Krypto knacken für Anfänger

Vasilij Schneidermann

Oktober 2018

Outline

Abschnitt 1

Intro

Sprecher

- Vasilij Schneidermann, 26
- Software-Entwickler bei bevuta IT GmbH
- mail@vasilij.de
- https://github.com/wasamasa
- http://emacshorrors.com/
- http://emacsninja.com/

Motivation

- Kryptographie ist nicht mehr wegzudenken
- Viele Exploits mit einfallsreichen Namen (ROBOT, KRACK, EFAIL)
- Viele Papers über Side-Channel-Angriffe (Meltdown, Spectre, TLBleed, Foreshadow)
- Dennoch: Viele Menschen ignorieren Krypto oder beschäftigen sich lieber mit Kryptowährungen
- Wie steigt man in das Thema ein?
- Wie schwierig ist es wirklich?

Kontext

- Suche nach interessanten Programmierübungen
- https://cryptopals.com/
  - Interessante Claims (Minimum an Mathematik, realistische Angriffe)
  - Gutes Design, aufeinander aufbauend
  - Viele Felder abgedeckt (symmetrische/asymmetrische Krypto, Signaturen, PRNG, Hashes, Zero-Knowledge Proofs, Protokolle/Handshakes)
  - Programmiersprache ist irrelevant
  - Offline lösbar (ursprünglich via E-Mail)
  - Maximale Freiheit gegeben

Grundlagen

- Confidentiality, Integrity, Authenticity
- Symmetrische, asymmetrische Verschlüsselung
- Plaintext, Ciphertext
- Key, IV, Nonce
- Block Cipher, Stream Cipher

Abschnitt 2

Ausgewählte Angriffe

Kandidaten

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

Verwendete Programmiersprache

- Ich habe mich für Ruby entschieden
  - Ausdrucksstark
  - Umfangreiche Standardbibliothek
  - Erfüllt alle Vorbedingungen (OpenSSL, Bignums)
  - Gelegenheit meine Ruby-Skills aufzubessern

Crack an MT19937 seed

- Involviert (noch) keine Krypto
- MT19937 ist ein populärer PRNG
- Manche Leute nutzen es für Krypto...
- Manche Leute seeden es mit der aktuellen Zeit...
- Gegeben sei der erste MT19937-Output mit der aktuellen Zeit von vor wenigen Minuten als Seed
- Wie knackt man den Seed?

Crack an MT19937 seed

```ruby
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

- Ein PRNG generiert eine eindeutige Zahlenfolge für einen Seed
- Wenn man den gleichen Seed verwendet, erhält man die gleiche Zahlenfolge
- Idee: Alle möglichen Zeitstempel als Seed probieren, testen ob die erste generierte Zahl übereinstimmt

Crack an MT19937 seed

```
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

- Aufwand: Vernachlässigbar
- Passiert häufiger als man denkt: https://arxiv.org/abs/1802.03367
- Workaround: Niemals mit erratbaren Daten seeden, CSPRNG vom Betriebssystem verwenden (gute Bibliotheken tun das von Haus aus)
- Kombination vieler Entropiequellen ist beliebt (PID, systemspezifische Daten, etc.), aber nicht viel besser: https://blog.cr.yp.to/20140205-entropy.html

Single-byte XOR cipher

- Moderne Variante der Caesar-Verschlüsselung
- Jedes Byte vom Plaintext wird mit einem geheimen Byte durch den XOR-Operator kombiniert
- XOR ist umkehrbar: $x \oplus y = z, z \oplus y = x, z \oplus x = y$
- Gegeben sei ein auf diese Weise verschlüsselter Text auf Englisch
- Wie knackt man diesen Text?

