An Executable Model for JFKr

An ACL2 approach to key-establishment protocol verification

Presented by: David Rager
November 22, 2004
About ACL2

- J S Moore and Matt Kaufmann
- ACL2 - A Computational Logic for Applicative Common Lisp
- Applicative – functional
- Hierarchical
  - Prove lemmas to prove larger theorems
- Doubles as an executable model
Diffie-Helman

- Model based off a state containing a list of “knowledge”
- From “knowledge” compute diffie-helman components
- From diffie-helman components, show:
  - Forward secrecy - yes
  - Authentication - no
  - Replay Attack - yes
  - DoS - no
  - ID Protection - no
Basic DH Model Parts

1. If an actor has neither of the two nonces used in a DH computation, it can not derive the DH key.
2. An intruder does not begin with either of the two relevant nonces.
3. Nonces are not released during the communication (requires induction).
4. Since nonces are not released during communication and the intruder has neither of the two relevant nonces, the intruder can not obtain the DH key.

- Should be easy
Knowledge-state Attempt #1

- Contained redundant data
- Should trim to basic building blocks

'((A (nonce a)
  (exp (nonce g) (nonce a))
  (exp (nonce g) (nonce b))
  (exp (nonce g) (mult (nonce a) (nonce b))))

(B (nonce b)
  (exp (nonce g) (nonce a))
  (exp (nonce g) (nonce b))
  (exp (nonce g) (mult (nonce a) (nonce b))))

(I (exp (nonce g) (nonce a))
  (exp (nonce g) (nonce b))))
Knowledge-state Attempt #2

'( ;; I is for initiator
  (I (nonce x)
    (base g)
    (mod b)
    (term (mod (exp g y) b)))))

;; R is for Responder
(R (nonce y)
  (base g)
  (mod b)
  (term (mod (exp g x) b)))))

;; N is for intruder
(N (base g)
  (mod b)
  (term (mod (exp g x) b))
  (term (mod (exp g y) b)))))

- The need to move forward preempted developing this more

November 29, 2004
Onto JFKr

\[ N_i, x_i \]

If initiator knows group \( g \) in advance

\[ x_i = g^{d_i} \]

DH group

\[ t_r = \text{hash}_K(x_r, N_r, N_i, IP_i) \]

Derive a set of keys from shared secret and nonces

\[ x_r^d = x_r^d = x \quad K_{a,e,v} = \text{hash}_x(N_r, N_i, \{a, e, v\}) \]

"Hint" to responder which identity to use

\[ e_i = \text{enc}_{K_e}(ID_i, ID_r', sa_r, \text{sig}_{K_r}(N_r, N_i, x_r, x_i, g_i)) \]

\[ h_i = \text{hash}_{K_a}^{('i',e_i)} \]

Check integrity before decrypting

\[ e_r = \text{enc}_{K_e}(ID_r, sa_r, \text{sig}_{K_r}(x_r, N_r, x_i, N_i)) \]

\[ h_r = \text{hash}_{K_a}^{('r',e_r)} \]

Real identity of the responder

[ Aiello et al. ] and Shmatikov

November 29, 2004
JFKr Properties

- Secrecy (via DH like key agreement)
- Prevention of DoS (via cookie)
- Forward secrecy (use of nonce for randomization)
- Authentication (public/private key signatures)
- ID Protection (ID not revealed until protocol nearing completion)
What I Did

- Executable model
- 1200 lines of ACL2 code
- Relatively low global constant usage
Capturing Perfect Cryptography

First idea:

- Make all knowledge broadcast in previous transmissions a constant $> 100$
- Make all private knowledge $< 100$
- Show that the attacker never gains something $< 100$
Capturing Perfect Cryptography

- Second idea:
  - Assign a probability threshold for what is acceptable
  - Requires tracking probability of a “crack” cumulatively
  - Could use to show mathematical weakness in encryption schemes
Capturing Perfect Cryptography

Third idea:
- Create generic functions “encapsulated” together with some other functions
- These functions don’t begin with any definitions
- Through instantiation of the function set, can prove that there exist functions that satisfy certain properties
- Useful for specifying mysterious properties like keyed-hash and digital signatures
- Can specify “perfect” encryption

None of these ideas were actually used by me
Capturing Perfect Cryptography

My Version:
- Computed a hash of an integer list:
  
  ```lisp
(defun compute-hash (int-list key acc)
  (if (atom int-list)
      acc
      (compute-hash (cdr int-list) key
                    (* key (+ acc (car int-list)))))))
  ```

- Signatures similar
- Symmetric keys modeled with one + key value
  - Encryption was an addition operation and decryption was a subtraction operation
- Asymmetric keys modeled with +/- key values
  - Encryption and decryption were same
  - + for encryption
  - - for decryption
Main Function

- Uses “honest” participants
- Chose to make it an explicit calling of recursion instead of a case statement

(defun run-honest (network-s initiator-s responder-s)
  (mv-let
    (network-s-after-1 initiator-s-after-1)
    (step1-honest network-s initiator-s)
    (mv-let
      (network-s-after-2 responder-s-after-2)
      (step2-honest network-s-after-1 responder-s))
    ...
  )
)

... left out part so it would fit ...

