Timing Attack
從入門到發現沒時間了

Allen Chou

Aug. 20, 2022

關於這場 Workshop

- 目標客群：沒聽過 Timing attack 或聽過但不太確定那是啥的人
- 背景知識：會最基本的 programming，有高中以上的數學程度
- 會有 Hand-on 但不要求要做出來

如果現在驚覺這場 workshop 不適合你，塊陶 R

如果我有講錯，請大膽地舉手讓大家（包含我）知道。

1. 背景知識：至少有基本的機率與統計概念
2. Hand-on：應該會有一兩個，如果你能在短短幾分鐘內完美地寫出來，那這場 workshop 對你來說可能太簡單了

什麼是 Timing Attack

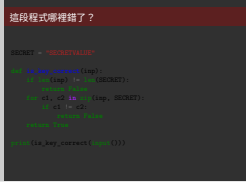
什麼是 Timing Attack

這段程式哪裡錯了？

```python
SECRET = "SECRETVALUE"

def is_key_correct(inp):
    if len(inp) != len(SECRET):
        return False
    for c1, c2 in zip(inp, SECRET):
        if c1 != c2:
            return False
    return True

print(is_key_correct(input()))
```

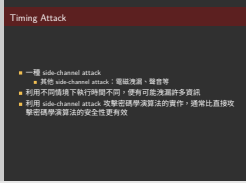
執行時間？

```python
print('INCORRECT_LENGTH:', timeit.timeit(
    lambda: is_key_correct('INCORRECT_LENGTH')
    , number=N) / N * 10e6)
print('SECRETABCDE:      ', timeit.timeit(
    lambda: is_key_correct('SECRETABCDE')
    , number=N) / N * 10e6)
print('SECRETVALUE:      ', timeit.timeit(
    lambda: is_key_correct('SECRETVALUE')
    , number=N) / N * 10e6)
```

```
[allen@allen-pc timing-attack]$ python3 test.py
INCORRECT_LENGTH: 1.1840343299991218
SECRETABCDE:      5.092783100008091
SECRETVALUE:      6.626864189966
[allen@allen-pc timing-attack]$
```


Timing Attack

- 一種 side-channel attack
  - 其他 side-channel attack：電磁洩漏、聲音等
- 利用不同情境下執行時間不同，便有可能洩漏許多資訊
- 利用 side-channel attack 攻擊密碼學演算法的實作，通常比直接攻擊密碼學演算法的安全性更有效



1. side channel 是一種利用電腦系統的 implementation 缺陷而洩漏的資訊來做的攻擊，而不是利用演算法本身的弱點。
2. 舉個例子，假設現在我發明並實作了一個密碼學演算法，他可以拿一把 key 加密再解密
3. 這個演算法有瑕疵，例如我可以藉由密文的 pattern 回推明文，這是密碼學演算法本身的缺陷，屬於 cryptanalysis（密碼分析）領域
4. 我設計的演算法非常成功，沒有人發現有問題，但我程式寫得爛透了會 overflow 以致可以 dump 出我的 key，這是軟體開發的錯誤
5. 如果我程式沒有出錯，但在正常執行過程中，卻會洩漏一些資訊，以致攻擊者可以利用這些資訊來還原出機密資料，這是 side-channel attack

所以剛剛那段程式碼的問題在哪？

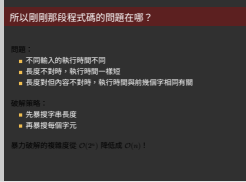
問題：

- 不同輸入的執行時間不同
- 長度不對時，執行時間一樣短
- 長度對但內容不對時，執行時間與前幾個字相同有關

破解策略：

- 先暴搜字串長度
- 再暴搜每個字元

暴力破解的複雜度從 $\mathcal{O}(2^n)$ 降低成 $\mathcal{O}(n)$！

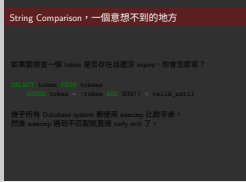


String Comparison，一個意想不到的地方

如果要檢查一個 token 是否存在且還沒 expire，你會怎麼寫？

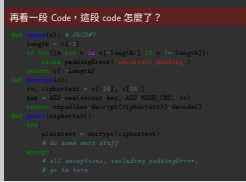
```
SELECT token FROM tokens
    WHERE token = :token AND NOW() < valid_until
```

