ΠΡΟΣΤΑΣΙΑ ΚΑΙ ΑΣΦΑΛΕΙΑ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ
ΥΣ13 ΕΑΡΙΝΟ 2016

Project #3

Μέρες Καθυστέρησης 4 από 10

ΚΙΤΣΑΚΗΣ ΒΑΣΙΛΗΣ
ΜΔΕ Η/Α
ΑΜ: 2014509

Εισαγωγή

Στην εργασία αυτή μας ζητήθηκε να δημιουργήσουμε ένα rainbow table, ώστε να υλοποιήσουμε μια online- offline attack σε ένα service που υπάρχει στο sbox. Στο παραπάνω service στέλνεται μια σειρά από κωδικούς, οι οποίοι είναι hashed από την συνάρτηση κατακερματισμού Blake. Σκοπός μας είναι να δημιουργήσουμε ένα rainbow table με βάση την συνάρτηση blake, χρησιμοποιώντας μια κατάλληλη reduction function, και να βρούμε έναν τρόπο να υποκλέψουμε την μετάδοση του hash, από το σύστημα ενδοεπικοινωνίας των process, που εδώ είναι το D-Bus. Τα rainbow tables χρησιμοποιούνται ώστε να μπορέσουμε να μειώσουμε τον χώρο αποθήκευσης των κωδικών και των hashes. Αντί να αποθηκεύσουμε όλα τα hashes για όλους τους δυνατούς κωδικούς (μήκους 6 στην περιπτωσή μας, με 64 επιτρεπτούς χαρακτήρες,με 64^6 πιθανούς συνδυασμούς) , ξεκινάμε από μερικούς κωδικούς, και στην συνέχεια εφαρμόζουμε εναλλάξ την hash function και την reduction function. Στο τέλος αποθηκεύουμε μόνο τον αρχικό κωδικό και το τελικό hash. Στην περέπτωσή μας κάναμε αλλο ένα βήμα, στο τελικό hash εφαρμόσαμε άλλο ένα reduction για διευκόλυνση στην διαχείρισει τον δεδομένων που αποθηκεύουμε, και έτσι το rainbow table (η κάθε σειρά) ξεκινάει και τελειώνει σε plain-text (password). Τα παραπάνω αφορούν το κομμάτι offline της επίθεσης, μετά από την κατασκευή του rainbow table και κατά την ώρα της επίθεσης, αφού βρούμε το hash που θέλουμε να σπάσουμε αναζητούμε μέσα στον rainbow table για το hash.
Έτσι έχουμε κάτι τέτοιο
plain_start --H_fun-> H0 --R0->plain1--H_fun-> H1 –R1->plain1..--H_fun-> HN--RN->plain_last
plain_start --H_fun-> H0 --R0->plain1--H_fun-> H1 –R1->plain1..--H_fun-> HN—RN->plain_last
.
.
.
plain_start --H_fun-> H0 --R0->plain1--H_fun-> H1 –R1->plain1..--H_fun-> HN—RN->plain_last

Αρχικά ψάχνουμε αυτά που έχουμε αποθηκευμένα (τα plain_last) και αν δεν βρεθεί πρέπει να ψάξουμε μέσα στις αλυσίδες. Συγκεκριμένα ο αλγόριθμος είναι ο εξείς:

1. find plaint=plain_last if true break else
2. hash=hash_function (plain); plain=reduction_funcion(hash); (έχουν αυτή τη σειρά επειδή εμείς καταλήγουμε σε plain στο τέλος
3. go to 1.
4. if found get plain_start (βρήκαμε την σειρά που υπάρχει το hash που ψάχνουμε)
5. find pre-hash

Υποκλοπή Hashes

Η υποκλοπή των hashes με την εντολή dbus-monitor η οποία μας δίνει πληροφορίες για τα μηνύματα που ανταλλάσσουν οι διεργασίες πάνω από το d-bus.