Single-byte XOR cipher

```
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

```ruby
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

- Größte Herausforderung: Brauchbare Scoring-Funktion schreiben
- Verschlüsselung mit längeren Schlüsseln lassen sich ähnlich knacken
- Es gibt kaputte Krypto-Systeme die auf diese Schwierigkeitsstufe zurückfallen

CBC bitflipping attacks

- Diese Übung nutzt nicht trivial knackbare Krypto (AES)
- ECB wird nicht mehr empfohlen, deswegen wird hier CBC genutzt
- Angenommen ein Angreifer findet einen mit AES-CBC verschlüsselten Cookie der nach `comment=1234567890&uid=3` aussieht
- Der Angreifer möchte diesen Cookie so bearbeiten, dass `uid=0` drin steht um Admin zu werden
- Es ist nicht möglich für den Angreifer den Cookie zu entschlüsseln, bearbeiten und erneut verschlüsseln (da Schlüssel und IV unbekannt)
- Was passiert wenn man den Ciphertext direkt bearbeitet?

CBC bitflipping attacks

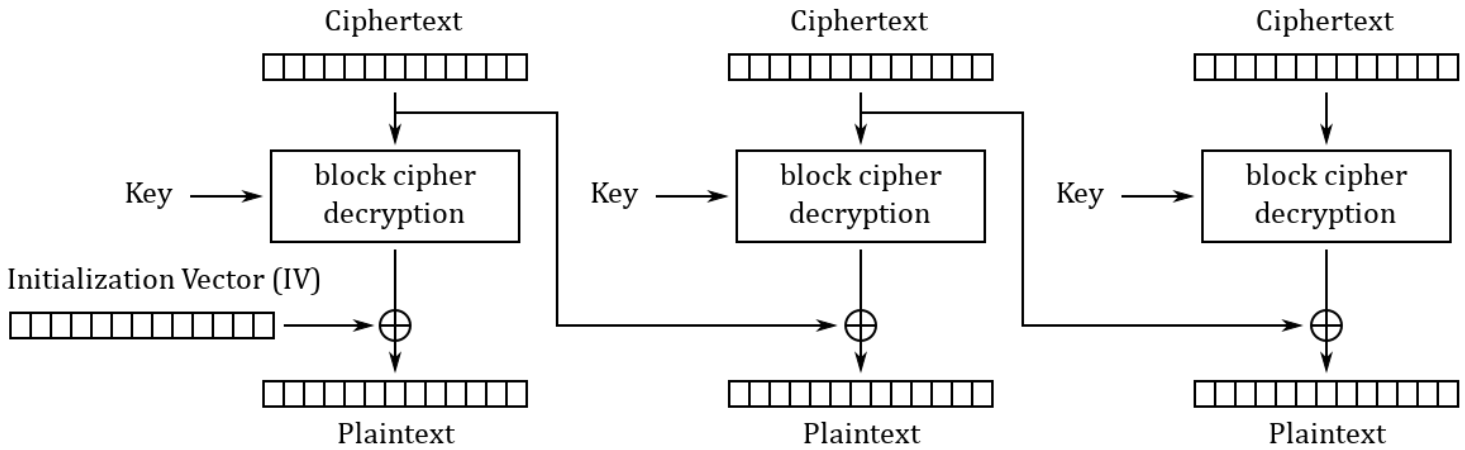
Modifikation: Erstes Byte wurde mit zufälligem Byte mit XOR kombiniert

```
 regular ciphertext: 24fe5dcfa80f182d3e1ee5f486723e9b33516b7a2846b1..
tampered ciphertext: 66fe5dcfa80f182d3e1ee5f486723e9b33516b7a2846b1..
  regular plaintext: 636f6d6d656e743d313233343536373839302675696 43d33
 tampered plaintext: 06ef88d48792df331838931d121fca227b302675696 43d33
```

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

Resultat: Erster Block ist komplett anders, erstes Byte vom zweiten Block wurde auf die gleiche Weise manipuliert

