//火狐教程：侧边栏视图、缩略图视图，**ReadMe**，**Firefox 入门教程**，**Firefox 帮助**，本文更新地址；
Site/App Isolation VS ABE
多浏览器/多配置模型：
Google团队在该报告中的贡献：
SSBs
在单浏览器中实现Site/App Isolation
多浏览器/配置的好处：
Site/App Isolation 的局限：
下面对表中的跨站型/非同源攻击进行详细解释：
哪些特点使但浏览器/配置模式会暴露如上这么多的漏洞呢？总结起来就3点：
定义Isolated Sites/Apps：
通过HTTP Header初始化：
Chrome的实现：
设计目标：
威胁模式：
Chrome实验性试用：

//chromium.org/site-isolation,
//chromium.org/isolated-sites,
//ccs-2011.pdf,
//该项目进展,

//Site/App Isolation VS ABE,

- 比如Site Isolation安全保护出于的2个关键地方：

入口限制：类似于ABE的一条规则：Accept from SELF++
状态信息隔离：类似于ABE的一条规则：Anon from *

- 再比如，Site Isolation使用2种策略组：

内置，需要下载：等价于ABE的第三方规则+系统规则
网站定义策略组：等价于ABE的网站规则

- 再比如，使用文件下载规则，相同

==========

多浏览器/多配置模型：
Site隔离还处于实验阶段的安全机制，因为真正实现站点隔离需要考虑到对内存的额外需求，沙盒的使用对性能的影响，Site信息传递等

安全专家一直建议使用多配置/浏览器上网：一个配置用来冲浪，一个配置用来登录敏感Sites；
从这个观点出发，产生2个疑问：多浏览器配置真的可以实现安全性能增强么，如果可以那么哪些属性起到关作用了？是否可以在同一个浏览器中实现多个配置共存?

很多安全机制都是互相借鉴思想的，Site/App Isolation与ABE的思想几乎是一样的，这一点知道读了下面这个论文后才会更加体会到：
(Firefox用户早就可以实现了，而且之前法和早 已经提出了这种思想)

以反射型XSS举例，攻击者制作一个恶意的URL，其中包含了一串攻击代码，然后把用户的浏览器导航到该URL，诱使合法Site回应该恶意攻击代码。
如果使用了2个配置，那么当用非敏感配置浏览时，某个银行网站上的一个恶意URL是无法获取到用户的帐户信息的。

多浏览器/多配置的好处主要来自2个：状态信息隔离、入口限制

- 状态信息隔离：一个浏览器内的代码无法获取和共享另一个浏览器的同意Site的状态/授权信息
- 入口限制：即从一个浏览器进入某个敏感Site的方式是有限的，从而避免了之前那种反射型URL方式进入某个敏感Site的可能，从而也进一步确保了状态信息的隔离

状态信息隔离是很关键/重要的，比如历史嗅探-History Sniffing、缓存时间攻击-cache timing attacks等攻击无须通过恶意URL代码来实现，也就是说关于入口限制是不够的，比如使用状态信息隔离技术。
不仅是这些Web应用的漏洞，状态信息隔离甚至还能保护敏感Site免受对浏览器漏洞的利用。

此外，Site隔离的好处还有对多Sites的访问控制上，我们可以使用白名单来控制对于某些Sites的隔离的存储数据的请求及其入口。这能实现一系列保护。

该隔离机制的代价是兼容性问题，某些特殊类型Web Site可能会不正常，因为该机制对深度链接及第三方Cookie的限制，

为了评估该机制带来的好处，我们用Alloy语言描述了我们的协议模型。为了模范化我们的协议中的必要概念，我们丰富了之前在Alloy上的一些概念，比如EntryPoints、RenderingEngine等。
我们的分析得出了之前起草的协议存在2个问题：HTTP重定向、入口限制与状态信息隔离之间的一个无法预期的操作。

Google团队在该报告中的贡献：
-具体化了多浏览器/多配置的好处为2个基本概念
-给出了一个用单浏览器实现多浏览器/多配置的效果，并确保了一些类型Sites的兼容性/可操作性
-我们使用标准模型实现该机制的安全性，调整设计以便预防未知漏洞
-利用Mozilla的飞行员平台评估了该机制的兼容性，我们是首次使用该平台进行学术研究

