A Perfect CRIME? TIME Will Tell

Tal Be’ery, Imperva

OWASP
The Open Web Application Security Project
• Web Security Research Team Leader at Imperva
• Holds MSc & BSc degree in CS/EE from TAU
• 10+ years of experience in IS domain
• Facebook “white hat”
• Speaker at RSA, BlackHat, AusCERT
• Columnist for securityweek.com
• Introduction
  - Compression Primer
  - CRIME attack revisited
• Expanding CRIME
  - Increasing the attack surface with HTTP responses
• TIME attack
  - Exploiting timing side channel
• Conclusions & mitigations
Introduction
Compression on the Web
Pre-CRIME

- Based on the GZIP algorithm
- Common compression
  - HTTP Response Body
- Uncommon compressions
  - HTTP Request body
  - Header compression
  - SSL/TLS Compression
    - Servers: Open SSL, others
    - Clients: Chrome
  - SPDY
    - Servers: Apache MOD_SPDY, others
    - Clients: All but IE
Two step compression process
- LZ77 to compress reoccurring strings
- Huffman code to compress frequent symbols

Good compression rate with low overhead
- Memory
- CPU
- Compression dictionaries
Compression – LZ Algorithms

- Lempel Ziv, late 70s

- Compress repeating strings
  - Lossless
  - Asymptotically optimal
  - No overhead (No extra dictionary)
001:001 In the beginning God created the heaven and the earth.

001:002 And the earth was without form, and void; and darkness was upon the face of the deep. And the Spirit of God moved upon the face of the waters.

- 001:001 In the beginning God created heaven and earth. And the earth was without form, and void; and darkness was upon the face of the deep. And the Spirit of God moved upon the face of the waters.
Huffman Code

- David Huffman - 1952
- Assign shorter codes (in bits) for frequent symbols (letters)

<table>
<thead>
<tr>
<th>X</th>
<th>p(X)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0.171</td>
</tr>
<tr>
<td>b</td>
<td>0.031</td>
</tr>
<tr>
<td>c</td>
<td>0.057</td>
</tr>
<tr>
<td>d</td>
<td>0.092</td>
</tr>
<tr>
<td>e</td>
<td>0.274</td>
</tr>
<tr>
<td>f</td>
<td>0.052</td>
</tr>
<tr>
<td>g</td>
<td>0.042</td>
</tr>
<tr>
<td>h</td>
<td>0.130</td>
</tr>
<tr>
<td>i</td>
<td>0.149</td>
</tr>
<tr>
<td>j</td>
<td>0.002</td>
</tr>
</tbody>
</table>

Build Tree

Assign Code

<table>
<thead>
<tr>
<th>X</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>00</td>
</tr>
<tr>
<td>b</td>
<td>11110</td>
</tr>
<tr>
<td>c</td>
<td>0110</td>
</tr>
<tr>
<td>d</td>
<td>1110</td>
</tr>
<tr>
<td>e</td>
<td>10</td>
</tr>
<tr>
<td>f</td>
<td>0111</td>
</tr>
<tr>
<td>g</td>
<td>11110</td>
</tr>
<tr>
<td>h</td>
<td>010</td>
</tr>
<tr>
<td>i</td>
<td>110</td>
</tr>
<tr>
<td>j</td>
<td>11111</td>
</tr>
</tbody>
</table>
Compression leaks data!

- Kelsey - 2002: “Compression and Information Leakage of Plaintext”
- Rizzo and Duong - 2012
- Compression Ratio Info-leak Made Easy (CRIME)
- Chosen Plaintext Attack
- Targets compression information
A chosen-plaintext attack (CPA) is an attack model for cryptanalysis which presumes that the attacker has the capability to choose arbitrary plaintexts to be encrypted and obtain the corresponding ciphertexts.
2. CRIME
JavaScript makes a request to the target server

3. Attacker can see encrypted packet lengths

1. Attacker makes a guess
CPA attacker algorithm:

- Guess(0) = a known prefix of the secret string
- Symbol = array of the secret alphabet (i.e {a,b,c..})
- Until the whole secret is recovered
- Guess(n) = Guess(n-1) + symbol(i)
- Payload = original payload + Guess(n)
- Measure length
  - For a correct guess String repeated gets compressed shorter length (encryption does not change size)
- If successful
  - n++, i=0 // proceed to guessing the next secret’s symbol
- Else
  - i++ // try another alphabet symbol
- Repeat
- Attacker is an eavesdropper – can see ciphered text
- Attacker creates HTTP request interactively (via script)
  - Full control (almost): URL
  - Can predict: Most headers
  - Does not control or see: cookies
  - Encrypted on wire
  - Not accessible from script
    - Same Origin Policy
    - “HTTP only”
Compression Leaks Data

