# *Project #3*
# *ΥΣ13 ΕΑΡΙΝΟ 2016*

*Αλέξανδρος Λαποκωνσταντάκης*
*1115201200088*
*(2)*

Κοιτώντας τα αντίστοιχα tutorials για το D-BUS έφτιαξα το απλό python script intercept.py, το οποίο λάμβανε όλα τα μηνύματα του session πάνω στο dbus, χωρίς περιορισμούς σε interface, signal names κτλ, αφού μου είναι άγνωστα,  και το έτρεξα στο sbox.

Κάποια στιγμή το πρόγραμμα έπιασε τα ακόλουθα 12 hashes, απ'ό,τι φαίνεται είναι οι hashed κωδικοί πάνω στους οποίους θα πρέπει να πειραματιστώ με το rainbow table.

grabbed hash
c8c1588eedf7e40f1069c03dc92955a9:a748968488c9d3468fe17c150ffd5ada577af040063236a
724d304d522abf9d6
grabbed hash
99e6a6e9c28f27c022c6944ab1bab3b7:21c9c2be188e79fb68603d8f76c821f8b26c1736fe04db6
59517772db6da9a78
grabbed hash
1f055543d76fa6bf54767c372df76c74:f0088407a582dd2a5ea2a87f3eb5dfdf69d12d7965836af7
151076039b6b82ec
grabbed hash
d101e8f618e4e0d9a3996ca131dedca8:2bba574a450b75db90c4b111d94dac83f2265db65a2919
eba11edb0d165ce540
grabbed hash
0d012a6c3964eadf0cb02a92eb2a1cdd:56ac639337b0870141f1cdb93cec1cb8a6e979c811b5480
d31818d9e29ed16f7
grabbed hash
4cc1320abfc8142e032631f4ec52b559:a4bc37ea6e3e62dd59049824af5ec9e647eed6b878ea1a0
baf41ff6ba92a50eb
grabbed hash
ce726cc15f2936e91a6ecbfcd1ce3264:f6e395fd19313780b6b64837ade6d85a5090f3e6b3c1137
c3c4b3c1c9f3850d9
grabbed hash
629fce1722e7bb2141406510d724cb04:c1da9474dc02c8d53264e4a590c1485df0ed3564952fa0
a32ce55e51d53f86b0
grabbed hash
9b716be7c184ad82e8245393d6a81230:653df9c2391d75e7286fe6e57d74b7fe0aff1a90cd7ccd6
a511fa829c59ed6d1
grabbed hash
32b677b13c29aada4fdf74d4454123de:65fcd2a8405740337705c2cd8758e7e656bea42fef9a8e5
25ec31688827e6235
grabbed hash
584e0476340ded01f60af3595cd4088b:b684dc9f4aae366c8523430c4a94a5948e12daa6741b3f1
feb69d70b0fd4e3a2
grabbed hash
648059edb8e24b1030718e6479650176:b14a2b24f9e701005a2cdd2d441aab0cd656a16494a12
dc515d297ec80b04802

Φαίνεται ότι σε κάθε signal που έλαβε το script μέσω του D-BUS περιέχονται 2 hashes, χωρισμένα με το : . Το ένα έχει 32 ψηφία και το άλλο 64.

Ο αλγόριθμος που θα χρησιμοποιήσω για τη δημιουργία του rainbow table είναι ο ίδιος που χρησιμοποιείται για τα hashes που έπιασα, δηλαδή ο BLAKE.

Ωστόσο εμφανιζόταν στο terminal το error:

```
Traceback (most recent call last):
  File "/usr/lib/python2.7/dist-packages/dbus/connection.py", line 230, in
maybe_handle_message
    self._handler(*args, **kwargs)
TypeError: msg_handler() takes exactly 1 argument (3 given)
```

Ενώ ο handler είχε τη μορφή

```
def msg_handler(msg):
        print "session got hash " + msg
```

, δηλαδή δεχόταν ένα όρισμα, το signal που λάμβανε το service είχε 3, άρα έπρεπε να τον τροποποιήσω. Η στατική τροποποιήση είχε παρόμοια αποτελέσματα, δηλαδή αν δοκίμαζα

```
def msg_handler(msg, arg1, arg2):
        print "session got hash " + msg
```