SSBs
使用多浏览器/配置浏览Web的浏览器，我们称为SSB，及Stie-Specific Browser，典型的SSB有Mozilla的Prism和Fluid。
SSBs只是单纯为了Site-SPecific，可能会产生很严重的管理更难，特别是在大量Sites之间进行交互操作时；Site/App Isolation机制可以在一个浏览器中实现SSB浏览器的好处，同时也不会给用户管理带来严重负担。比如，该机制允许用户的通过一个链接-Link从一个Web应用无缝安全的进入另一个Web应用。
//何谓Prism浏览器，Fluid浏览器，

在单浏览器中实现Site/App Isolation
在一个浏览器中进行细化隔离的思想已经被很多研究者尝试过了，但都未总结出要实现单浏览器的细化隔离的关键难点。
现代浏览器都通过Sandbox来保护本地文件系统-Local File System免受浏览器漏洞利用，比如，IE在Vista系统上引入了保护模式-Protected Mode保护文件系统免受被控制/攻破的的渲染引擎-Compromised Render Engine修改。Google Chrome浏览器类似。可惜，所有这些沙盒机制都不能保护Web App信息状态，比如Cookie和其他本地存储，免受被控制的渲染引擎的攻击。

- OP浏览器通过强制限制对插件开放的跨站请求API-Cross-Origin API来隔离状态信息与插件。
- Gazelle浏览器则进一步限制了整个渲染引擎，从而防止了一个恶意Web对象-entity攻破其自己的渲染引擎再进一步获取其他Web应用的状态信息。可惜，Gazelle浏览器会阻止一切跨站资源请求，除非它们的MIME类型是库格式，比如JS、CSS，它对兼容性/可行性代价带大

办法：一个避免限制跨站资源请求API-Cross-Origin request API兼容性代价的办法就是允许一个Web应用准确声明它的组成URLs。

- Tahoma浏览器允许Web应用通过特定的mainfest文件列举一组保护在本域名保护范围的URL名单。Tahoma对每个Web应用使用一个独立的状态信息容器，所以一个Web应用的状态信息与另一个Web应用是隔离的，可惜，Tahoma未实现其他多浏览/配置的优点，比如阻止Non-Sensitive Site直接进入Sensitive Sites。
- OMash仅附带Cookie到同源请求上，有效的隔离了状态信息。每个进入新的Site都会创建一个新的会话。这个方法降低了反射型XSS-Rflected XSS，CSRF-Cross-Site Request Forgery和点击劫持-Click-Jacking，因为另外的Site无法通过超链接或者框架-Iframe来劫持一个活动的会话。可惜，OMash浏览器无法实现跨多会话的状态信息共享。

- CSP机制试图通过白名单(保存在一个外部JS文件上)来放行Web Sites上的脚本来预防XSS
- SOMA试图通过让Web Site手动授权Web内容嵌入者-Content Embedder，来减少XSS和CSRF攻击
- 可惜，上面这些机制都仅仅针对某类攻击，比如XSS或CSRF，它们不能完全实现多浏览器/配置的安全增强，比如防御对渲染引擎的利用。

多浏览器/配置的好处：
假设：
1、仅在浏览器A中输入Sensitive Site的帐号密码，从而确保Sensitive Site的状态信息仅保存在浏览器A中
2、绝不在浏览器A中点击书签或URLs跳转到Non-Sensitive Sites，也绝不在浏览器A中点击从Non-Sensitive Sites跳转到Sensitive Sites的链接，从而确保浏览器A不会泄露敏感信息给浏览器B，也不会被浏览器B篡改敏感信息。
如果用户严格遵照啊方面的2点，那么所有敏感信息都会只存在于浏览器A中，而与浏览器B隔离；此外，还可以维持浏览器A的整体性，因为浏览器B不能够影响浏览器A，从而免受反射型XSS攻击。

这2点规则可阻止下表所列的攻击的跨站式Cross-Orifin versions：

Cross-origin Attacks	Entry-point Restriction	State Isolation	Separate browser
Reflected XSS	✓		✓
Session Fixation	✓		✓
Cross-Origin Resource Import	✓	✓	✓
Click-jacking		✓	✓
CSRF	✓	✓	✓
Visited Link Sniffing		✓	✓
Cache Timing Attack		✓	✓
Rendering Engine Hi-jacking		✓	✓