幾乎所有 Database system 都使用 memcmp 比較字串，
然後 memcmp 遇到不匹配就直接 early exit 了。



再看一段 Code，這段 code 怎麼了？

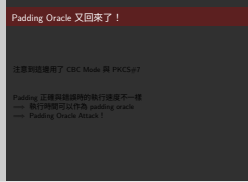
```python
def unpad(c): # PKCS#7
    length = c[-1]
    if len([b for b in c[-length:] if b != length]):
        raise paddingError('incorrect padding')
    return c[:-length]
def decrypt(c):
    iv, ciphertext = c[:16], c[16:]
    aes = AES.new(secret.key, AES.MODE_CBC, iv)
    return unpad(aes.decrypt(ciphertext)).decode()
def parse(ciphertext):
    try:
        plaintext = decrypt(ciphertext)
        # do some more stuff
    except:
        # all exceptions, including paddingError,
        # go to here.
```

Padding Oracle 又回來了！

注意到這邊用了 CBC Mode 與 PKCS#7

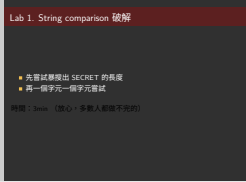
Padding 正確與錯誤時的執行速度不一樣
$\Longrightarrow$ 執行時間可以作為 padding oracle
$\Longrightarrow$ Padding Oracle Attack！



Lab 1. String comparison 破解

- 先嘗試暴搜出 SECRET 的長度
- 再一個字元一個字元嘗試

時間：3min （放心，多數人都做不完的）

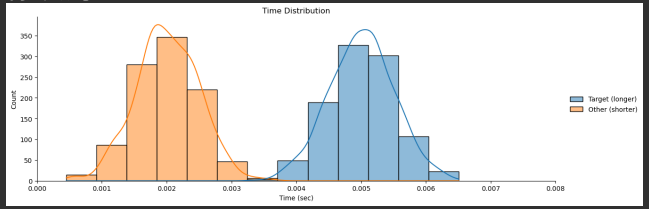


Classifier

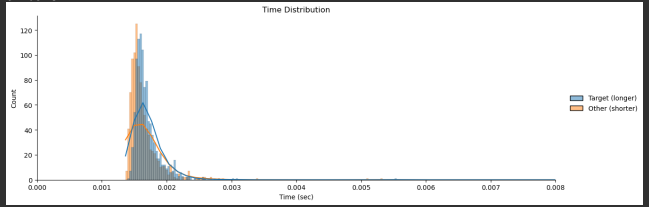
Classifier

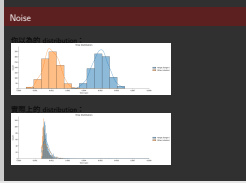
Noise

你以為的 distribution：



實際上的 distribution：





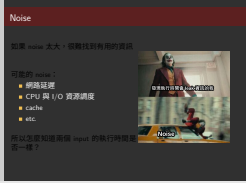
Noise

如果 noise 太大，很難找到有用的資訊

可能的 noise：

- 網路延遲
- CPU 與 I/O 資源調度
- cache
- etc.

所以怎麼知道兩個 input 的執行時間是否一樣？



問題

現在有兩個 payload 的執行時間的 distribution
如何知道兩個 payload
執行時間是否一樣？如果不一樣，哪個比較快？

學過統計的：t-test 炸下去，p-value < 0.05，賀！
沒學過統計的：蛤？

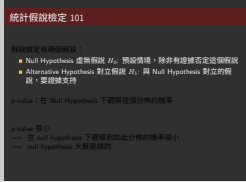
統計假說檢定 101

假說檢定有兩個假設：

- Null Hypothesis 虛無假說 H_0: 預設情境，除非有證據否定這個假說
- Alternative Hypothesis 對立假說 H_1: 與 Null Hypothesis 對立的假說，要證據支持

p-value：在 Null Hypothesis 下觀察這個分佈的機率

p-value 很小
$\Longrightarrow$ 在 null hypothesis 下觀察到如此分佈的機率很小
$\Longrightarrow$ null hypothesis 大概是錯的