(mv-let
  (network-s-after-5 initiator-s-after-5)
  (step5-honest network-s-after-4 initiator-s-after-3)
  (mv-let
    (network-s-after-6 responder-s-after-6)
    (step6-honest network-s-after-5 responder-s-after-4)
    (mv network-s-after-6
      initiator-s-after-5
      responder-s-after-6)))

November 29, 2004
A Simple Honest Step Function

; doesn't matter who responder is
(defun step1-honest (network-s my-s) ; doesn't matter who responder is
  (let* ((Ni (nonce-mine my-s))
      (Xi (compute-public-dh-value
          *g*
          (dh-exponent my-s)
          *b*))
    ;; updates are alists
    (network-update (list (cons 'Ni Ni)
                           (cons 'Xi Xi)
                           (cons 'Src-ip (ip my-s))))
    (my-update
     (list (cons 'cost-cpu (+ (cost-cpu my-s) 1))
           (cons 'cost-mem (+ (cost-mem my-s) 1))
           (cons 'public-dh-value-mine Xi)
           (cons 'role 'initiator))))
  (mv
   (acons 1 network-update network-s)
   (append my-update my-s))))
A Simple Dishonest Step Function

(defun step1-dishonest (network-s my-s) ; doesn't matter who responder is
  (let* ((Ni -1)
         (Xi -1)
         (Src-ip (ip my-s))
         ;; updates are alists
         (network-update (list (cons 'Ni Ni)
                                (cons 'Xi Xi)
                                (cons 'Src-ip Src-ip)))
         (my-update (list (cons 'cost-cpu (+ (cost-cpu my-s) 0))
                          (cons 'cost-mem (+ (cost-mem my-s) 0))))))
  (mv
    (acons 1 network-update network-s)
    (append my-update my-s))))
A More Complex Step Function…

;;; responder is processing and sending message
(defun step2-honest (network-s my-s)
  (let* ((Ni (ni-msg1 network-s))
         (Nr (nonce-mine my-s))
         (Src-ip (Src-ip-msg1 network-s))

  
  ;;(Xr (xi-msg1 network-s))

  (Xr (compute-public-dh-value *g* (dh-exponent my-s) *b*))

  (Gr *g*)

  (Tr (compute-hash (list Xr Nr Ni src-ip)
                 (private-key my-s)
                 0))

  (network-update
   (list (cons 'Ni Ni)
         (cons 'Nr Nr)
         (cons 'Xr Xr)
         (cons 'Gr Gr)
         (cons 'Tr Tr)
         (cons 'src-ip (ip my-s)))))

;;; no real update to my state, since I'm throwing it away - stateless
(my-update
  (list (cons 'cost-cpu (+ (cost-cpu my-s) 2))
         (cons 'cost-mem (+ (cost-mem my-s) 0))
         (cons 'public-dh-value-mine Xr)))))

November 29, 2004
Example

Script:

(defconst *initiator-beg-state*
  (list
   (cons 'nonce-mine *initiator-nonce*)
   (cons 'dh-exponent
         *initiator-dh-exponent*)
   (cons 'ip *initiator-ip*)
   (cons 'private-key *initiator-private-key*)
   (cons 'cost-cpu 0)
   (cons 'cost-mem 0)
   (cons 'id-mine *initiator-id*)
   (cons 'sa-mine *initiator-sa*)))

(step1-honest nil *initiator-beg-state*)

RESULTS:

(((1 (NI . 200)
    (XI . 202942)
    (SRC-IP . 100001)))
 ((COST-CPU . 1)
  (COST-MEM . 1)
  (PUBLIC-DH-VALUE-MINE . 202942)
  (ROLE . INITIATOR)
  (NONCE-MINE . 200)
  (DH-EXPONENT . 5091)
  (IP . 100001)
  (PRIVATE-KEY . 12481)
  (COST-CPU . 0)
  (COST-MEM . 0)
  (ID-MINE . 50)
  (SA-MINE . 412)))
(defun run-dos-interleaved (network-s initiator-s responder-s)
  (mv-let
    (network-s-after-1 initiator-s-after-1)
    (if (evenp (random-int))
      (step1-dishonest network-s initiator-s)
      (step1-honest network-s initiator-s))

    (mv-let
      (network-s-after-2 responder-s-after-2)
      (step2-honest network-s-after-1 responder-s)
      (mv network-s-after-2 initiator-s-after-1 responder-s-after-2)))))
Mem DoS Thm

(defun run-dos-interleaved-mem-safe-generic
  (implies (and (beginning-statep initiator-s)
                (beginning-statep responder-s))
           (mv-let (network init resp)
                    (run-dos-interleaved nil initiator-s responder-s)
                    (declare (ignore network))
                    (and (<= (cost-mem resp)
                             (cost-mem init))))))

:hints ("Goal" :in-theory (disable
                          DH-EXPONENT
                          ID-MINE
                          IP
                          NONCE-MINE
                          PRIVATE-KEY
                          SA-MINE)))

Currently proves in 60 seconds.

November 29, 2004
Future Work

- Model man in the middle attack
- Form encapsulate functions for perfect encryption
- Prove termination of the protocol
- If JFKr terminates for both parties, then they agree on the key
- One party terminates iff the other party terminates
Conclusions

- Showed safety from memory DoS attacks
- Separation of actor states allows better inductive reasoning
- ACL2 proficiency increased
Resources

- Davis, Jared for ACL2 help.