Κατασκευή Rainbow Table

Η υλοποίηση έγινε σε c. Αρχικά πρέπει να φτιάξουμε τα plain_s για τα οποία βασιστίκαμε στον compiler της c να τα δημιουργείσει ψευδοτυχαία μέσα στα όρια τον επιτρεπτών συμβόλων. Έπειτα κάνουμε τα επιθυμητά hash-reduce και καταλήγουμε στα plain_last. Η πρώτη ιδέα ήταν να αποθηκεύονται σε αρχείο αλλά ήταν πολύ αργό και δεν μου άρεσε, έτσι αποφάσισα να τα αποθηκεύει στη stack. Με την εντολή ulimit -s μπορούμε να θέσουμε την stack σε ότι μέγεθος θέλουμε ώστε να χωρέσει ο πινακας που θα αποθηκεύσουμε τα plain_s , plain_last. Φτιάχνουμε ένα δισδιάστατο πίνακα με γραμμές τόσες όσες είναι οι αλυσίδες που επιθυμούμε να έχουμε και 12 στήλες, σε κάθε γραμμή τα πρώτα 6 είναι το plain_last και τα 6 τελευταία είναι το plain_start. Χωρίζουμε τον πίνακα σε N μέρει και N threads γράφουν παράλληλα στον πίνακα με την βοήθεια του openmp. Όταν δημιουργηθεί ο πίνακας τον ταξινομούμε με βάσει τα plain_last (δηλαδή τα πρώτα 6 στοιχεία) με την βοήθεια της qsort. Έτσι 'εχουμε ένα ταξινομημένο rainbow table. Δίνονται ενδεικτικά τα νούμερα για μία δοκιμή που δούλεψε με σχετικά καλή πιθανότητα, $80*10^6$ αλυσίδες με 1500 reductions για 4 threads χρόνος δημιουργίας 7 ώρες, χρόνος ταξινόμησης 5 δευτερα (περίπου), μέγεθος 916 MB. Να σημειωθεί επίσης ότι σαν όρισμα σύγκρισης στην qsort έχει μπει μια μικρή παραλλαγή της fast_compare που είναι μέχρι και 10 φορες γρηγορότερη απο την strcpy.

Χρήση του Rainbow Table

Στην πρώτη προσπάθεια για ψάξιμο, το rainbow table δεν ήταν ταξινομημένο και το search ήταν παράλληλη σειριακή αναζήτηση, χωριζόταν ο πίνακας σε N μέρη και N threads ψάχνανε, κάθε ένα σε ένα κομμάτι, σειριακά. Για το παραπάνω παράδειγμα ήθελε 3,5 λεπτά σε 8 threads! Μετά έγινε η δοκιμή με την ταξινόμηση. Πλέων το ψάξιμο δεν γίνεται παράλληλα, γίνεται όμως με binary search με την βοήθεια της bsearch. Για το παραπάνω παράδειγμα ήθελε σχεδόν ένα δευτερόλεπτο, πολύ μεγάλη βελτίωση. Μιας και ήταν τόσο γρήγορο δεν χρειαζόταν να γραφτεί η search παράλληλη. Το πρώτο βήμα είναι να κάνει μία αναζήτηση για το hash που μας έδωσε ο χρήστης. Το hash σαν είσοδος είναι 64 δεκαεξαδικοί χαρακτήρες που αντιπροσωπεύουν μια 32 BYTE λέξη. Έτσι ανα δύο τους μετατρέπουμε σε ακεραίους και τους βάζουμε σε πίνακα 32 θέσεων, έτσι ο πίνακας BitSequence διαβάζει τους αντίστοιχους ASCii χαρακτήρες και έχουμε ενα BitSequence hash[32] με το οποίο μπορούμε να δουλέψουμε. Ύστερα υλοποιούμε τον αλγόριθμο που βρίσκεται παραπάνω. Στην ουσία ξεκινάμε απο το τέλος, κανουμε reduction το hash μας και συγκρίνουμε το αποτέλεσμα με όλα τα τελευταία στοιχεία των αλυσίδων και συνεχίζουμε τα reduction-hash μέχρι να βρεθεί ή να τερματίσει το μήκος των