**OWASP**
The Open Web Application Security Project

- **Attack model**
  - Use the URL attacker controls
  - Guess byte by byte

```
POST /sessionid=a HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:Cookie: sessionid=d8e8fca2dc0f896fd7cb4cb0031ba249

POST /sessionid=d HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:Cookie: sessionid=d8e8fca2dc0f896fd7cb4cb0031ba249
```
1. Attacker makes a guess

2. CRIME JavaScript makes a request to the target server

3. Attacker can see encrypted packet lengths
• Some issues with Huffman coding
  - Some chars representation < 1 byte
  - Good guess might get unnoticed

• Solution
  - Send some more requests with the same chars blend (same Huffman coding) in different order to (different LZ77 compression) to eliminate chars redundancy issues
CRIME Aftermath

• SPDY implementations cancel/modify header compression
• Chrome disabled SSL compression
Extending CRIME for HTTP responses
**CRIME attack “ingredients”**

<table>
<thead>
<tr>
<th>CRIME element</th>
<th>HTTP request</th>
<th>HTTP response</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encryption</td>
<td>SSL</td>
<td>SSL</td>
</tr>
<tr>
<td>Compression</td>
<td>GZIP</td>
<td>GZIP</td>
</tr>
<tr>
<td>Secret element location</td>
<td>Request header</td>
<td>Response body</td>
</tr>
<tr>
<td>Secret element</td>
<td>Cookie value</td>
<td>Application specific</td>
</tr>
<tr>
<td>Secret element prefix/suffix</td>
<td>Cookie name</td>
<td>Application specific</td>
</tr>
<tr>
<td>Chosen plain text location</td>
<td>URL</td>
<td>Application specific</td>
</tr>
</tbody>
</table>
• We need a secret with a known prefix/suffix

• Luckily, they are everywhere..
  - The applications’ secrets are in their content i.e. delivered by HTTP response body
  - Secrets are often structured with a fixed prefix or suffix
## CRIME attack “ingredients”

<table>
<thead>
<tr>
<th>CRIME element</th>
<th>HTTP request</th>
<th>HTTP response</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encryption</td>
<td>SSL</td>
<td>SSL</td>
</tr>
<tr>
<td>Compression</td>
<td>GZIP</td>
<td>GZIP</td>
</tr>
<tr>
<td>Secret element location</td>
<td>Request header</td>
<td>Response body</td>
</tr>
<tr>
<td>Secret element</td>
<td>Cookie value</td>
<td>Application specific</td>
</tr>
<tr>
<td>Secret element prefix/suffix</td>
<td>Cookie name</td>
<td>Application specific</td>
</tr>
<tr>
<td>Chosen plain text location</td>
<td>URL</td>
<td>Application specific</td>
</tr>
</tbody>
</table>
• Application specific - yet not infrequent
• Many applications embed user input (as expressed with HTTP parameters) within their response
• In fact, many times parameters will be embedded even if there are no parameters on the original request
<table>
<thead>
<tr>
<th>CRIME attack “ingredients”</th>
<th>HTTP Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRIME element</td>
<td>Secret element</td>
</tr>
<tr>
<td>Encryption</td>
<td>Secret element</td>
</tr>
<tr>
<td>Compression</td>
<td>Secret element</td>
</tr>
<tr>
<td>Secret element location</td>
<td>Chosen plain text</td>
</tr>
<tr>
<td>Secret element prefix/suffix</td>
<td>HTTP response</td>
</tr>
<tr>
<td>Secret element value</td>
<td>SSL</td>
</tr>
<tr>
<td>Secret element name</td>
<td>GZIP</td>
</tr>
<tr>
<td>Application specific</td>
<td>Response body</td>
</tr>
<tr>
<td>Application specific</td>
<td>Application specific</td>
</tr>
<tr>
<td>Application specific</td>
<td>Application specific</td>
</tr>
</tbody>
</table>
Google Scholar PoC

Add articles - guess@gmail.com

Find articles that you've written and add them to your profile. Later, you can edit or delete the articles in your profile or add more articles to your profile.

author: "guess@gmail.com"  Search article groups
Response vs. Request

Pros

- HTTP response body compression is a very common practice - cannot be easily turned off
- Attacking the secret data itself and not some intermediate (cookie)
Response vs. Request

Cons

• User input is encoded before embedding into the response to protect against injection attacks. Therefore the attack target is limited to mostly alphanumeric characters.

