Knowing just enough crypto to be dangerous

Vasilij Schneidermann

June 2019

Outline

Section 1

Intro

About

- Vasilij Schneidermann, 26
- 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

Picking a suitable programming language

- I went for Ruby
  - Expressive
  - Extensive standard library
  - Covers all prerequisites (OpenSSL, bignums)
  - Chance to improve my Ruby skills

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

```
def random_number(seed)
  Random.new(seed).rand(2**32)
end

now = Time.now.to_i
seed = now - 123
rng_output = random_number(seed)
```

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 first generated number matches up

Crack an MT19937 seed

```ruby
def crack_it(start_time, rng_output)
  seed = start_time
  loop do
    return seed if random_number(seed) == rng_output
    seed -= 1
  end
end

puts "Predictable seed: #{seed}, output: #{rng_output}"
puts "Cracked seed: #{crack_it(now, rng_output)}"
```

Crack an MT19937 seed

- Complexity: Negligible
- Happens more often than you'd think: https://arxiv.org/abs/1802.03367
- 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

- Modern equivalent of the caesar cipher, but with XOR instead of rotation
- Each byte of the plain text is combined with a secret byte using the XOR operator
- 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, how would you 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

```ruby
ENGLISH_HISTOGRAM = {
  ' ' => 0.14,
  :other => 0.09,
  'e' => 0.12,
  't' => 0.09,
  'a' => 0.08,
  'o' => 0.07,
  'i' => 0.06,
  'n' => 0.06,
  # ...
}

def frequencies(string)
  result = Hash.new { |h, k| h[k] = 0 }
  total = string.length
  string.each_char { |char| result[char] += 1 }
  result.each { |k, v| result[k] = v.to_f / total }
  result
end
```

Single-byte XOR cipher

```
def chi_squared(hist1, hist2)
  score = 0
  hist1.each do |k, v1|
    v2 = hist2[k] || 0
    next if v1.zero?
    score += (v1 - v2)**2 / v1
  end
  score
end

def english_score(string)
  return 0 unless string.ascii_only?
  input = string.downcase.tr('^ a-z', '.')
  histogram = frequencies(input)
  histogram[:other] = histogram['.'] || 0
  histogram.delete('.')
  score = 1 / chi_squared(ENGLISH_HISTOGRAM, histogram)
  score *= 2 if histogram[:other] < 0.05
  score
end
```

Single-byte XOR cipher

```
best_score = 0
best_solution = ''

(0..255).each do |key|
  solution = str(xor_buffer_with_byte(CIPHERTEXT, key))
  score = english_score(solution)
  if score > best_score
    best_score = score
    best_solution = solution
  end
end

puts "score: #{best_score}"
puts 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 we'll use CBC mode instead
- 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 ciphertext: 24fe5dcfa80f182d3e1ee5f486723e9b33516b7a2846b1..
tampered ciphertext: 66fe5dcfa80f182d3e1ee5f486723e9b33516b7a2846b1..
  regular plaintext: 636f6d6d656e743d3132333435363738393026756964 3d33
 tampered plaintext: 06ef88d48792df331838931d121fca227b30267569643d33
```

```
0x24 ^ 0x66 == 0x42
0x39 ^ 0x7b == 0x42
```

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

CBC bitflipping attacks

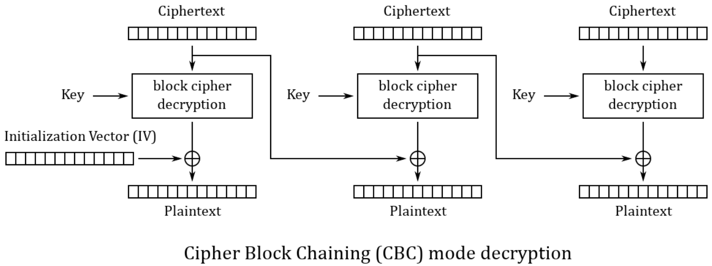

Figure: Source: Wikipedia

CBC bitflipping attacks

```
KEY = random_bytes(16)
IV = random_bytes(16)
PLAINTEXT = 'comment=1234567890&uid=3'
CIPHERTEXT = aes_cbc_encrypt(PLAINTEXT.bytes, KEY, IV)

def check(ciphertext)
  plaintext = str(aes_cbc_decrypt(ciphertext, KEY, IV))
  params = decode_query_string(plaintext)
  uid = params['uid']
  puts "checking #{plaintext.inspect}..."
  raise 'invalid string' unless uid
  uid.to_i
end
```