αλύσιδων. Αμα τερματήσει σημαίνει δεν βρέθηκε. Αμα βρεθεί σημαίνει ότι στην αλυσίδα υπάρχει το plain που είναι το prehash του hash που δώσαμε, έτσι για την συγκεκριμένη γραμμή υπολγίζουμε τα “ζευγάρια” plain->hash, και όταν πετύχουμε το hash μας κρατάμε το plain (που παραγει το hash μας) και σταματάμε τους υπολογισμούς. Στο πρόγραμμα έχει αποφευχθεί η χρήση του οτιδήποτε μπορεί να θεωρηθεί ακριβό, όπως πολλές χρήσεις συναρτήσεων μέσα σε επαναλήψεις (είναι σχεδον ένα μεγάλο main), αντικατάσταση συναρτήσεων της <string.h> με γρηγορότερες.
Επίσεις στην reductio-funcion στα ορίσματα της έχουμε βάλει και τον αριθμό της (θέση της στην αλυσίδα) τον οποίο χρησιμοποιεί στους υπολογισμούς, έτσι ώστε η καθε μια να είναι διαφορετική με την επόμενη και την προηγούμενη της αλλά ίδια με την απο πάνω και κατω της με σκοπό να μειωθούν οσο το δυνατόν περισότερο τα colisions. Αρχικά είχε χρησιμοποιηθεί η ίδια συνάρτηση παντού και σε μια αλυσίδα 1000 είχε μέχρι και 9 ίδια plaintexts!
Από διάφορες προσπάθειες επαληθεύτηκε οτι άμα πάμε σε μεγάλα νούμερα σε αριθμό αλυσίδων μεγαλώνει το μέγεθος αποθήκευσης του πίνακα, ενώ σε αριθμό reduction (μήκος) μεγαλώνει ο χρόνος επεξεργασίας (εκτέλεσης).
Μια βελτίωση που θα μπορούσε να γίνει, να παραλληλοποιηθεί το ψάξιμο έτσι ώστε να μπορέσουμε να διαχειριστούμε αλυσίδες μεγαλύτερου μήκους. Για 80εκ αλυσίδες και 4000 reductions το ψάξιμο θέλει 13 δευτερόλεπτα και δεν μας βολεύει στην περιπτωσή μας. Επίσεις θα μπορούσε να έχει υλοποιηθεί και συνθήκη εξόδου από το πρόγραμμα!
Ένα πρόβλημα που καθηστερεί το search δεν λύθηκε μιας και η ταχύτητά του με ικανοποίησε. Το προβλημα είναι λόγο collision να καταλήξουμε σε λανθασμένα σε ένα plain_last και το πρόγραμμα θα θεωρήσει ότι το plaintext υπάρχει στην αλυσίδα και θα πάει να την ψάξει, δεν θα το βρει φυσικά και θα την υπολογήσει όλη την αλυσίδα αυξάνοντας έτσι την πολυπλοκότητα του κώδικα. Για κάθε λάθος που βρίσκει αυξάνεται ο χρόνος αναζήτησης. Επειδή οι χρόνοι ήταν πάντα στα δευτερόλεπτα δεν το διόρθωσα.

Επίθεση

Για την επίθεση χρησιμοποιήθηκε rainbow table $80*10^6$ αλυσίδες με 2000 reductions η καθε μια. Η μνήμη που δεσμεύτηκε ήταν 945.7 MB χρόνος δημιουργίας 4 ώρες και 50 λεπτά με 8 threads, χρόνος ταξινόμησης 5 δευτερόλεπτα, χρόνος αναζήτησης στην χειρότερη 3 δευτερόλεπτα.
Το hash που βρέθηκε είναι το εξής
69a23839223a73ec8b25b1d6fd60b56d085dee047e5708eb318f0739c0992ff5
το οποίο προέρχετε από το εξής plaintext
migYzL
το οποίο μας έδωσε το εξής serial κατά την σύνδεσή μας
SERIAL:20160524170246-069593be52970780d9af53fb79301f16

