0x00 前言

验证码的初衷是人机识别。不过大多时候，只是用来增加一些时间成本，降低频率而已。

如果仅仅是为了消耗时间，能否不用图片，完全用程序来实现？

0x01 如何消耗时间

先来思考一个问题：写一个能消耗对方时间的程序。

消耗时间还不简单，休眠一下就可以了：

```
Sleep(1000)
```

这确实消耗了时间，但并没有消耗 CPU。如果开了变速齿轮，瞬间就能完成。

要消耗 CPU 也不难，写一个大循环就可以了：

```
for i = 0 to 1000000000
end
```

不过这和 Sleep 并无本质区别。对方究竟有没有运行，我们从何得知？

所以，我们需要一个返回结果 —— 只有完整运行才有正确答案。

```
result = 0
for i = 0 to 1000000000
    result = result + i
end
return result
```

通过返回的结果，我们就能判断对方是否完整的运行了程序。

不过上面这个问题，毕竟还是 too simple。小学生都知道，用数列公式就可以直接算出结果，根本不用花时间去跑循环。

那么有什么算法，是无法用公式推测的？

显然，单向散列函数就是。例如一个经典的问题：

```
MD5(X) == X
```

就无法用公式来解决了。要找出答案，只能一个个穷举过去，从而花费大量时间。

但对于验证者，只需将收到的答案，计算一次就可判断对错，因此可轻易校验。

这就是 PoW（Proof-of-Work），用来证明对方投入工作的方法。

当然，上面的例子太困难了，而且答案可以重复使用，所以还需改进。例如：

```
MD5("问题" + X) == "0000......"
```

我们只要求散列结果的前几位是 0 就可以。这样位数越小，答案就越容易找到。同时增加一个盐值，让答案不能重复使用。

事实上，比特币就用到了类似的方式，使用了 SHA-256 作为散列函数。这样只能穷举，无法用更快的方法投机取巧，体现了挖矿工作的价值。

用散列函数实现的 PoW，就叫 Hashcash (https://en.wikipedia.org/wiki/Hashcash)。

0x02 传统应用

Hashcash 早不是新鲜事，曾经在反垃圾邮件中就已使用。

例如用户写完邮件时，客户端将「收件地址 + 邮件内容」作为 Salt，然后计算符合条件的答案：

```
Hash(X, Salt) == "000000..."
```

最后将找到的 X 附加在邮件中并发送。服务端收到后，即可鉴定发送这封邮件，是否花费了计算工作。

对于正常用户来说，额外的几秒并不影响使用；但对于制造垃圾邮件的人，就大幅增加了成本。

传统策略，大多通过 IP、账号等限制。攻击者可以用大量的马甲和代理，来绕过这些限制。

而使用了 PoW，就把瓶颈限制在硬件上 —— 计算有多快，操作才能多快。

0x03 Web 应用

同样的，Hashcash 也能用于 Web。例如论坛，可在发帖时计算：

```
Hash(X, 帖子内容) == "000000..."
```

不过，不同于邮件客户端可在后台自动计算，发帖时如果卡上好几秒，将会大幅降低用户体验。

因此不能选择 帖子内容、标题 等这些用户输入作为盐值。而是用传统验证码的方式，后端下发一个随机数。

前端使用这个随机数作为盐值 —— 这样页面打开时，就可以开始计算了。

```
# 后端 - 分配
```

```
session["pow_code"] = rand()
.
# 前端 - 挖矿
while Hash(X, pow_code) == "000000..."
    X = X + 1
end
```

我们选择一个适中的难度，例如 10 秒。通过多线程，还可以更快的完成计算任务，同时不影响用户体验。

正常情况下，用户发帖前就已计算完成。提交时，将其附带上。

如果提交时还未算出，则等待计算完成。（发帖太快，有灌水嫌疑）

```
# 前端 - 提交
wait X
submit(..., X)

# 后端 - 校验
if Hash(X, session["pow_code"]) == "000000..."
    ok
else
    fail
end
0
```

这样，就实现了一个「测试机器算力」的验证码。

目前已有提供 hashcash 第三方验证的网站，例如 hashcash.io (https://hashcash.io/)。

0x04 Web 性能

当然在 Web 中使用，性能也是一大问题。如果 10 秒的脚本计算，用本地程序只需 1 秒，那攻击者就可以使用本地版的外挂了。

好在如今有 asm.js，可接近原生性能；对于较老的浏览器，也可以使用 Flash 作后补。在上一篇文章 0x08 节 (http://www.cnblogs.com/index-html/p/frontend-slow-encrypt.html) 中已详细讲解。

