Knowing Just Enough Crypto to be Dangerous

Vasilij Schneidermann

April 2018

Outline

Section 1

Intro

About

- Vasilij Schneidermann, 25
- Software developer at bevuta IT, Cologne
- mail@vasilij.de
- `https://github.com/wasamasa`
- `http://emacshorrors.com/`
- `http://emacsninja.com/`

Motivation

- The current state of crypto is worrisome
- More attacks found than ever
- Rise in papers on side-channel attacks
- Yet: Most people ignore crypto or focus on a specific application (like, crypto currencies)
- How does one learn it?
- How hard can it be?

Context

- Looking for programming challenges, most were boring
- Cryptopals challenges:
  - Well designed, incremental
  - Cover several fields (symmetric/asymmetric crypto, signing, PRNG, hashing, zero-knowledge proofs, protocols/handshakes)
  - Programming language doesn't matter
  - Can be completed offline
  - You measure your own progress

Basics

- Confidentiality, Integrity, Authenticity
- Symmetric and asymmetric cryptography
- Plaintext, ciphertext
- Key, IV, nonce
- Block and stream cipher modes

Section 2

Selected attacks

Candidates

- Crack an MT19937 seed
- Single-byte XOR cipher
- CBC bitflipping attacks
- Break “random access read/write” AES CTR
- Compression Ratio Side-Channel Attacks

Crack an MT19937 seed

- This one doesn't even involve crypto
- MT19937 is a very popular PRNG
- Some people use it for crypto...
- Some people seed it from the current time...
- Given a MT19937 output seeded with a UNIX timestamp from a few minutes ago, how do you figure out the seed?

Crack an MT19937 seed

```
(use extras posix (prefix random-mtzig mt19937:))

(define (random-number seed)
  (let ((rng (mt19937:init seed)))
    (mt19937:random! rng)))

(define now (inexact->exact (current-seconds)))
(define then (- now 123))
(define rng-output (random-number then))
```

Crack an MT19937 seed

- PRNG generates a specific sequence of numbers for a given seed
- If you use the same seed as for a previous run, you get the same numbers
- Idea: Try possible timestamps as seed values, check whether generated numbers match up

Crack an MT19937 seed

```
(define (crack-it starting-time rng-output)
  (let loop ((seed starting-time))
    (if (= (random-number seed) rng-output)
        seed
        (loop (sub1 seed)))))

(printf "Predictable seed: ~a, output: ~a\n" then rng-output)
(printf "Cracked seed: ~a\n" (crack-it now rng-output))
```

Crack an MT19937 seed

- Complexity: Negligible
- Workaround: Never seed with predictable data, use the CSPRNG your OS provides for seeding (good libraries will do that for you)
- Combining many different entropy sources (PID, number of cores, etc.) is a popular alternative, but not much better: `https://blog.cr.yp.to/20140205-entropy.html`

Single-byte XOR cipher

- Equivalent of the caesar cipher, but with XOR instead of rotation
- XOR is reversible, $x \oplus y = z, z \oplus y = x, z \oplus x = y$
- Given a message in English with every byte XOR'd against a secret byte, figure out the message

Single-byte XOR cipher

- We can do this by introducing a scoring function for a piece of text
- The more it looks like English, the higher the score
- Non-ASCII gives a failing score
- Use Chi-Squared test for comparing given to ideal distribution
- The decryption with the best score is the right one

Single-byte XOR cipher

```
(define (hexdecode string)
  (map (cut string->number <> 16)
       (string-chop string 2)))

(define ciphertext
  (hexdecode (string-append "48434248404e452b5868636e666e2b"
                            "796e626c65782b787e7b796e666e")))

(define (str bytes)
  (list->string (map integer->char bytes)))

(define (ascii? string)
  (every (lambda (char) (<= 0 (char->integer char) 127))
         (string->list string)))
```