Ακολουθουν στιγμιότηπα από της κονσόλες

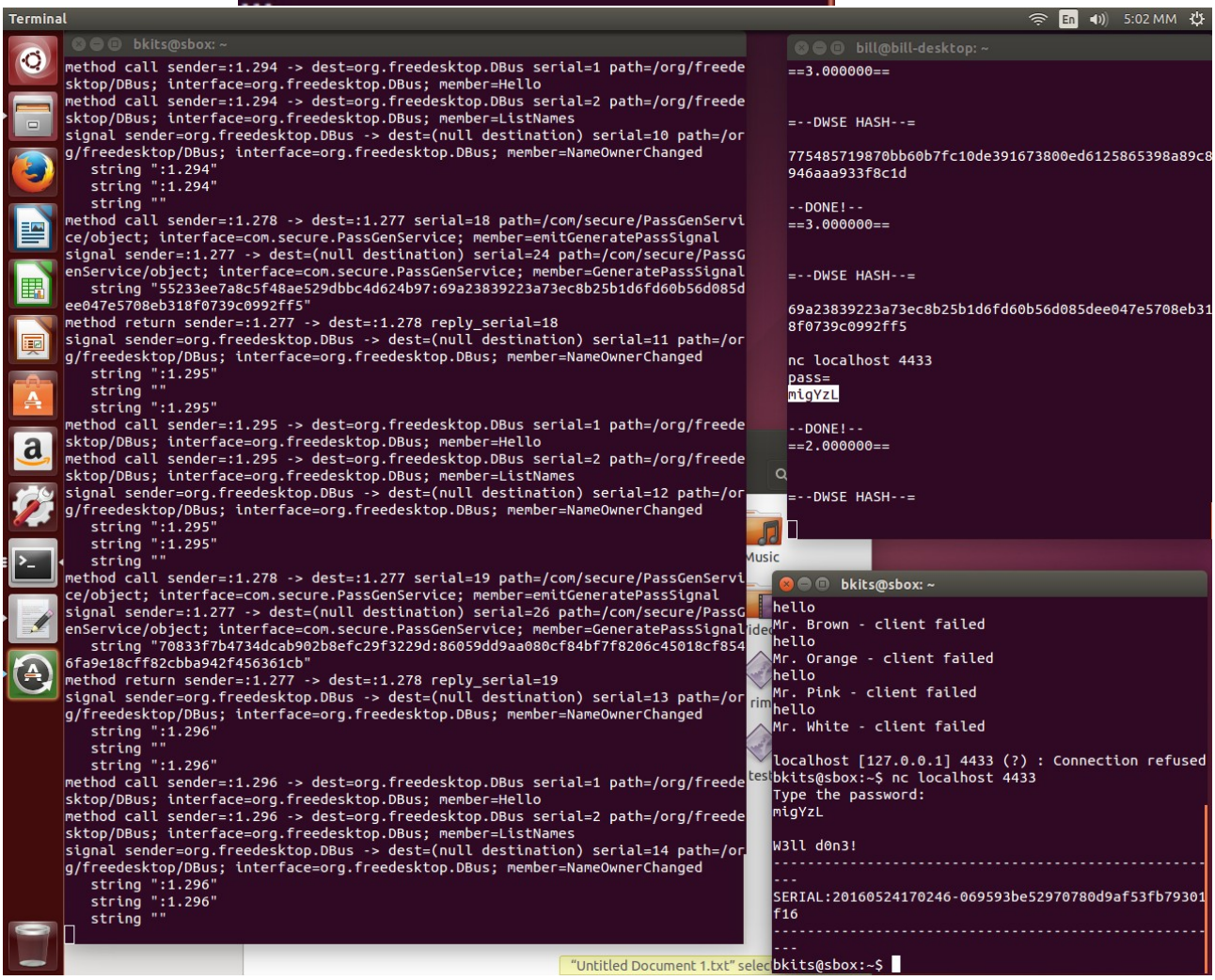
```
method call sender=:1.278 -> dest=:1.277 serial=18 path=/com/secure/PassGenServi
ce/object; interface=com.secure.PassGenService; member=emitGeneratePassSignal
signal sender=:1.277 -> dest=(null destination) serial=24 path=/com/secure/PassG
enService/object; interface=com.secure.PassGenService; member=GeneratePassSignal
   string "55233ee7a8c5f48ae529dbbc4d624b97:69a23839223a73ec8b25b1d6fd60b56d085d
ee047e5708eb318f0739c0992ff5"
=--DWSE HASH--=

69a23839223a73ec8b25b1d6fd60b56d085dee047e5708eb31
8f0739c0992ff5

nc localhost 4433
pass=
migYzL

--DONE!--
==2.000000==
localhost [127.0.0.1] 4433 (?) : Connection refused
bkits@sbox:~$ nc localhost 4433
Type the password:
migYzL

W3ll d0n3!
------------------------------------------------
---
SERIAL:20160524170246-069593be52970780d9af53fb79301
f16
------------------------------------------------
---
```