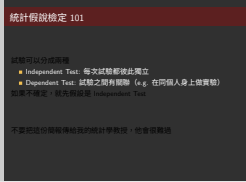
統計假說檢定 101

試驗可以分成兩種

- Independent Test: 每次試驗都彼此獨立
- Dependent Test: 試驗之間有關聯（e.g. 在同個人身上做實驗）

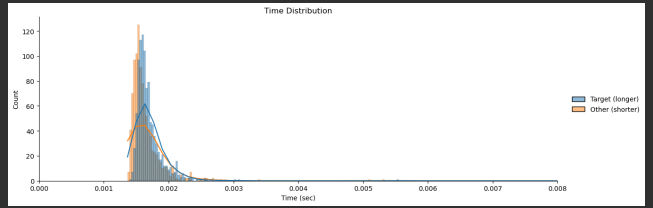
如果不確定，就先假設是 Independent Test

不要把這份簡報傳給我的統計學教授，他會很難過

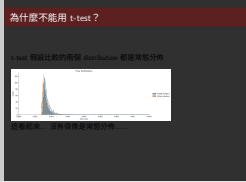


為什麼不能用 t-test？

t-test 假設比較的兩個 distribution 都是常態分佈



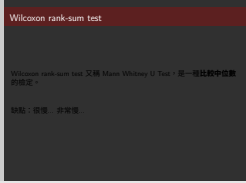
這看起來... 沒有很像是常態分佈.....



Wilcoxon rank-sum test

Wilcoxon rank-sum test 又稱 Mann Whitney U Test，是一種**比較中位數**的檢定。

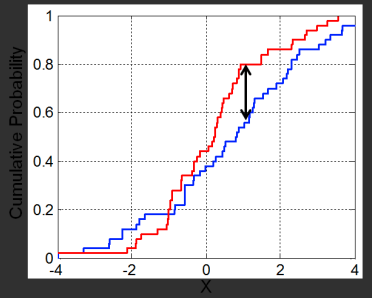
缺點：很慢... 非常慢...



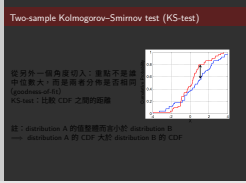
1. 無母數檢定 nonparametric test：當 population 分佈未知、不為常態分布，或樣本數很少時使用

Two-sample Kolmogorov–Smirnov test (KS-test)

從另外一個角度切入：重點不是誰中位數大，而是兩者分佈是否相同 (goodness-of-fit)
KS-test：比較 CDF 之間的距離



註：distribution A 的值整體而言小於 distribution B
$\Longrightarrow$ distribution A 的 CDF 大於 distribution B 的 CDF

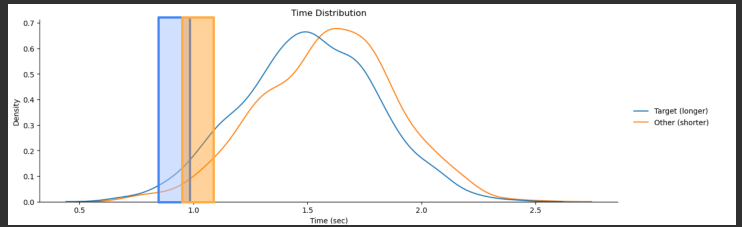


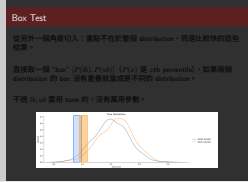
Box Test

從另外一個角度切入：重點不在於整個 distribution，而是比較快的這些結果。

直接取一個 “box” $[P(lb), P(ub)]$（$P(x)$ 是 xth percentile），如果兩個 distribution 的 box 沒有重疊就當成是不同的 distribution。

不過 lb, ub 要用 tune 的，沒有萬用參數。





1. 比較慢的結果可能是外在因素（網路的 jitter、CPU 剛好被搶走之類的）導致的，這些數字沒有參考價值。
2. 如果只比較最快的，可能會被特例影響 e.g. 剛好這次有 cache，所以不能單看最快的請求，而是要看分佈中靠左的數字
3. 通常 box 越寬 false positive 越高，越窄 false negative 越高