εμφανιζόταν ότι λάμβανα μόνο 1 argument αντί για  3, οπότε χρησιμοποίησα τη “δυναμική” μορφή κλήσης:

```
def msg_handler(*args):
          print "intercepted "
          for arg in args:
                    print arg
```

Μ' αυτόν τον τρόπο θα μπορούσα επίσης να δω όποια άλλα arguments υπάρχουν και έτσι να συγκεκριμενοποιήσω τον signal handler. Έτσι, μεταξύ άλλων signals, έλαβα το com.secure.PassGenService που μοιάζει να είναι το bus name ή dbus_interface που χρησιμοποιείται για τη μεταφορά του hash. Δοκίμασα να κάνω:

```
bus.add_signal_receiver(msg_handler, dbus_interface = "com.secure.PassGenService")
```

αντί για το bus.add_signal_receiver(msg_handler)

Τώρα τα μόνα signals που εμφανίζονταν ήταν τα hashes, δηλαδή αυτά που μ' ενδιέφεραν.

---

**Rainbow Table**

Αποφάσισα να υλοποιήσω το rainbow table και την επίθεση σε python, έχοντας κοιτάξει παρόμοιες υλοποιήσεις και script. Γι' αυτό χρειαζόμουν την υλοποίηση της BLAKE και μια δική μου reduction function. Το rainbow table θα είναι N chains, που η καθεμία σε κάθε τμήμα της θα χρησιμοποιεί διαφορετική reduction function, προφανώς η καθεμία θα 'ναι παραλλαγή της αρχικής. Δηλαδη ο πρώτος κρίκος της θα είναι το starting_point, ο επόμενος θα προκύψει από την R1 kai blake_hash, ο επόμενος από την R2 kai blake_hash, μέχρι τον τελευταίο που θα είναι το blake_hash του password που έχει προκύψει απ την Rk function. Κάθε αλυσίδα θα αρχίζει με διαφορετικό password, που θα προκύπτει τυχαία απ' τους χαρακτήρες που χρησιμοποιούνται και θα έχει μήκος 6 χαρακτήρων.

Το hash που περνάται για αναζήτηση στο rainbow table ακολουθεί τη διαδικασία:
Ελέγχεται αν το hash είναι κάποιο απ' τα endpoints του rainbow table, αν είναι το password βρέθηκε. Αλλιώς, γίνεται reduced πίσω στους πιθανούς κωδικούς, με τα επιτρεπτά ψηφία και μήκος 6, και ξανα-hashάρεται, για να ελεγχθεί ξανά αν το νέο hash είναι ένα απ τα endpoints του rainbow table. Το ίδιο επαναλαμβάνεται μέχρι να βρεθεί το hash, και η διαδικασία σε κάθε βήμα της χρησιμοποιεί το αμέσως “προηγούμενο” reduction function. Δηλαδή ξεκινάει με το Rk, και περνάει απ τα Rk-1, Rk-2, … μέχρι το R0, οπότε και το password τελικά δε βρέθηκε. Αν όμως το hash βρεθεί σε κάποιο απ' τα βήματα, παίρνω το starting_point που αντιστοιχεί στο hash-end_point, και ανακατασκευάζω την αλυσίδα με κλήσεις hash-R1-hash-R2-…, μέχρι να συναντήσω και πάλι το hash, οπότε ο ζητούμενος κωδικός θα'ναι αυτός που προηγείται του hash.

Reduction Function

Ιδανικά το αποτέλεσμα της reduction function, δηλαδή η συμβολοσειρά που θα ανήκει στο πεδίο τιμών των πιθανών κωδικών (6 χαρακτήρες απ' τους συγκεκριμένους δοσμένους), θα πρέπει να εξαρτάται απ' όλα τα ψηφία του hash. Η διαδικασία που ακολούθησα για να φτάσω μέχρι αυτήν την κατάσταση, εως κάποιο βαθμό, ήταν:

Ο αριθμός των πιθανών χαρακτήρων που μπορεί να περιέχει το κάθε ψηφίο του ζητούμενου κωδικού 64 συνολικά (64^6 πιθανοί κωδικοί). Έστω ότι το hash που παράγει η BLAKE είναι 64 bit (η ίδια διαδικασία θα ισχύει και για τα 32 bit αναλογικά). Δημιουργώ έναν πίνακα (όνομα aux) από structs, το οποίο αντιστοιχεί κάθε ένα απ' τα πιθανά σύμβολα με ένα αριθμό απ' το 0 έως το 63 (64 σύμβολα).
Πχ a : 0, b : 1, … ! : 62, @ : 63, έτσι μετά απ’ τις όποιες πράξεις – υπολογισμούς κάνω το αποτέλεσμα mod 64 και το αντιστοιχίζω με την τιμή του.
Έστω p η συμβολοσειρά που παράγει η R(reduction) function και h το hash που παίρνει ως όρισμα και p[0], p[1], … , p[5] , h[0], h[1], …, h[63] οι χαρακτήρες τους αντίστοιχα.
Πρώτη ιδέα ήταν να κάνω generate ένα random p για αρχή και ύστερα να αθροίζω όλα τα ψηφία του h **(ουσιαστικά τους ASCII κωδικούς τους)** και το άθροισμα να το διαιρώ με κάθε p[i], i=0, .., 5. Τον αριθμό που προέκυπτε θα τον έκανα %64 και ανάλογα με το αποτέλεσμα, το

pi θα έπαιρνε το σύμβολο που αντιστοιχεί σ'αυτόν με βάση τον πίνακα aux που έφτιαξα. Προφανώς αν το έκανα έτσι θα προέκυπτε ένα p με χαρακτήρες που μεταξύ τους θα είχαν την ίδια “αντιστοιχία” που θα είχε και το random string που έκανα generate, οι επόμενες πράξεις δεν άλλαζαν κάτι επι της ουσίας στη διαδικασία παραγωγής του string(σχεδόν ίδιο με brute force), εκτός του ότι οι νέοι χαρακτήρες θα ανήκαν στους επιτρεπτούς:

(ψευδοκώδικας με λίγη python)

```
p = rand(possible characters, length 6)
for i in range(0, 64)
        sum += h[i]
p[i] = (sum / p[i]) % 64
```

Μια βελτίωση αυτού ήταν να χωρίσω το h σε 6 groups, αντιστοιχώντας ένα σε κάθε ψηφίο του ζητούμενου p (για 64 bit έχω 64 = 5 * 12 + 4). Η διαδικασία που ακολούθησα ήταν ίδια με πριν, αυτη τη φορά όμως υπολόγιζα το άθροισμα του κάθε group.

```
gn = 6 #number of groups
For j in range(0, gn) #for each group in the hash
        group_sum[j] = 0
        for h[i] in g[j]: #for each hash character in this group
                group_sum += h[i]
        end
end
for j in range(0, 6):
        p[i] = group_sum[j]%64
```

Αυτή τη φορά το randomization είναι λίγο μεγαλύτερο, με τον κάθε χαρακτήρα να διαφοροποιείται απ τους υπόλοιπους ανάλογα με το άθροισμα της ομάδας χαρακτήρων του hash που του αντιστοιχούν. Πάλι όμως πρόκειται για πολύ μικρό randomization αφού το κάθε ψηφίο του group δεν κάνει μεμονωμένα κάποια διαφορά, περά απ τη συμμετοχή του στο άθροισμα (πχ αντιμεταθέτοντας 2 ψηφία σ' ένα group το p παραμένει το ίδιο).

Το θέμα είναι η ακριβής θέση των ψηφίων να επηρεάζει το τελικό αποτέλεσμα, άρα θα πρέπει να προσθέσω πράξεις μεταξύ τους και διαφοροποιήσεις ανάλογα με τη θέση τους. Έτσι μετά τον υπολογισμό των αθροισμάτων μπορώ να προσθέσω και στο κάθε group, έστω g[i] τα μέλη του:

```
for(i=0;  i<6; i++){
                for(j=0; j < shift; j += 2){
                        long transit;
                        pair[i][m] = bitXor((g[j] << 1), g[j+1]) << (m / 2);
                        hyp_sum[i] += pair[i][m];
                }
                m = 0;
```

```
        }
for(i=0; i<6; i++)
        p[i] = (hyp_sum[i] XOR group_sum[i]) % 64
```

Τώρα η hyp_sum περιέχει για κάθε group ένα άθροισμα του τύπου
[ (h[j] << 1) XOR h[j+1] ] << i  + … + [ (h[n] << 1) XOR h[n+1] ] <<  n
Μ’ αυτόν τον τρόπο το αποτέλεσμα εξαρτάται πλέον και απ’ τη θέση του κάθε ζεύγους και τη θέση του κάθε χαρακτήρα στο ζεύγος.