Site/App Isolation 的局限：
我们定义为跨站式-Cross-Origin Versions是与同源式-Same-Origin Versions相对的：前者攻击来自非受害Sites，后者都来自受害Site，比如从一个Facebook的Page到另一个Facebook的Page的CSRF攻击；使用分开浏览器不仅不能阻止上述攻击的同源式-Same-Origin Versions of these Attacks，而且不能阻止仅同源类攻击，比如存储型/持久型XSS，因为这种攻击可以仅在受害浏览器实现。

下面对表中的跨站型/非同源攻击进行详细解释：
假设攻击者想要攻击一个用户已经授权的Web Site，并且有能力诱使受害者另一个非同源的恶意Site；
此外，假设用户按照上面的2个原则使用了多浏览器/配置：

- 反射型XSS-Relected XSS(Type 1 XSS)：攻击者诱使受害者访问一个恶意URL(在Non-Sensitive浏览器中)，该URL允许攻击者的脚本只受害者的源-Site执行；不过，用户对与该受害的源-Site的授权信息在另一个Sensitive浏览器中，所以攻击者的脚本无法获取到用户在该受害源-Site的授权信息。
- 会话锁定-Session Fixation：攻击者在一个受控制的URL中包含进一个已知的会话ID，然后诱使用户访问该URL，并骗用户登录，一旦用户登录后，攻击者就可利用该共享的会话ID伪装受害者身份；不过，因为受害者仅在Sensitive浏览器中输入帐号密码，所以攻击者无法通过会话ID共享授权信息。
- 跨域资源调用-Cross-Origin Resource Import：攻击者的Page请求一个来自预加害的Site的资源，比如一个脚本或样式。如果受害者已经在用一个浏览器中授权过了该Site，那么攻击者的请求就能获取到受害者在该Site的授权信息了。
- 点击劫持-Click-Jacking：攻击者用透明框架-Iframe包含一个预加害的(或者说是可被利用的)Page(比如Adobe Flash在线设置Page)，覆盖在攻击者伪造的一个Page上的某个部分(该部分很可能诱使用户去点击)。攻击者很可能诱使用户不知不觉的某个Sensitive-Site上的某处(比如，不知不觉的删除已经登录的帐户，打开摄像头等)。不过，因为用户只能在在Sensitive浏览器中登录攻击者想要加害/利用的Sites，所以在Non-Sensitive浏览器中的框架无效了。
- 跨站资源伪造-CSRF：很容易理解，攻击者在其设计的Page上发送一个子文档请求-Subdocument Request到一个Sensitive Site，浏览器会自动附带上已经登录的Sensitive Site的授权信息，比如Cookie，那么攻击者就可以篡改用户在该Site的状态了。同理，多浏览器/配置无法自动附加已登录的信息。
- 访问连接嗅探攻击-Visited Links Sniffing：攻击者的网站可能通过改变用户的访问过的连接的颜色或者样式来与未访问的链接区分开，然后再用JS或CSS来发现哪些是访问过的链接，来嗅探到用户的浏览历史。尽管一些预防措施已被一些主流浏览器推荐并采用了，新的攻击方式也被曝光可以绕开这些防御错误嗅探用户浏览历史。不过，在多浏览器/配置下用户的历史记录保存在不同的配置中。
- 缓存计时攻击-Cache-Timing Attack：与历史嗅探攻击相似，攻击者可以通过监测载入一个目标资源的时间从而确定用户访问了哪些链接。
- 渲染引擎劫持-Rendering-Engine Hijacking：牛B的攻击者可能会利用浏览器的漏洞劫持浏览器渲染引擎，单引擎浏览器(Firefox或Safari等)则意味着攻击者获得了访问所有用户数据的权限，比如用户的Sites Cookies或者Page内容。这些攻击也会被应用到多渲染进程的浏览器，如果这些浏览器需要通过渲染引擎来执行同源策略-Same-Origin Policy(Chrome或者IE等)。不过，当多浏览器/配置时，攻击者对一个被置于沙盒中的渲染引擎的劫持不会使另一个渲染引擎的Sites受威胁。

哪些特点使但浏览器/配置模式会暴露如上这么多的漏洞呢？总结起来就3点：

- 恶意Sites可以自由的向目标Sites发起请求
- 恶意Sites可以发起请求，这些请求允许攻击者访问/使用目标/受害/预加害的Sites的授权信息，比如Cookie或会话数据
- 恶意Sites可以利用渲染引擎直接访问内存状态信息和磁盘状态数据。