Box Test

Morgan & Morgan 提出一個找 Box Test 參數的演算法：

1. 嘗試在 box width $w = 1$ 下測試 lower bound $lb = 0$ 到 50 的 accuracy，選出 accuracy 最高的前五名。
2. 針對這五個潛在 lb，嘗試 $w = [0.5, 6]$ 並找到最能平衡 FP 與 FN (即 |FP-FN| 最小) 的 box width w^*。
3. 使用新的 box width w^* 重複 step 1，找到 accuracy 最高的 lower bound lb^*，令其為最終的 lower bound。
4. 使用 lb^* 嘗試在 $w^* \pm 0.7$ 之間，找到最能平衡 FP 與 FN 的 box width，令其為最終的 box width。

Box Test

Morgan & Morgan 提出一個找 Box Test 參數的演算法：

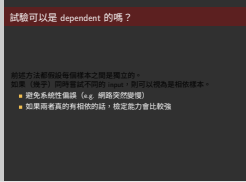
1. 嘗試在 box width $w = 1$ 下測試 lower bound $lb = 0$ 到 50 的 accuracy，選出 accuracy 最高的前五名。
2. 針對這五個潛在 lb，嘗試 $w = [0.5, 6]$ 並找到最能平衡 FP 與 FN (即 |FP-FN| 最小) 的 box width w^*。
3. 使用新的 box width w^* 重複 step 1，找到 accuracy 最高的 lower bound lb^*，令其為最終的 lower bound。
4. 使用 lb^* 嘗試在 $w^* \pm 0.7$ 之間，找到最能平衡 FP 與 FN 的 box width，令其為最終的 box width。

1. 其實就是類似於 gradient descent 的概念，先快速找到好的 lower bound 與 box width，再去 fine tune 數字

試驗可以是 dependent 的嗎？

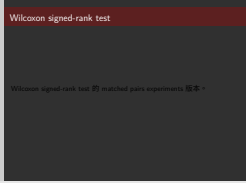
前述方法都假設每個樣本之間是獨立的。
如果（幾乎）同時嘗試不同的 input，則可以視為是相依樣本。

- 避免系統性偏誤（e.g. 網路突然變慢）
- 如果兩者真的有相依的話，檢定能力會比較強



Wilcoxon signed-rank test

Wilcoxon signed-rank test 的 matched pairs experiments 版本。



基於 rank-sum 概念的 classifier

如果要暴搜的值不是 binary 的呢？
其實只要把每次試驗的 rank 抓出來比較就可以了。

1. 每個可能的值都送一次請求，把時間做排名。
2. 上述步驟重複 N 次，並取 N 次試驗排名的平均。
3. 找排名平均最大/最小的值。

L-Estimators

L-Estimators 是 order statistics 的線性組合。

常見的 L-Estimators：

- 中位數（直接取 50th percentile）
- Midhinge（25th percentile 與 75th percentile 的平均）
- Trimean （$\frac{Q_1+2Q_2+Q_3}{4}$）

有些（非全部）L-Estimators 可以用來衡量 central tendency。

L-Estimators

如果同時測不同 input，只需要知道兩種 input 的時間差異是否接近 0 即可
$\Longrightarrow$ 檢查時間差的 central tendency 是否接近 0
$\Longrightarrow$ 可以使用 L-Estimator