Άρα έχει επιτευχθεί τουλάχιστον μια R που εξαρτάται απ' το κάθε ψηφίο του hash, ανεξάρτητα απ' το πόσο ισχυρή είναι πραγματικά τελικά. Για να φτιάξω πολλές reduction functions βασισμένες σ' αυτήν, που θα παράγουν λίγο διαφορετικά αποτελέσματα ανάλογα με κάποια παράμετρο (R1, R2, … ,Rk), για μείωση των collisions, θα περνάω μία επιπλέον παράμετρο k, που θα χρησιμοποιείται και πάλι μόνο στο τέλος, σε πρόσθεση και σε XOR, για να έχει μεγαλύτερο αντίκτυπο.

```
for(i=0; i<6; i++)
        p[i] = ( ( (hyp_sum[i] + k) XOR group_sum[i]) XOR k ) % 64
```

---

Αντί να χρησιμοποιήσω την υλοποίηση της blake σε python, αποφάσισα να χρησιμοποιήσω τη C wrapper έκδοσή της, καθώς όπως λέει έχει περίπου 40πλάσια ταχύτητα. Το πρόβλημα ήταν να μπορέσω να κάνω import το blake_wrapper.py. Έγραψα κώδικα σε python για ευκολία , αλλά δεν μπόρεσα να κάνω link το blake_ref.c και blake_ref.h ως library για να χρησιμοποιήσω το blake_wrapper.py.  Προσπαθούσα να το τρέξω σε windows, κι έτσι χρησιμοποιούσα τον DMC compiler για να φτιάξω τη libblake ως .dll (/dmc/bin/dmc -WD blake_ref.c) αλλά καμία απ’ τις εκδόσεις της python (32bit, 64bit) δεν την αναγνώριζε.  Το libblake.dll αρχείο φτιαχνόταν σύμφωνα με το αρχείο blake_ref.def που δημιουργούσα αλλά αν και ακολουθούσα τη διαδικασία για να φτιάξω το library, στην εκτέλεσή του το python script (όταν έτρεχα python rainbow.py ab2gd!) που το έκανε import εμφάνιζε:
Windows Error [Error 193] %1 is not a valid windows application
Τελικά δεν έβγαλα άκρη με python και η C θα 'ταν έτσι κι αλλιώς γρηγορότερη, θα μπορούσα να χρησιμοποιήσω και threads για το rainbow table, οπότε μετέφερα όλο τον κώδικα σε C. Το πρόβλημα να κάνω link το library για το blake_wrapper.py βέβαια παρέμεινε. Δοκίμασα να το κάνω compile σε Dev-C++ κάνοντας link το libblake.so όμως ούτε αυτό δούλεψε, ούτε με cygwin και εκτελώντας manually την εντολή gcc -O3 -shared -o libblake.so blake_ref.c μέσω cmd.

Δεν μπορούσα να τρέξω το πρόγραμμα στο sbox ή linux της σχολής επειδή, πέρα απ’ το ότι θα δημιουργούσε φόρτο, δεν είχα και root permission για να κάνω install το library και να το προσθέσω στο κατάλληλο path. Είπα να το τρέξω σε linux μέσω VMware player αλλά στο PC μου μπορούσα να του δόσω μόνο 2GB μνήμη, και δεν μπορούσε να εκτελέσει απλές λειτουργίες (όπως ν’ ανοίξει το firefox σε λιγότερο απο 15 δευτερόλεπτα) οπότε είχε ελάχιστη ισχύ για να φτιάξει το rainbow table offline σε ρεαλιστικό χρόνο.

Τα αρχεία που περιέχονται είναι τα:
**rainbow.c rainbow.h intercept.py**
(τα rainbow για το rainbow table και το intercept για το dbus)

Το rainbow.c δέχεται ως παραμέτρους το hash που θέλω να σπάσω, τον αριθμό των chains και το length τους, υποστηρίζει hash οποιουδήποτε μεγέθους (πχ 32, 64) ενώ υπάρχει μια test random “hash” (char* H) σε σχόλια για δοκιμή του randomization της R .