如果算力实在不够，也可以使用后备方案 —— 传统图形验证码。

这样，高性能用户可享受更好的体验，低性能用户也能保障基本功能。

这也算是鼓励大家使用现代浏览器吧：）

0x05 致命缺陷

不过，语言上的性能差距还是有限的，外挂不会纠结于此，而是使用更强力的武器 —— GPU。

Hashcash 的本质就是跑 hash，这是 GPU 最擅长的。例如著名的 oclHashcat (http://hashcat.net/oclhashcat/)，和 CPU 完全不在一个数量级。

对抗硬件的并行计算，大致有如下方案和思路：

- 硬件瓶颈
- 移植难度
- CPU 算法
- 以暴制暴
- 代码混淆
- 串行模式

前 3 个在上一篇文章 0x09 节 (http://www.cnblogs.com/index-html/p/frontend-slow-encrypt.html) 提到了，下面讨论一些不同的。

0x06 以暴制暴

如果我们也能在 Web 中调用显卡计算，那 GPU 版的外挂就毫无优势了。

不过，这个想法似乎有些遥远。尽管目前主流浏览器都支持 WebGL，但都只局限于渲染加速上，并未提供通用计算接口。

当然，也可以通过一些 hack 的方式，例如曾有人尝试用 WebGL 挖比特币 (https://github.com/derjanb/hamiyoca)，但效率并不高。

如果未来 WebCL 成为标准，或许还能考虑。

0x07 代码混淆

上回讨论慢加密时，曾提到为什么要性能优化。因为自己创造加密算法是不推荐的，所以得优化现有的算法。

不过，相比账号安全，验证码的要求则低得多，而且随时可以更换算法，因此不妨自己来创造一个。

自创的加密算法，强度显然没有保障。但我们可以从「隐蔽性」上着手 —— 将代码混淆到难以读懂，这时，考验对方的则是逆向能力了。

这和之前写的《对抗假人 —— 前后端结合的 WAF》 (http://www.cnblogs.com/index-html/p/frontend-based-war.html)有点类似。不过，如果混淆能做到足够好，还需要 PoW 机制吗？

有胜于无。因为浏览器指纹、用户行为等信息，都是可以通过沙盒模拟的。而工作量计算，必须消耗硬件资源，才能得出结果。

因此，使用了 PoW 就能增加攻击者一些硬件成本。

0x08 串行模式

Hashcash 的原理，决定了它是可以并行计算的。有什么样的算法，是无法并行计算的？

如果每次计算都依赖上次结果，就无法并行了。例如之前讨论的 slowhash：

```
function slowhash(x)
    for i = 0 to 1000000000
        x = hash(x)
    end
    return x
end
```

这种串行的计算，自然是无法拆分的。但能用到 PoW 上吗？

显然不行！因为 PoW 虽然计算困难，但得 容易鉴定。而这种方式，鉴定时也得重复算一遍，成本太大了。

但在现实中，只要设计得当，还是可以尝试的 —— 我们使用类似 UGC 的模式，让用户来贡献算力！

首先需要一个访问量较大的网站，在其中悄悄放置一个脚本。利用在线的用户，来生成问题和答案。

```
# 隐蔽的脚本
Q = rand()
A = slowhash(Q)

submit(Q, A)
```

当然，这项工作必须足够隐蔽，防止被好奇的用户发现，提交错误的答案。

当后端题库有一定的积累时，就可以使用验证码的模式了。用户访问时，后端从题库中随机抽取一个问题，安排给前端计算：

```
# 后端 - 分配问题
Q = select_key_from_db()
session["pow_ques"] = Q

# 前端 - 计算问题
A = slowhash(Q)
```

用户提交时，后端无需任何计算，直接通过查表，即可判断答案是否正确：

```
# 前端 - 提交
submit(..., A)

# 后端 - 鉴定
Q = session["pow_ques"]
if A == db[Q]
    ok
else
    fail
end
0
```