只要 L-Estimator 所估測的時間差的 central tendency 低於一定的 threshold，就視為時間差為 0，即兩者相等。

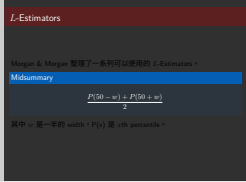
L-Estimators

Morgan & Morgan 整理了一系列可以使用的 L-Estimators。

Midsummary

$$\frac{P(50-w)+P(50+w)}{2}$$

其中 w 是一半的 width，P(x) 是 xth percentile。



Lab 2. 實作基於 rank-sum 概念的 classifier

自己動手實作一次基於 timing attack 的 padding oracle attack 吧。

時間：5 分鐘（大概會做不完，沒關係的）
https://reurl.cc/m3DdQj

Mitigation

Mitigation

如何解決 Timing Attack

Timing Attack 三要素：

1. 不同 input 會有不同執行時間
2. input 是使用者可控的
3. 可以聽到有效的 timing leakage

策略一：Constant-time implementation

只要使不同 input 都有相同執行時間，就沒有資訊洩漏了！

~~看吧，一個沒有時間資訊會洩漏的世界完成了。~~

String Comparison, Revisited

```python
SECRET = "SECRETVALUE"

def is_key_correct(inp):
    if len(inp) != len(SECRET):
        return False
    for c1, c2 in zip(inp, SECRET):
        if c1 != c2:
            return False
    return True

print(is_key_correct(input()))
```


String Comparison, Revisited

避免 Fast-failing 就可以了對吧？

```python
def is_key_correct(inp):
    result = True
    if len(inp) != len(SECRET):
        result = False
    for c1, c2 in zip(inp, SECRET):
        if c1 != c2:
            result = False
    return result

print(is_key_correct(input()))
```

Well... it depends.

如何被 Compiler 與 CPU 背刺

- Compiler（包含 JIT compiler）非常擅長 optimization，有可能在 optimize 的過程中把 side-channel 放回去
  e.g. 重新加入 Fast-failing。
- 在高階語言看起來像是一行 code，在指令層級可能不是 constant-time 的。
  e.g. short-circuit evaluation
- CPU 的 speculative execution 也可能重新創造 side-channel。
- 在 virtual memory address 看起來就在隔壁的 address 實際上可能在完全不同的地方。

如何被 Compiler 與 CPU 背刺
- Compiler（包含 JIT compiler）非常擅長 optimization，有可能在 optimize 的過程中把 side-channel 放回去
  e.g. 重新加入 Fast-failing。
- 在高階語言看起來像是一行 code，在指令層級可能不是 constant-time 的。
  e.g. short-circuit evaluation
- CPU 的 speculative execution 也可能重新創造 side-channel。
- 在 virtual memory address 看起來就在隔壁的 address 實際上可能在完全不同的地方。

1. branch prediction：先 train 好 CPU，之後看有沒有 branch prediction 有沒有錯就知道 input 是什麼了
2. virtual memory address：可能兩個 address 在不同條 RAM 或一個在 swap 裡面，於是不同情境就有不同執行時間了

String Comparison

```python
def is_key_correct(inp):
    result = 0
    for c1, c2 in zip(inp, SECRET):
        result |= c1 ^ c2

    return result == 0 & len(inp) == len(SECRET)

print(is_key_correct(input()))
```


1. 那個 & 是為了避免 short circuit evaluation
2. 這個寫法還是會洩漏 SECRET 長度

Padding Oracle, Revisited

所以想修好剛剛的 Padding Oracle Attack，只需要確保 constant-time 就可以了... 對嗎？

不是，在那之前，先保護 ciphertext 的 integrity 吧！

$\Longrightarrow$ 使 input 不是使用者可任意操作的。

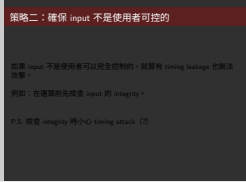
1. 用 AES-GCM 或任何 Encrypt-then-MAC 的方法保護 ciphertext
2. 就算有 timing leakage，攻擊者也沒辦法操控 input

策略二：確保 input 不是使用者可控的

如果 input 不是使用者可以完全控制的，就算有 timing leakage 也無法攻擊。

例如：在運算前先檢查 input 的 integrity。

P.S. 檢查 integrity 時小心 timing attack（?）



1. 例如用 == 檢查 checksum，或是根據 input 去戳 memory address（造成資料被寫入 cache 或造成 page fault），都可能造成額外的 timing attack

String Comparison, revisited, again.

```python
def is_key_correct(inp):
    return hmac(inp, KEY) != hmac(SECRET, KEY)

print(is_key_correct(input()))
```

就算攻擊者知道 `HMAC(inp, KEY)` 的前 n 個 bit 是正確的，也無法有效率地構造出使 `HMAC(inp', KEY)` 的前 $(n+1)$ 個 bit 符合預期的 `inp'`。