• Less sterile environment:
  - Response body might be altered due to other reasons
  - Input might get embedded more than once
The TIME attack
Motivation

- Crime attack model has some very limiting attack preconditions: Eavesdropping AND web page control
- Directing user traffic to a controlled site is a fairly easy task
- But eavesdropping to victim’s traffic with other site is a much harder requirement
- If only we could drop the eavesdropping requirement...
• Imperva – 3.2013
• Timing Info-leak Made Easy (TIME)
• Chosen Plaintext Attack
• Targets timing information leakage
• HTTP Payload size may carry sensitive information
  - Moreover, HTTP payload size differences detection is sufficient to extract the sensitive information
• Using timing measurements attacker can distinguish HTTP payload size differences
• These timing measurements can be done with javascript on attacker site
• Result - attackers can learn the user’s sensitive information using javascript from their site, with no eavesdropping!
- Attacker has the capability to choose arbitrary plaintexts and obtain timing observations on their traffic
- **Attacker no longer needs to be an eavesdropper!**
  - Expanding the attack scope

![Diagram showing the attack model](image-url)
Sensitive Info in HTTP Payload Size

- HTTP request
  - CRIME for request to extract cookie data
- HTTP response
  - Extended CRIME to extract response data
  - Access a behind authentication resource for user login status detection
  - Application specific: e.g. number of digits in bank account balance
- Moreover, HTTP payload size **differences** detection is sufficient to extract the sensitive information
Now that we know a lot about a visitor’s browser, we can mix and match several techniques — six, by my count — that http://maliciouswebsite/ can use to learn what other websites a visiting browser is logged in to — an online bank, social network, email provider, a local home router’s Web interface, and basically anything else.
Timing Reveals Payload Size

https://developers.google.com/speed/docs/best-practices/payload
Google web page speed tips for developers:

- “The amount of data sent in each server response can add significant latency to your application”
- “In addition to the network cost of the actual bytes transmitted, there is also a penalty incurred for crossing an IP packet boundary.”

https://developers.google.com/speed/docs/best-practices/payload
Timing Oracle

- Client send a window of packets
- Waits RTT for ACK
- RTT time is noticeable
- Attacker can easily distinguish
  - Size(request) <= window
  - Size(request) > window
- If payload length is exactly on data boundary, attacker can determine 1 byte differences

http://ulam2.cs.luc.edu/ebook/chap03.html
- Sent with Chrome
- Sends 2 packets and wait
- If you need to send 3 packets – pay extra

<table>
<thead>
<tr>
<th>No.</th>
<th>Time</th>
<th>Protocol</th>
<th>Length</th>
<th>Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>2284</td>
<td>0.000000000</td>
<td>TCP</td>
<td>66</td>
<td>27983 &gt; http [SYN] Seq=0 win=8192 Len=0 MSS=1460 W:</td>
</tr>
<tr>
<td>2298</td>
<td>0.177681000</td>
<td>TCP</td>
<td>66</td>
<td>http &gt; 27983 [SYN, ACK] Seq=0 Ack=1 win=14480 Len=0</td>
</tr>
<tr>
<td>2299</td>
<td>0.000920000</td>
<td>TCP</td>
<td>54</td>
<td>27983 &gt; http [ACK] Seq=1 Ack=1 win=65536 Len=0</td>
</tr>
<tr>
<td>2317</td>
<td>0.183176000</td>
<td>TCP</td>
<td>1514</td>
<td>[TCP segment of a reassembled PDU]</td>
</tr>
<tr>
<td>2318</td>
<td>0.183176000</td>
<td>TCP</td>
<td>1514</td>
<td>[TCP segment of a reassembled PDU]</td>
</tr>
<tr>
<td>2326</td>
<td>0.169969000</td>
<td>TCP</td>
<td>60</td>
<td>http &gt; 27983 [ACK] Seq=1 Ack=1461 Win=8960 Len=0</td>
</tr>
<tr>
<td>2327</td>
<td>0.000052000</td>
<td>HTTP</td>
<td>55</td>
<td>GET /?FTyNcuZg9XheUwA8l7mM9aUGk7WtuTdxsy8Na9i Matthews</td>
</tr>
<tr>
<td>2328</td>
<td>0.000039000</td>
<td>TCP</td>
<td>60</td>
<td>http &gt; 27983 [ACK] Seq=1 Ack=2921 Win=11776 Len=0</td>
</tr>
<tr>
<td>2332</td>
<td>0.167268000</td>
<td>TCP</td>
<td>60</td>
<td>http &gt; 27983 [ACK] Seq=1 Ack=2922 Win=11776 Len=0</td>
</tr>
<tr>
<td>2333</td>
<td>0.006509000</td>
<td>TCP</td>
<td>1502</td>
<td>[TCP segment of a reassembled PDU]</td>
</tr>
</tbody>
</table>
• This Apache server implements a window of 3 packets

• If it needs to send the fourth – pays extra RTT
Create HTTP request with XHR
  - XHR adheres to SOP
  - Allows GET requests to flow
  - If headers allow show response
  - If not, abort
  - We don’t care for the response
  - Timing leaks the request size