Single-byte XOR cipher

```
(define (xor-bytes-with-byte bytes byte)
  (map (lambda (b) (bitwise-xor b byte)) bytes))

(define english-histogram
  (alist->hash-table
   '((#\space . 0.14) (#\. . 0.09)
     (#\e . 0.12) (#\t . 0.09) (#\a . 0.08)
     (#\o . 0.07) (#\i . 0.06) (#\n . 0.06)
     (#\s . 0.06) (#\h . 0.06) (#\r . 0.05)
     (#\d . 0.04) (#\l . 0.04) (#\u . 0.02)
     ;; ...
     )))
```

Single-byte XOR cipher

```scheme
(define (frequencies string)
  (let ((ht (make-hash-table))
        (total (string-length string)))
    (for-each (lambda (char)
                (hash-table-update!/default ht char add1 0))
              (string->list string))
    (hash-table-walk ht (lambda (k v)
                          (hash-table-set! ht k (/ v total))))
    ht))
```

Single-byte XOR cipher

```
(define (chi-squared hist1 hist2)
  (hash-table-fold
   hist1
   (lambda (k v1 score)
     (let ((v2 (hash-table-ref/default hist2 k 0)))
       (if (zero? v1)
           score
           (+ score (/ (expt (- v1 v2) 2) v1)))))
   0))
```

Single-byte XOR cipher

```scheme
(define (english-score string)
  (if (ascii? string)
      (let* ((input (string-downcase string))
             (input (irregex-replace/all "[^ a-z]" input "."))
             (hist (frequencies input))
             (score (/ 1 (chi-squared english-histogram hist))))
        (if (< (hash-table-ref/default hist #\. 0) 0.05)
            (* score 2)
            score))
      0))
```

Single-byte XOR cipher

```
(let loop ((byte 0)
           (best-score 0)
           (best-solution ""))
  (if (< byte 256)
      (let* ((solution (str (xor-bytes-with-byte ciphertext byte)))
              (score (english-score solution)))
        (if (> score best-score)
            (loop (add1 byte) score solution)
            (loop (add1 byte) best-score best-solution)))
      (begin
        (printf "Score: ~a\n" best-score)
        (print best-solution))))
```

Single-byte XOR cipher

- Hardest part: Coming up with a usable scoring function
- Keys longer than a single byte can still be cracked with a similar approach
- Some broken cryptosystems revert to this difficulty level. . .

CBC bitflipping attacks

- Let's move on to actual crypto with AES
- ECB is broken, so this one uses CBC mode
- Suppose an attacker retrieved a cookie encrypted with AES-CBC, resembling `comment=1234567890&uid=3`
- The attacker likes to modify the cookie to end in `uid=0` to become admin, however they can't just decrypt, modify and re-encrypt
- Watch what happens if they just modify the ciphertext and what the resulting plaintext is. . .

CBC bitflipping attacks

Modification: XOR the first byte with a random byte

```
regular:   636f6d6d656e743d3132333435363738393026756964 3d33
tampered:  81436eafdd906ac37874635465fa81fb3a30267569643d33
```

Result: First block is completely different, first byte of second block has been XOR'd with that random byte

CBC bitflipping attacks

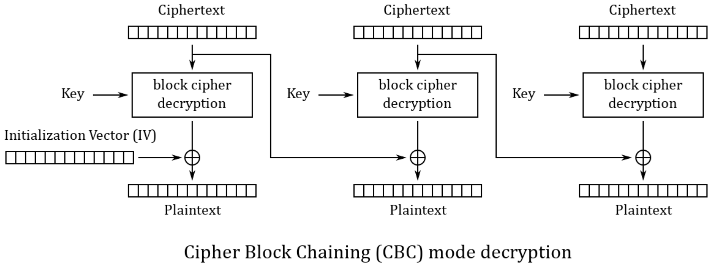

Figure: Source: Wikipedia

CBC bitflipping attacks

```
(define key (random-bytes 16))
(define iv (random-bytes 16))
(define plaintext "comment=1234567890&uid=3")
(define ciphertext
  (aes-cbc-encrypt (pkcs7pad (bytes plaintext) 16) key iv))

(define (check ciphertext)
  (let* ((plaintext (str (pkcs7unpad (aes-cbc-decrypt ciphertext
                     key iv))))
         (params (form-urldecode plaintext))
         (uid (alist-ref 'uid params)))
    (printf "checking ~s...\n" plaintext)
    (when (not uid)
      (error "invalid string"))
    (string->number uid)))
```