策略三：確保 timing leakage 不可測

對回應時間加上 random noise？
$\Longrightarrow$ 多測幾次就好了，而且是效能地獄

1. random noise 無法在理論上避免攻擊，但可能會讓攻擊成本高到難以實行

String Comparison，更好的解法

使出大絕：用別人寫好的程式。

- PHP：`hash_equals`, `password_verify`
- Java: `MessageDigest.isEqual`
- Python: `hmac.compare_digest`
- Go: 在 `crypto/subtle` 裡面的 `ConstantTime` 系列

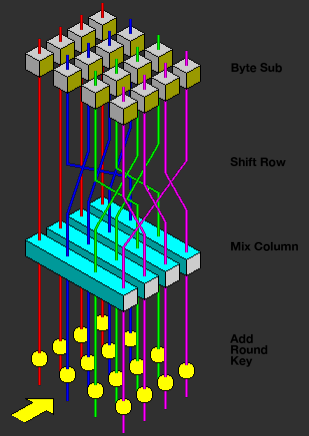
密碼學中的 Timing Attack

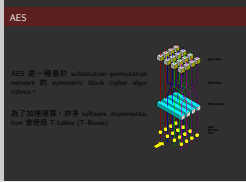
密碼學中的 Timing Attack

AES

AES 是一種基於 substitution–permutation network 的 symmetric block cipher algorithms。

為了加速運算，許多 software implementation 會使用 T-tables (T-Boxes)





AES 與 Cache-timing Attack

存取 T-tables 的 index 與 key 以及 plaintext 有關，
然後存取一個 memory address 的會導致資料被寫進 cache，
而且存取一個 memory address 的速度與是否有 cache-miss 有關，

所以...

聽起來像是 Timing Attack 呢。

Mitigation

讓存取 lookup table 變成 constant-time！

解法一：把四張 T-tables 都塞進 L1 cache 然後確保他不會被 flush 掉。
問題：在 time-sharing OS 上不可行。

解法二：硬體支援（e.g. AES-NI）
如果要又安全又有效率的 AES，只能訴諸硬體支援。

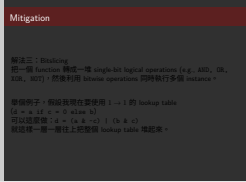
Mitigation
讓存取 lookup table 變成 constant-time！
解法一：把四張 T-tables 都塞進 L1 cache 然後確保他不會被 flush 掉。
問題：在 time-sharing OS 上不可行。
解法二：硬體支援（e.g. AES-NI）
如果要又安全又有效率的 AES，只能訴諸硬體支援。

1. djb 給的解法是：確保 AES computation 不會被 interrupt
2. 基本上現代的 CPU 都已經有內建 AES 硬體加速了

Mitigation

解法三：Bitslicing
把一個 function 轉成一堆 single-bit logical operations (e.g., `AND`, `OR`, `XOR`, `NOT`)，然後利用 bitwise operations 同時執行多個 instance。

舉個例子，假設我現在要使用 $1 \to 1$ 的 lookup table
(`d = a if c = 0 else b`)
可以這麼做：`d = (a & ~c) | (b & c)`
就這樣一層一層往上把整個 lookup table 堆起來。

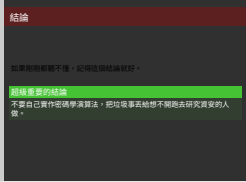


結論

如果剛剛都聽不懂，記得這個結論就好。

超級重要的結論

不要自己實作密碼學演算法，把垃圾事丟給想不開跑去研究資安的人做。



Browser 中的 Timing Attack

Browser 中的 Timing Attack

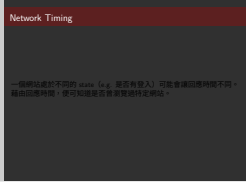
Same-origin policy 與 XS-Leaks

Same-origin policy（同源政策）規範不同 origin （網域）的 script 與 document 可以如何互動。
所以你瀏覽我的網站，不會讓我有辦法讀到你的信箱。

