length * sizeof(uint16_t))) // true, double compare each char in effective pattern and prefix if there is
 - ii. // match=1 and do
 - iii. and since partno are same, so will do 'else' block in bellow code.

```

if(match) {
    if(pattern->partno < ph->partno) {
        pattern->next_same = ph;
        if(ph_prev)
            ph_prev->next = ph->next;
        else
            pt->list = ph->next;
        ph->next = NULL;
        break;
    } else {
        while(ph->next_same && ph->next_same->partno < pattern->partno)
            ph = ph->next_same;
        pattern->next_same = ph->next_same;
        ph->next_same = pattern;
        return CL_SUCCESS;
    }
}

```


in 'else' block, insert according the partno with inc sequence
and **return CL_SUCCESS** means will not add into list

after the whole process, the ac tire for "OOO" is:

```
ac_root
|-> 'o'
  |-> 'o'
    |-> 'o'->list(test_ndb_sig2)--next_same-->test_ndb_sig1--next_same-->test_ndb_partsig
                                          NULL <--next_same--|
```

===== **cli_ac_addpatt** =====

```
uint16_t len = MIN(root->ac_maxdepth, pattern->length);
// root->ac_maxdepth is set via CLI_DEFAULT_AC_MAXDEPTH
if(len < root->ac_mindepth) return CL_EMALFDB;
// root->ac_mindepth is set via CLI_DEFAULT_AC_MINDEPTH

for(i = 0; i < len; i++)
    next = pt->trans[(unsigned char) (pattern->pattern[i] & 0xff)];
    if(!next) // this tran does not yet exist
        next = (struct cli_ac_node *) mpool_malloc(root->mempool, 1, sizeof(struct
cli_ac_node)); // allocate
        newtable = mpool_realloc(root->mempool, root->ac_nodetable,
root->ac_nodes * sizeof(struct cli_ac_node *)); // allocate a new node table to
copy over the old ones and store the new one, copy over is done automatically via
mpool_realloc
        root->ac_nodetable = (struct cli_ac_node **) newtable;
        root->ac_nodetable[root->ac_nodes - 1] = next;
        // put into the tire
        pt->trans[(unsigned char) (pattern->pattern[i] & 0xff)] = next;
        pt = next // pt will be pointed at the newly allocated/already exist node
        // create new pattern table and copy over
        newtable = mpool_realloc(root->mempool, root->ac_pattable,
root->ac_patterns * sizeof(struct cli_ac_patt *));
        root->ac_pattable = (struct cli_ac_patt **) newtable;
        root->ac_pattable[root->ac_patterns - 1] = pattern;

        // pt now points at newly allocated/already exist ac node
        // ph points at pattern list for newly allocated/already exist ac node
        ph = pt->list; // the list only exists when this node is a leaf node(which stands for
last char of an effective pattern) and patterns in the list share same leaf node(which
also means same effective pattern(only 2 or 3 chars long))
        ph_add_after = ph_prev = NULL;
```

while(ph) // if leaf is shared by other patterns(which is highly possible as only first 2 or 3 bytes of the signature is effectively used to build the ac tire), then try to insert it to the shared pattern list(pt->list), also if the pattern or sub-pattern are same, should add into a structure called pattern->next_same

```
if(!ph_add_after && ph->partno <= pattern->partno && (!ph->next ||  
ph->next->partno > pattern->partno))
```

```
    ph_add_after = ph;
```

```
    // same pattern length, same prefix length and same first two letters
```

```
    // ending in same leaf, need to further confirm if the two pattern are same or  
similar
```

```
    if((ph->length == pattern->length) && (ph->prefix_length ==  
pattern->prefix_length) && (ph->ch[0] == pattern->ch[0]) && (ph->ch[1] ==  
pattern->ch[1]))
```

```
        // if the characters part of the two pattern are exact the same, compare  
other info in the signature
```

```
        if(!memcmp(ph->pattern, pattern->pattern, ph->length * sizeof(uint16_t))  
&& !memcmp(ph->prefix, pattern->prefix, ph->prefix_length * sizeof(uint16_t)))
```

```
            // if no other regex special case, the two sig are exact match
```

```
            if(!ph->special && !pattern->special) match = 1
```

```
            if(ph->special == pattern->special)
```

```
                //compare the special info
```

```
                a1 = ph->special_table[i];
```

```
                a2 = pattern->special_table[i];
```

```
            else match = 0;
```

```
            if(match) // sig info is the same
```

```
                // insert into next_same(same signature list) and sorting according
```

```
to partno
```

```
            if(pattern->partno < ph->partno)
```

```
                pattern->next_same = ph; // insert into same pattern list
```

```
                if(ph_prev) ph_prev->next = ph->next; // remove ph from the  
leaf node's pattern list since it is added into same pattern list of current pattern
```

```
                else pt->list = ph->next; // removing from current pattern's list
```

```
            else
```

```
                while(ph->next_same && ph->next_same->partno <  
pattern->partno)
```

```
                    ph = ph->next_same;
```

```
                pattern->next_same = ph->next_same;
```

```
                ph->next_same = pattern;
```

```
        else
```

```
            // try next pattern in the list
```

```
            ph_prev = ph;
```

```
    ph = ph->next;

    if(ph_add_after) // insert
        pattern->next = ph_add_after->next;
        ph_add_after->next = pattern;
    else // append in head
        pattern->next = pt->list;
        pt->list = pattern;
```

eqmcc@http://blog.csdn.net/eqmcc