ΚΩΔΙΚΑΣ

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <stddef.h>
#include <omp.h>
#include "blake_ref.h"

#define LENGHT 2000
#define ROWS 80000000
static const char buffer[] =
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz012345678
9!@";

int hex_to_int(char c)
{if (c=='0') return 0;
 if (c=='1') return 1;
 if (c=='2') return 2;
 if (c=='3') return 3;
 if (c=='4') return 4;
 if (c=='5') return 5;
 if (c=='6') return 6;
 if (c=='7') return 7;
 if (c=='8') return 8;
 if (c=='9') return 9;
 if (c=='a') return 10;
 if (c=='b') return 11;
 if (c=='c') return 12;
 if (c=='d') return 13;
 if (c=='e') return 14;
 if (c=='f') return 15;
 if (c=='A') return 10;
 if (c=='B') return 11;
 if (c=='C') return 12;
 if (c=='D') return 13;
 if (c=='E') return 14;
 if (c=='F') return 15;
}

void R_R(BitSequence* plain, BitSequence* hash,int i)
{
int a,j;
for (a=0;a<6;a++)
```

```
{j=hash[a]+hash[2*a]+hash[a*a]+i+hash[i%32]+hash[16]+hash[3]+hash[30];
 plain[a]=buffer[j%64];
}

}

int fast_compare( BitSequence *ptr0, BitSequence *ptr1){
int len=6;
int fast = len/sizeof(size_t) + 1;
int offset = (fast-1)*sizeof(size_t);
int current_block = 0;
if( len <= sizeof(size_t)){ fast = 0; }
size_t *lptr0 = (size_t*)ptr0;
size_t *lptr1 = (size_t*)ptr1;
while( current_block < fast ){
if( (lptr0[current_block] ^ lptr1[current_block] )){
int pos;
for(pos = current_block*sizeof(size_t); pos < len ; ++pos ){
if( (ptr0[pos] ^ ptr1[pos]) || (ptr0[pos] == 0) || (ptr1[pos] == 0) ){
return (int)((unsigned char)ptr0[pos] - (unsigned char)ptr1[pos]);
}
}
}
++current_block;
}
while( len > offset ){
if( (ptr0[offset] ^ ptr1[offset] )){
return (int)((unsigned char)ptr0[offset] - (unsigned char)ptr1[offset]);
}
++offset;
}
return 0;
}

int plain_comp(BitSequence* plain1, BitSequence* plain2)
{int g;
 for (g=0;g<6;g++)
  {if (plain1[g]!=plain2[g]) return 1;}

return 0;
}

int hash_comp(BitSequence* hash1, BitSequence* hash2)
{int g;
 for (g=0;g<32;g++)
```

```
  {if (hash1[g]!=hash2[g]) return 1;}

return 0;
}


int main(int argc, char **argv)
{
int i,k,x,a,r,in_int[64],rr,aa,index,found=0;
char input[65];
FILE *f;
time_t start,end,tt;
double dur;
BitSequence
hash_gn[32],hash[32],plain[6],plain_par[6],plain_l[6],plain_s[6],*res,*p_r_t,
r_t[ROWS][12]
//*r_t
;
omp_set_num_threads(8);
//r_t=(BitSequence*)malloc(sizeof(BitSequence)*ROWS*12);
srand((unsigned) time(&tt));

start=time(NULL);