Use getTime() on XHR events
  - onreadystatechange

Noise elimination
  - Repeat the process (say 10 times) and obtain Minimal time
- HTML with Javascript, sending method is XHR
- PoC target edition.cnn.com
- Sends one byte diff requests alternately 10 times
- The longer request crosses the send window boundary
- The shorter is exactly within
- Measures requests time
- Outputs length and time
- Outputs the minimal timing values for both requests’ length
- Timing can be correctly captured
- Results are conclusive

<table>
<thead>
<tr>
<th>Name Path</th>
<th>Method</th>
<th>Status Text</th>
<th>Type</th>
<th>Initiator</th>
<th>Size</th>
<th>Time Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>edition.cnn.com</td>
<td>GET</td>
<td>(canceled)</td>
<td>Pending</td>
<td>XHR-timing-boundary.htm?</td>
<td>13B</td>
<td>723ms, 0.0 days</td>
</tr>
<tr>
<td>edition.cnn.com</td>
<td>GET</td>
<td>(canceled)</td>
<td>Pending</td>
<td>XHR-timing-boundary.htm?</td>
<td>13B</td>
<td>516ms, 0.0 days</td>
</tr>
<tr>
<td>edition.cnn.com</td>
<td>GET</td>
<td>(canceled)</td>
<td>Pending</td>
<td>XHR-timing-boundary.htm?</td>
<td>13B</td>
<td>740ms, 0.0 days</td>
</tr>
<tr>
<td>edition.cnn.com</td>
<td>GET</td>
<td>(canceled)</td>
<td>Pending</td>
<td>XHR-timing-boundary.htm?</td>
<td>13B</td>
<td>506ms, 0.0 days</td>
</tr>
<tr>
<td>edition.cnn.com</td>
<td>GET</td>
<td>(canceled)</td>
<td>Pending</td>
<td>XHR-timing-boundary.htm?</td>
<td>13B</td>
<td>491ms, 0.0 days</td>
</tr>
<tr>
<td>edition.cnn.com</td>
<td>GET</td>
<td>(canceled)</td>
<td>Pending</td>
<td>XHR-timing-boundary.htm?</td>
<td>13B</td>
<td>460ms, 0.0 days</td>
</tr>
</tbody>
</table>

Min first: 468
Min Second: 246
- Create HTTP request with iframe src
  - iframe adheres to SOP
  - Doesn’t allow parent to access the response content
  - Timing leaks the response size
- Use `getFrameTime()` on iframe events
  - `onLoad`
  - `Onreadystatechange` (IE)

- X-Frame-Options Header
• Create HTTP request with IMG src
  - Target resource is fetched even if not an image
  - not tamed by the X-Frame-Options header
  - Timing leaks the response size
• Use getTime() on img events
  - onLoad
  - Onreadystatechange (IE)
HTTP Payload size may carry sensitive information

- Moreover, HTTP payload size **differences** detection is sufficient to extract the sensitive information.

Using timing measurements attackers can distinguish HTTP payload size differences.

These timing measurement can be done with javascript on attacker site.

Result - attackers can learn the user’s sensitive information using javascript from their site, with no eavesdropping!
• SOP = Same Origin Policy
• “SOP - a mechanism that governs the ability for JavaScript and other scripting languages to access DOM properties and methods across domains”
• In order to prevent malicious scripts served from the attacker site to leak data from other site - browsers apply the Same Origin Policy (SOP)
Simple multimedia tags are exempt from SOP
  - Fetching images from other domains is OK

Enabling the Img src manipulation by a javascript does not seem to change the model

However, due to automation, it does
  - Interactively setting the URL
  - Measuring load time

Breaks SOP – allow data leak from one domain to another
Conclusions & mitigations
Key Contributions

- Resurrecting the CRIME attack with Extended CRIME attack against responses
- Introduced TIME attack to launch size diff attacks with **no eavesdropping requirement**
  - Original CRIME
  - Extended CRIME
  - Login detection
  - Application specific – e.g. # of digits in bank balance
Non-Mitigations

- Add Random Time Delays
- Lucky thirteen authors: “A natural reaction to timing based attacks is to add random time delays... to frustrate statistical analysis. In fact, this countermeasure is surprisingly ineffective”
• Browser should support and respect “X-Frame-Options” header for all content inclusion (not just IFRAME)
• By thus, allowing applications to take control over the presentation of their content on other domains
Applications

- Take control over your content
  - Implement CSRF protection
  - Use the `X-Frame-Options` header
- Beware of unknown parameters
- Deploy anti-automation measures
Questions
2002: Scientific paper, Kelsey
   - “Compression and Information Leakage of Plaintext”
2012: CRIME, Rizzo+Duong
   - Actual exploit for HTTP requests
3.2013: TIME, Be’ery + Shulman
   - Actual exploit for HTTP responses
   - Trading MITM with timing inference
7.2013: BREACH, Gluck+Harris+Prado
   - Actual exploit for HTTP responses