CBC bitflipping attacks



Cipher Block Chaining (CBC) mode decryption

Abbildung: Quelle: Wikipedia

CBC bitflipping attacks

```ruby
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

- Nicht nur CBC ist von diesem Verhalten betroffen (bei CTR wird der gleiche Block manipuliert)
- Lösung: Cookies signieren und Signatur verifizieren, bei ungültiger Signatur den Cookie nicht nutzen
- Schlechte Lösung: Prüfsumme hinzufügen und überprüfen
- Alternative: Verschlüsselung mit integrierter Authentication nutzen (z.B. AES-GCM)

Break “random access read/write” AES CTR

- Erneut AES, diesmal mit Stream-Cipher
- Angenommen ein Angreifer fängt eine mit AES-CTR verschlüsselte Nachricht ab
- Diese Nachricht stammt von einer Web-Anwendung die es Usern ermöglicht verschlüsselte Nachrichten zu editieren
- Es wird eine Eigenschaft von CTR für effizientes Editieren ausgenutzt (damit nur der nötige Bruchteil der Nachricht ersetzt werden muss)
- Der Angreifer hat Zugriff auf einen interessanten API-Call welcher einen neuen Ciphertext zurückgibt: `/edit?ciphertext=...&offset=...&newtext=...`

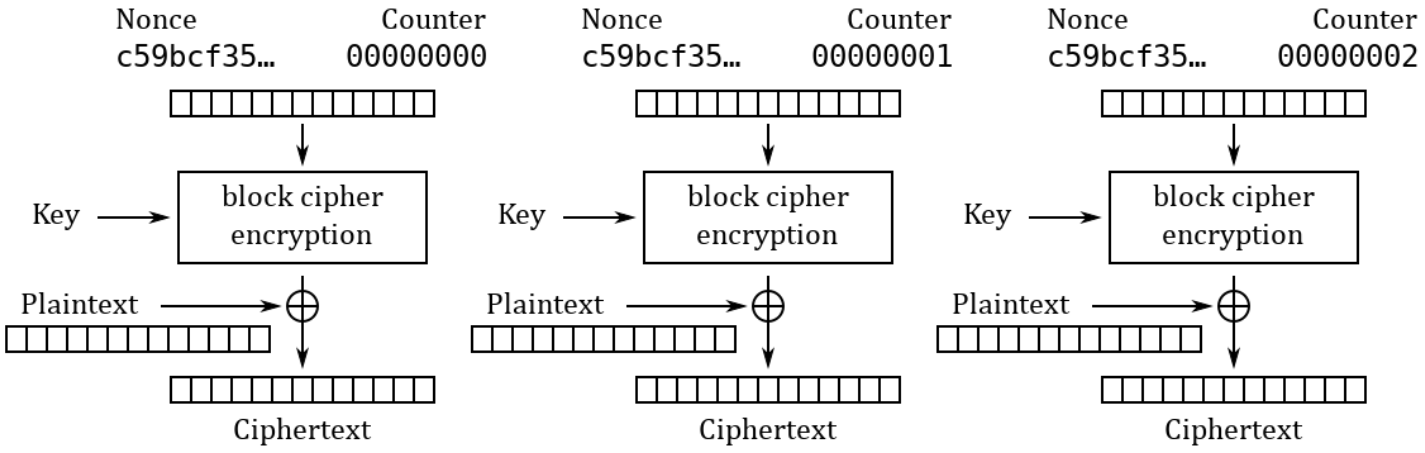
Break "random access read/write" AES CTR

```
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



Counter (CTR) mode encryption

Abbildung: Quelle: Wikipedia

Break “random access read/write” AES CTR

- Deutlich einfachere Transformation als bei CBC
- Plaintext wird mit XOR mit verschlüsseltem Keystream kombiniert
- Der Keystream ist mit einem Nonce parametrisiert
- $P_u \oplus E(k, K, N)$
- Wenn der Angreifer einen ihm bekannten Ciphertext mit dem existierenden kombiniert, passiert folgendes:
- $P_u \oplus E(k, K, N) \oplus P_k \oplus E(k, K, N) = P_u \oplus P_k$
- Der Angreifer kennt seinen eigenen Plaintext, aber nicht den der zu entschlüsselnden Nachricht
- $P_u \oplus P_k \oplus P_k = P_u$