#pragma omp parallel
{int n,id,r;int i,k;BitSequence hash[32],plain[6],plain_s[6],plain_l[6];
id=omp_get_thread_num();n=omp_get_num_threads();

for (r=id;r<ROWS;r=r+n)
{
//line_create(r_t,r);

for (i=0;i<6;i++)
{plain[i]=buffer[rand()%64];
r_t[r][i+6]=plain[i];}

for (k=0;k<LENGHT;k++)
{
Hash(256,plain,48,hash);
R_R(plain,hash,k);
}

for (i=0;i<6;i++)
r_t[r][i]=plain[i];
```

```
}//rows end
}//pragma end

end=time(NULL);
dur=difftime(end,start);
printf("==%lf==\n\n",dur);

qsort(r_t,ROWS,sizeof(BitSequence)*12,fast_compare);
/*
for (i=0;i<12;i++)
  {if (i==6) printf(",");
  printf("%c",r_t[ROWS-3][i]);}
printf("\n");
//printf("\n");

for (k=0;k<ROWS;k++)
{
for (i=0;i<12;i++)
  {if (i==6) printf(",");
  printf("%c",r_t[k][i]);}
printf("\n");
}

*/

x=1;
while (x==1)
{
printf("\n=--DWSE HASH--=\n\n");

fgets(input,65,stdin);
getchar();
start=time(NULL);
x=0;
for (i=0;i<64;i=i+2)
{in_int[x]=16*hex_to_int(input[i])+hex_to_int(input[i+1]);
 x++;}

for (i=0;i<32;i++)
hash_gn[i]=in_int[i];

//#pragma omp parallel
//{int n,id,r;int i,a,rr,found,rrr;BitSequence
```

```
plain_l[7],plain_s[6],plain[7],hash[32],plain_gn[6];
//id=omp_get_thread_num();n=omp_get_num_threads();

//for (r=id;r<ROWS;r=r+n){start for ROW
//strncpy(hash,hash_gn,32);

for(a=(LENGHT-1);a>=0;--a)
{for (i=0;i<32;i++)
 hash[i]=hash_gn[i];

 for(rr=a;rr<LENGHT;rr++)
 {R_R(plain,hash,rr);
  Hash(256,plain,48,hash);}
/*
for (i=0;i<6;i++)
 plain_par[i]=plain[i];


#pragma omp parallel
{int n,id,r,i,aa,found2;BitSequence plain_l[6],plain_s[6],plain[6],hash[32],*res;
id=omp_get_thread_num();n=omp_get_num_threads();
 for (i=0;i<6;i++)
 plain[i]=plain_par[i];
 //for(r=0;r<ROWS;r++)
*/

  //for (i=0;i<6;i++)
  // plain_l[i]=r_t[r][i];
p_r_t=&r_t[0][0];
res=(BitSequence*)bsearch(&plain,r_t,ROWS,sizeof(BitSequence)*12,fast_compare
);
    index=(res-p_r_t)/12;

  if (res!=0)
  {for (i=0;i<6;i++)
     {plain_s[i]=r_t[index][i+6];plain_l[i]=r_t[index][i];//printf("%c",plain_s[i]);
     }
  // printf(",0,");
 //  for (i=0;i<6;i++)
 //  printf("%c",plain_l[i]);
 //  printf("\n");
  // found=1;break;

strncpy(plain,plain_s,6);
Hash(256,plain,48,hash);
```

```
aa=0;
for (aa=0;aa<LENGHT;aa++)
{
if (hash_comp(hash,hash_gn)==0)
{printf("\nnc localhost 4433\n");
printf("pass=\n");
for (i=0;i<6;i++)
printf("%c",plain[i]);
printf("\n");found=2;
break;
}

Hash(256,plain,48,hash);
R_R(plain,hash,aa);
Hash(256,plain,48,hash);

}

  }// if isa



//}//pragma end


if(found==2) break;
}// without if found break

x=1;found=0;
printf("\n--DONE!--\n");


end=time(NULL);
dur=difftime(end,start);
printf("==%lf==\n\n",dur);

}


return 0;
}
```