CBC bitflipping attacks

```
;; existing byte is '3' and should become '0'
(define tampered-byte (bitwise-xor (char->integer #\3)
                                   (char->integer #\0)))
(define tampered
  ;; the uid is byte #8 of block #2, so manipulate it in block #1
  (update-at (cut bitwise-xor <> tampered-byte) 7 ciphertext))

(printf "regular UID: ~a\n" (check ciphertext))
(printf "tampered UID: ~a\n" (check tampered))
```

CBC bitflipping attacks

- Other cipher modes have similar behavior (with CTR the same block is affected, no corruption of other blocks)
- Solution: Sign your cookies, verify the signature to ensure it hasn't been tampered with
- Weaker solution: Introduce a checksum to validate the integrity
- Alternative: Use cipher mode with integrated authentication (like AES-GCM)

Break “random access read/write” AES CTR

- AES again, but this time with a stream cipher
- Suppose an attacker retrieves a message encrypted with AES-CTR
- The message originates from a web application that allows editing them and re-encrypts the result
- This re-encryption can be done efficiently thanks to CTR allowing you to “seek” into the keystream and allows you to patch in the changed portion of the text
- Luckily the attacker has access to `(edit ciphertext offset  newtext)` which returns the new ciphertext after editing

Break "random access read/write" AES CTR

```
(define key (random-bytes 16))
(define nonce (random (expt 2 32)))
(define ciphertext (aes-ctr-encrypt plaintext key nonce))

(define (edit* ciphertext key nonce offset newtext)
  (let* ((decrypted (aes-ctr-decrypt ciphertext key nonce))
         (before (take decrypted offset))
         (after (drop decrypted (+ offset (length newtext))))
         (patched (append before newtext after)))
    (aes-ctr-encrypt patched key nonce)))

(define (edit ciphertext offset newtext)
  (edit* ciphertext key nonce offset newtext))
```

Break "random access read/write" AES CTR

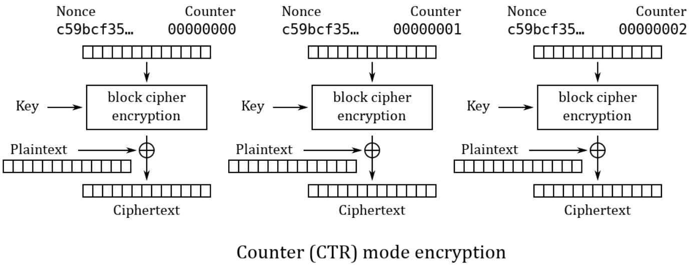

Figure: Source: Wikipedia

Break “random access read/write” AES CTR

- The transformation is far simpler than CBC
- Unknown plaintext is XORed with an encrypted key stream depending on a nonce
- $P_u \oplus E(k, K, N)$
- If the attacker XORs a known ciphertext with the existing one, something interesting happens:
- $P_u \oplus E(k, K, N) \oplus P_k \oplus E(k, K, N) = P_u \oplus P_k$
- The attacker knows his own plaintext, but not the other one
- $P_u \oplus P_k \oplus P_k = P_u$

Break "random access read/write" AES CTR

```
(define (decrypt ciphertext)
  (let* ((our-plaintext (random-bytes (length ciphertext)))
         (our-ciphertext (edit ciphertext 0 our-plaintext)))
    (xor-bytes
     (xor-bytes ciphertext our-ciphertext)
     our-plaintext)))

(print (str (decrypt ciphertext)))
```

Break “random access read/write” AES CTR

- Bonus: The `edit` procedure allows a crypto-agnostic (slow) way to decrypt the message one byte at a time
- Suppose the attacker compares an edited ciphertext with the original, it will always be different
- However if the edit didn’t change the content, both ciphertexts will be the same
- This can be used to guess part of the plaintext
- For a byte at a given offset, guess all possible values, one of them will reveal the plaintext byte
- Repeat for all possible offsets and join all found plaintext bytes

Break “random access read/write” AES CTR

- Ultimately, this attack is enabled by nonce reuse, randomize the nonce and the keystreams no longer match up
- For the bonus one, it should be impossible to tell if a guess was successful or better, the resulting encryption result shouldn’t be leaked
- Imagine if someone used this CTR property for something like FDE. . .