CBC bitflipping attacks

```
tampered_byte = '3'.ord ^ '0'.ord
tampered = CIPHERTEXT.clone
tampered[7] ^= tampered_byte

puts "regular UID: #{check(CIPHERTEXT)}"
puts "tampered UID: #{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
- Weak 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 the following API call which returns the new ciphertext after editing: `/edit?ciphertext=...&offset=...&newtext=...`

Break “random access read/write” AES CTR

```
KEY = random_bytes(16)
NONCE = random_bytes(16)
CIPHERTEXT = aes_ctr_encrypt(PLAINTEXT, KEY, NONCE)

def edit_internal(ciphertext, key, nonce, offset, newtext)
  decrypted = aes_ctr_decrypt(ciphertext, key, nonce)
  newtext.each_with_index { |byte, i| decrypted[offset + i] = byte }
  aes_ctr_encrypt(decrypted, key, nonce)
end

def edit(ciphertext, offset, newtext)
  edit_internal(ciphertext, KEY, NONCE, offset, newtext)
end
```

Break "random access read/write" AES CTR

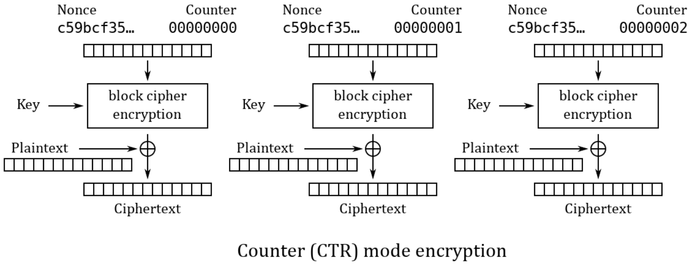

Figure: Source: Wikipedia

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

```
random_message = random_bytes(ciphertext.length)
edited_message = edit(ciphertext, 0, random_message)
puts str(xor_buffers(xor_buffers(ciphertext, edited_message),
                     random_message))
```

Break “random access read/write” AES CTR

- Bonus: `/edit` 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

```
def guess_byte(ciphertext, offset)
  (0..127).each do |byte|
    return byte if ciphertext == edit(ciphertext, offset, [byte])
  end
  raise "couldn't guess byte"
end

ciphertext.size.times { |i| print guess_byte(ciphertext, i).chr }
```

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
- Oracle: Mechanism revealing a piece of information to the attacker

Compression Ratio Side-Channel Attacks

```
def format_request(input)
  "POST / HTTP/1.1
Host: example.com
Cookie: sessionid=#{SESSIONID}
Content-Length: #{input.length}
#{input}
"
end

def oracle(input)
  key = random_bytes(16)
  nonce = random_bytes(16)
  payload = compress(format_request(input))
  aes_ctr_encrypt(payload.bytes, key, nonce).size
end
```

Compression Ratio Side-Channel Attacks

```
POST / HTTP/1.1
Host: example.com
Cookie: sessionid=447520626973742042756464686973742e
Content-Length: 21
sessionid=31415926
```

```
oracle('sessionid=31415926') #=> 117
```

Compression Ratio Side-Channel Attacks

```
POST / HTTP/1.1
Host: example.com
Cookie: sessionid=44752062697374204275646468697374 2e
Content-Length: 21
sessionid=41415926
```

```
oracle('sessionid=41415926') #=> 116
```

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

```
CHARSET = '0123456789abcdef'

def ctr_guess_byte(known)
  guesses = {}
  suffix = random_bytes(10, (128..255))
  CHARSET.each_byte do |byte|
    input = "sessionid=#{str(known + [byte] + suffix)}"
    guesses[byte] = oracle(input)
  end
  guesses.minmax_by { |_, v| v }
end
```

Compression Ratio Side-Channel Attacks

```
known = []
loop do
  min, max = ctr_guess_byte(known)
  if min[1] == max[1]
    if known.length >= 32
      return known
    else
      known = []
      redo
    end
  end
  known << min[0]
  report_progress(str(known))
end
```

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)
- Note: Compressing after encryption doesn't make much sense
- 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, especially if you're a web developer
- Crypto can be fun!

Questions?