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` erlaubt den Text Byte für Byte zu erraten
- Angenommen der Angreifer vergleicht den editierten Ciphertext mit dem ursprünglichen
- Sind die Inhalte verschieden, werden die Ciphertexts auch verschieden sein
- Sind die Inhalte gleich, werden die Ciphertexts auch gleich sein
- Wenn der Angreifer durch Zufall einen Edit gefunden bei dem der gleiche Ciphertext zurückgegeben wird, hat er einen Teil des Plaintexts erraten
- Kleinstmöglicher Edit: 1 Byte
- 1 Byte kann mit maximal 256 Edits erraten werden
- Offset inkrementieren und das nächste Byte erraten

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

- Ursache: Nonce wird wiederverwendet, wenn man das vermeidet ist der Keystream ein anderer und der Angriff ist nicht mehr erfolgreich
- Bonus: Möglichst wenig Informationen leaken, exzessive Zugriffe loggen und blockieren
- Gedankenexperiment: Was wäre wenn jemand diese Eigenschaft von CTR auf FDE anwendet?

Compression Ratio Side-Channel Attacks

- Side-Channel-Angriff, umgeht Krypto vollständig
- Angenommen der Angreifer ist MITM, fängt HTTP-Nachrichten ab und kann zu diesen neuen Text hinzufügen bevor sie durchgereicht werden
- Der Angreifer möchte den Cookie im HTTP-Header erraten
- Das ist möglich wenn die Nachricht vor der Verschlüsselung komprimiert wird und man auf die Größe der Nachricht prüft

Compression Ratio Side-Channel Attacks

- Bei Kompression werden wiederkehrende Zeichenketten gefunden und mit einer kürzeren ersetzt
- Angenommen wir komprimieren einen Text der `sessionid=abcdef` beinhaltet, dann würde dieser Text etwas besser komprimiert wenn danach ein `sessionid=a` folgt
- Würde stattdessen `sessionid=b` folgen, würde der Text etwas schlechter komprimiert
- Die Unterschiede werden üblicherweise in Bits gemessen, oft beträgt der Unterschied aber mehr als ein Byte
- Oracle: Ein Mechanismus der dem Angreifer ein Stück Information verrät

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=4475206269737420427564646869737 42e
Content-Length: 21
sessionid=41415926
```

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

Compression Ratio Side-Channel Attacks

- Liste erzeugen von ausprobierten Bytes und Längen
- Wenn eine Länge kürzer als alle anderen ist, wurde das Byte korrekt erraten
- Erratenes Byte wird zu der Liste bekannter Bytes hinzugefügt und das nächste Byte erraten
- Wenn es keinen Treffer gibt wurde entweder der gesamte Cookie erraten oder es gab einen Fehler und man fängt erneut an
- False Positives lassen sich durch das Hinzufügen von unkomprimierbaren Daten reduzieren

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

- Diese Übung ist eine vereinfachte Version von Angriffen wie CRIME, BREACH und HEIST
- Es gibt keinen Fix (außer Kompression zu deaktivieren)
- Workarounds:
  - Block-Cipher nutzen (verlangsamt den Angriff lediglich)
  - Web-Server zufällig langen Text an die Response anhängen lassen (Mitigation von nginx)
  - Länge jeder Response gleich machen (es gibt einen abgelaufenen TLS RFC Draft dafür)
  - XSRF-Tokens nutzen um geklaute Cookies nutzlos zu machen (viel Glück das zu jeder Webanwendung hinzuzufügen. . .)

Abschnitt 3

Outro

Zusammenfassung

- Es gibt viel Krypto die keine fortgeschrittene Mathematik erfordert und viele Angriffe auf diese
- Side-Channel-Angriffe sind erstaunlich effektiv (da sie Krypto umgehen)
- "Don't roll your own crypto" bezieht sich auf Krypto-Primitive und -Systeme
- Die Challenges lohnen sich, insbesondere für Web-Entwickler
- Es macht Spaß Krypto auf diese Art und Weise anzugehen

Fragen?