使用预先计算的方式，避免了繁重的鉴定工作。同时，把计算交给用户来完成，可大幅节省硬件成本。

当然，这种模式还有很多需要考虑的地方，这里只是介绍下基本思路，以后再详细讨论。

> 相比 hashcash 题解时间有一定的随机性，slowhash 的时间是固定的，因此难度更可控。

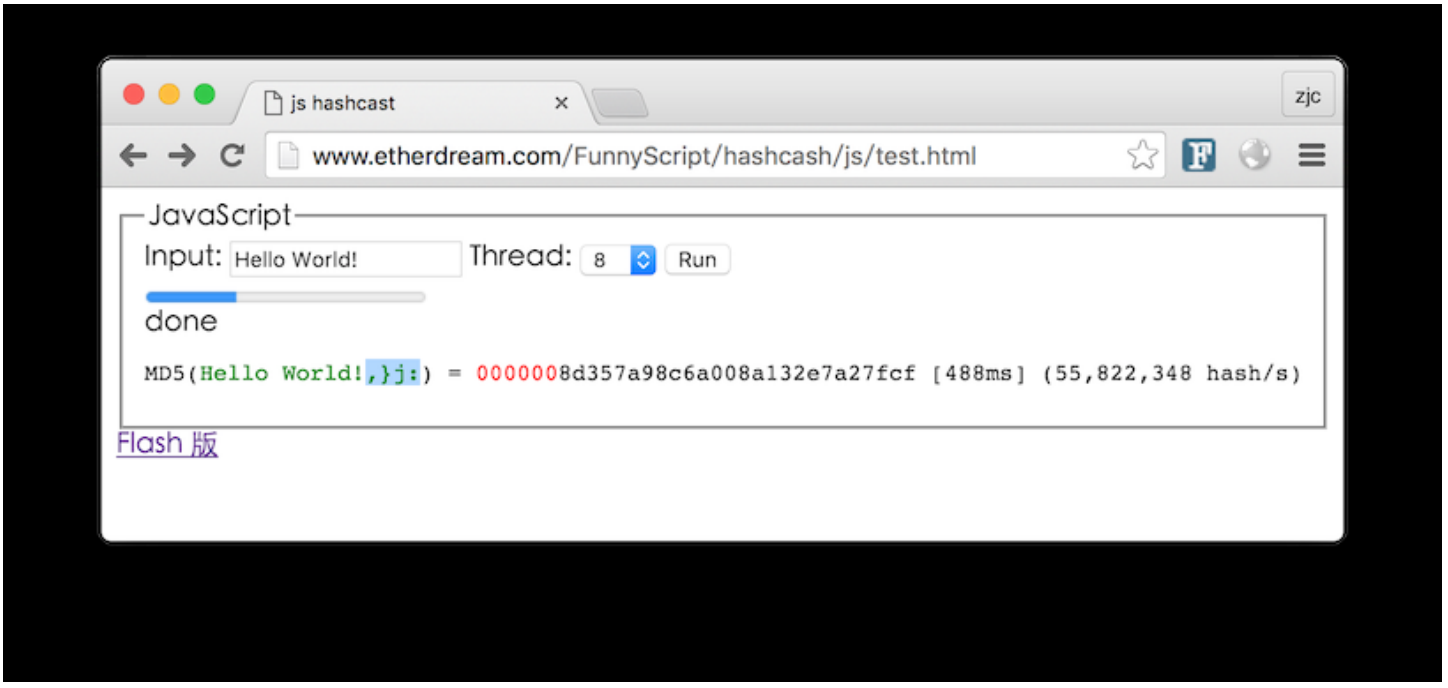
0x09 演示

因为 Hashcash 比较简单，所以这里演示一个 md5 版的，使用 asm.js 和 flash 实现，并对算法做了一定优化。

https://github.com/EtherDream/proof-of-work-hashcash (https://github.com/EtherDream/proof-of-work-hashcash)

如果想看详细的算力速度，可以查看这个 Demo：

http://www.etherdream.com/FunnyScript/hashcash/js/test.html
(http://www.etherdream.com/FunnyScript/hashcash/js/test.html)



看起来好像不慢，不过对比 GPU 的速度 (http://hashcat.net/oclhashcat/) 就相形见绌了。所以，使用经典算法的 Hashcash，简直就是不堪一击的。

至于串行模式的 PoW，涉及到很多策略和数据积累，本文就演示了，下回单独讨论。

0x0A 总结

最后来对比下，算力验证和传统图形验证的区别。

	验证方式	验证对象	用户体验	拦截假人
传统验证	图像识别	人脑	需要交互	部分拦截
算力验证	问题解答	电脑	无感知	无法拦截

论效果，当然还是传统的图形验证更好，但这是以牺牲用户体验为代价的。

硬件在不断的发展，识图软件会越来越强大。而人脑始终是有限的，优势会越来越小，最终导致验证码越来越复杂。

但是算力验证则不同。硬件的发展，也会带动浏览器的算力提升，最终只需将问题难度调高即可。

当然，安全防御涉及的领域越多越好。每一个方案都不是无敌的，都只是为了增加一些攻击成本而已。

所以算力验证，结合传统防御方案，才能出发挥价值。