Cross-site leaks (XS-Leaks) 是指利用 side-channel attack 來洩漏本來被 SOP 保護的資源。

Network Timing

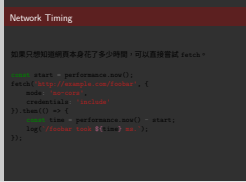
一個網站處於不同的 state（e.g. 是否有登入）可能會讓回應時間不同。
藉由回應時間，便可知道是否曾瀏覽過特定網站。



Network Timing

如果只想知道網頁本身花了多少時間，可以直接嘗試 `fetch`。

```
const start = performance.now();
fetch('http://example.com/foobar', {
    mode: 'no-cors',
    credentials: 'include'
}).then(() => {
    const time = performance.now() - start;
    log(`/foobar took ${time} ms.`);
});
```

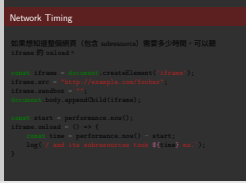



Network Timing

如果想知道整個網頁（包含 subresource）需要多少時間，可以聽 iframe 的 onload。

```
const iframe = document.createElement('iframe');
iframe.src = "http://example.com/foobar";
iframe.sandbox = "";
document.body.appendChild(iframe);

const start = performance.now();
iframe.onload = () => {
    const time = performance.now() - start;
    log(`/ and its subresources took ${time} ms.`);
}
```

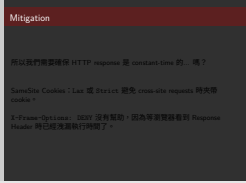
1. onload 會在所有 resource 都下載完且 JS 都執行完才觸發，加上 sandbox 屬性會阻擋 JS 執行，所以會更趨近於單純的 network measurement

Mitigation

所以我們需要確保 HTTP response 是 constant-time 的... 嗎？

SameSite Cookies：`Lax` 或 `Strict` 避免 cross-site requests 時夾帶 cookie。

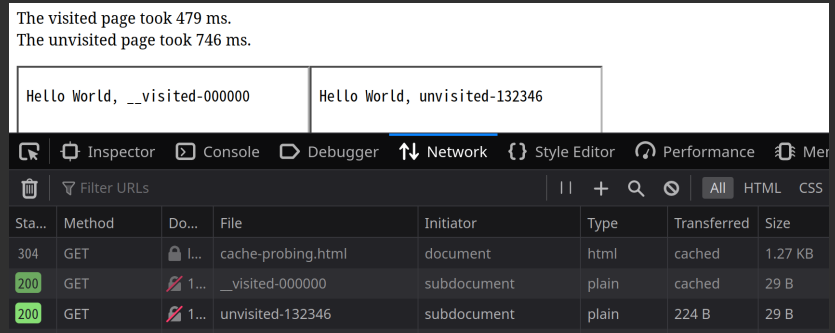
`X-Frame-Options: DENY` 沒有幫助，因為等瀏覽器看到 Response Header 時已經洩漏執行時間了。

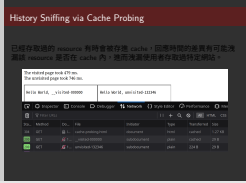


1. 兩者差在 cross-site top-level navigation 時 Strict 不會帶 cookie（所以會破壞很多網站）但 Lax 會

History Sniffing via Cache Probing

已經存取過的 resource 有時會被存進 cache，回應時間的差異有可能洩漏該 resource 是否在 cache 內，進而洩漏使用者存取過特定網站。

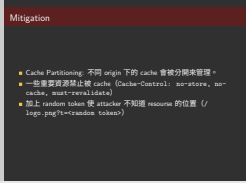




1. 用 cache 讀到 geolocation 的例子
2. 這方法很不準確，例如從 cache 拿資料怎麼可能需要快 500ms，這是 JS 的限制，但多測幾次效果還是不錯
3. 其實有更準確且更有效率的 cache probing

Mitigation

- Cache Partitioning: 不同 origin 下的 cache 會被分開來管理。
- 一些重要資源禁止被 cache（`Cache-Control: no-store, no-cache, must-revalidate`）
- 加上 random token 使 attacker 不知道 resourse 的位置（`/logo.png?t=<random token>`）



Conclusion

- Timing leakage 難以偵測且不好修復。當一個程式並非 constant-time 時，應警覺有沒有可能洩漏資訊。
- 但並非所有 timing leakage 都是有用的，不用看到不是 constant-time 的程式就在尖叫。
- 密碼學好難，交給懂的人去處理就好。