Compression Ratio Side-Channel Attacks

- This one is a side-channel attack and circumvents crypto
- Suppose the attacker is MITM and intercepts encrypted traffic resembling HTTP
- Additionally to that they can inject their own content (like, by changing the query to contain a search term)
- They know there's a cookie inside the header and want to guess it
- If the response is compressed before encryption, this can be done by checking the compressed size

Compression Ratio Side-Channel Attacks

- Compression generally works by finding repeating subsequences and replacing these with something shorter
- Suppose we compress a string containing `sessionid=abcdef`, a subsequent `sessionid=a` will result in better compression than a subsequent `sessionid=b`
- Generally, the difference in reduction is measured in bits, but will often be enough to differ by a byte

Compression Ratio Side-Channel Attacks

```
(define (format-request input)
  (format "POST / HTTP/1.1
Host: example.com
Cookie: sessionid=~a
Content-Length: ~a
~a
" session-id (string-length input) input))

(define (oracle input)
  (let ((key (random-bytes 16))
        (nonce (random (expt 2 32))))
    (length (aes-ctr-encrypt (bytes (compress (format-request input)))
                             key nonce))))
```

Compression Ratio Side-Channel Attacks

```
POST / HTTP/1.1
Host: example.com
Cookie: sessionid=Q0hJQ0tFTiBTY2hlbWUgcmVpZ25zIHN1cHJlbWU
Content-Length: 21
sessionid=Pdu0Jaesh9n
```

```
(oracle "sessionid=Pdu0Jaesh9n") ;=> 121
```

Compression Ratio Side-Channel Attacks

```
POST / HTTP/1.1
Host: example.com
Cookie: sessionid=Q0hJQ0tFTiBTY2hlbWUgcmVpZ25zIHN1cHJlbWU
Content-Length: 21
sessionid=Qdu0Jaesh9n
```

```
(oracle "sessionid=Qdu0Jaesh9n") ;=> 120
```

Compression Ratio Side-Channel Attacks

- Try each byte and record the guesses
- A guess with a shorter compression size is likely to be correct
- Add the guessed byte to the list of known bytes
- If there's no good guess, either we've failed early or there's no more bytes to guess and we're done
- To avoid false positives, add uncompressable (random) junk

Compression Ratio Side-Channel Attacks

```
(define (guess-byte known)
  (let ((guesses (make-hash-table))
        ;; this improves our chances considerably
        (suffix (random-bytes 10 from: 128 to: 256)))
    (for-each (lambda (byte)
                (let* ((guess (append known (list byte) suffix))
                       (input (format "sessionid=~a" (str guess))))
                  (hash-table-set! guesses byte (oracle input))))
              charset)
    (min-max-by cdr (hash-table->alist guesses))))

(define (guess-bytes)
  (let loop ((known '()))
    (receive (min max) (guess-byte known)
      (if (< (cdr min) (cdr max))
          (let ((known (append known (list (car min)))))
            (report-progress (str known) "guessed: ")
            (loop known))
          known))))
```

Compression Ratio Side-Channel Attacks

- This is a simplified version of actual attacks, like CRIME, BREACH, HEIST
- No real fix for this one (other than disabling compression)
- Other workarounds:
    - Use crypto that pads to block sizes (like AES-CBC, easy to work around)
    - Have the web server add random junk to the end (can be probably worked around with repeated guessing)
    - Add padding that makes the length uniform (as suggested by an expired TLS RFC draft)
    - Use XSRF tokens to mitigate the results of cookie stealing (good luck applying that to every web application...)

Section 3

Outro

Summary

- There's lots of crypto out there not involving hard math
- Good amount of well-understood attacks
- Side-channel attacks are scary and circumvent crypto
- Crypto systems aren't necessarily as safe as the primitives they consist of
- "Don't roll your own crypto" applies to primitives and cryptosystems
- You should totally do the cryptopals challenges

Questions?