Conclusion

- Timing leakage 難以偵測且不好修復。當一個程式並非 constant-time 時，應警覺有沒有可能洩漏資訊。
- 但並非所有 timing leakage 都是有用的，不用看到不是 constant-time 的程式就在尖叫。
- 密碼學好難，交給懂的人去處理就好。

References I

1. Bernstein, D. J. Cache-timing attacks on AES. (2005).
2. Boneh, D. & Shoup, V. A graduate course in applied cryptography. Draft 0.5 (2020).
3. Contributors. XS-Leaks Wiki. `https://xsleaks.dev/` (2022).
4. Crosby, S. A., Wallach, D. S. & Riedi, R. H. Opportunities and limits of remote timing attacks. ACM Transactions on Information and System Security (TISSEC) **12,** 1–29 (2009).
5. Jia, Y., Dong, X., Liang, Z. & Saxena, P. I Know Where You've Been: Geo-Inference Attacks via the Browser Cache. IEEE Internet Computing **19,** 44–53 (2015).
6. Mayer, D. & Sandin, J. Time Trial: Racing Towards Practical Remote Timing Attacks. (2014).

References I

1. Bernstein, D. J. Cache-timing attacks on AES. (2005).
2. Boneh, D. & Shoup, V. A graduate course in applied cryptography. Draft 0.5 (2020).
3. Contributors. XS-Leaks Wiki. https://xsleaks.dev/ (2022).
4. Crosby, S. A., Wallach, D. S. & Riedi, R. H. Opportunities and limits of remote timing attacks. ACM Transactions on Information and System Security (TISSEC) **12,** 1–29 (2009).
5. Jia, Y., Dong, X., Liang, Z. & Saxena, P. I Know Where You've Been: Geo-Inference Attacks via the Browser Cache. IEEE Internet Computing **19,** 44–53 (2015).
6. Mayer, D. & Sandin, J. Time Trial: Racing Towards Practical Remote Timing Attacks. (2014).

References II

7. Morgan, T. D. & Morgan, J. W. Web timing attacks made practical. Black Hat (2015).
8. P.I.E.Staff. Split Tokens: Token-Based Authentication Protocols without Side-Chan https://paragonie.com/blog/2017/02/split-tokens-token-based-authentication-protocols-without-side-channels.
9. Paar, C. & Pelzl, J. Understanding cryptography: a textbook for students and practitioners. (Springer Science & Business Media, 2009).
10. Soatok. Soatok' s Guide to Side-Channel Attacks. https://soatok.blog/2020/08/27/soatoks-guide-to-side-channel-attacks/.

References II

7. Morgan, T. D. & Morgan, J. W. Web timing attacks made practical. Black Hat (2015).
8. P.I.E.Staff. Split Tokens: Token-Based Authentication Protocols without Side-Chan https://paragonie.com/blog/2017/02/split-tokens-token-based-authentication-protocols-without-side-channels.
9. Paar, C. & Pelzl, J. Understanding cryptography: a textbook for students and practitioners. (Springer Science & Business Media, 2009).
10. Soatok. Soatok' s Guide to Side-Channel Attacks. https://soatok.blog/2020/08/27/soatoks-guide-to-side-channel-attacks/.

References III

11. Soatok.
Timing Attack on SQL Queries Through Lobste.rs Password Reset.
https://soatok.blog/2021/08/20/lobste-rs-password-reset-vulnerability/.

12. tanglei.name. 计时攻击 Timing Attacks.
https://coolshell.cn/articles/21003.html.

13. Taubert, T. Bitslicing, An Introduction.
https://timtaubert.de/blog/2018/08/bitslicing-an-introduction/.

References III

11. Soatok
Timing Attack on SQL Queries Through Lobste.rs Password Reset.
https://soatok.blog/2021/08/20/lobste-rs-password-reset-vulnerability/

12. tanglei.name. 计时攻击 Timing Attacks.
https://coolshell.cn/articles/21003.html.

13. Taubert, T. Bitslicing, An Introduction.
https://timtaubert.de/blog/2018/08/bitslicing-an-introduction/

The End