研究发现，这不是某个浏览器的问题，而是目前所有主流浏览器的共同缺点。很多类型的Web Sites可以通过逐个解决上述的攻击漏洞，来模拟多浏览器/配置的行为模式。但这些改变的代价是兼容性问题，因为一些合法的第三方Sites也被禁止访问用户的Cookies了。

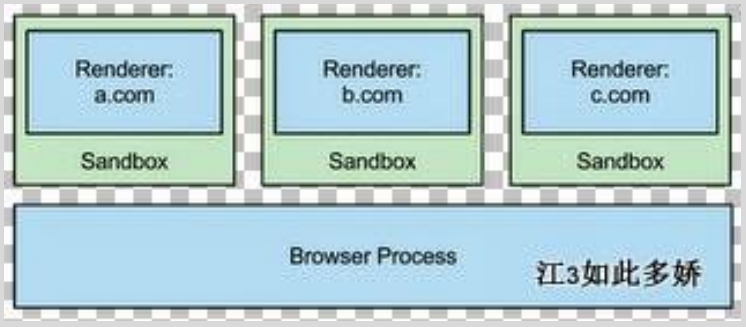
定义Isolated Sites/Apps：
Web App隔离虽然增强了安全性，但也付出了兼容性代价。隔离Cookie和内存状态信息，不光阻止了恶意网站篡改敏感Sites的数据，也阻止了合法Sites之间共享状态/授权/配置等信息。比如，Facebook链接允许Sites通过Facebook获取用户的身份信息，如果隔离App的话，这就不能实现了。为了实现Sites对这种信息分享的要求，我们使用一种可选择/自由参与策略-Opt-in Policy,来允许Sites自己决定是否将其Web App与其他Web App隔离。
我们必须十分谨慎的决定这种参与/选择性机制，以免带来新的安全隐患，所以我们还要考虑到隔离的单位/范畴。首先给出一种不完成的选择性参与机制-HTTP Header的后果，然后描述一种源范畴Orgin-Wide的可见方式-host-meta，最后再细化为子源范畴Sub-Origin的级的web应用的方法-mainfest文件。

通过**HTTP Header**初始化：
我们首先尝试通过自定义的HTTP Header(如X-App-Isolation:1)来定义Web App的范畴，如果浏览器收到的回应中包含这个头，那么浏览器就视后面收到的来自该源/Site的反应属于同一个隔离的Web应用。
该方法的第一个弊端是它不能确定收到的反应是否有权代表整个域。这种代表权限确认性的缺失，可以使域下的某个子部分(比如foo.com/~username/)的拥有者可以选择把整个域都加入同一个Web应用。

==========

Chrome的实现：

设计目标：
Chrome的多进程框架提供了很多速度、稳定性、安全性的好处，它允许位于不同Tab的Web Pages并行运行，它允许用户在其他Tab崩溃后继续浏览。因为渲染进程不需要直接访问网络、磁盘、设备，Chrome可以在沙盒中运行它们。从而可限制攻击者利用渲染引擎的漏洞的破坏范围，是其更难接触到文件系统、设备、特权页面(e.g. 设置页或者扩展管理页)或者其他配置中的Pages(e.g. 隐私模式)



不过，Chrome的沙盒还可以发挥更大的用处：Sites隔离。目前Chrome尽力将不同Web Sites的Pages置于多渲染进程中，但是出于兼容性考虑，仍然有很多来自不同Sites的Pages（如跨站框架-Cross-Site Iframes）共处同一个进程。这种情况下，Chrome只能依赖渲染引擎执行同源策略来隔离不同Sites了。

Chrome的Site Isolation机制就是把Chrome的渲染引擎变为Web App的安全边界，即使存在Webkit漏洞时。Chrome的目标是支持一个"site-per-process"策略，来确保渲染引擎中至多包含来自一个Site的Pages。浏览器进程可以基于进程中的Sites赋予其他进程对其Cookie和其他资源有限的访问权。

威胁模式：
对于"site-per-process"策略，我们假设攻击者可以诱使用户访问特定的链接，从而攻击者能够利用渲染进程的漏洞在沙盒中运行任意的代码，我们假设攻击者想要盗取信息或者滥用访问其他Web Sites的特权。

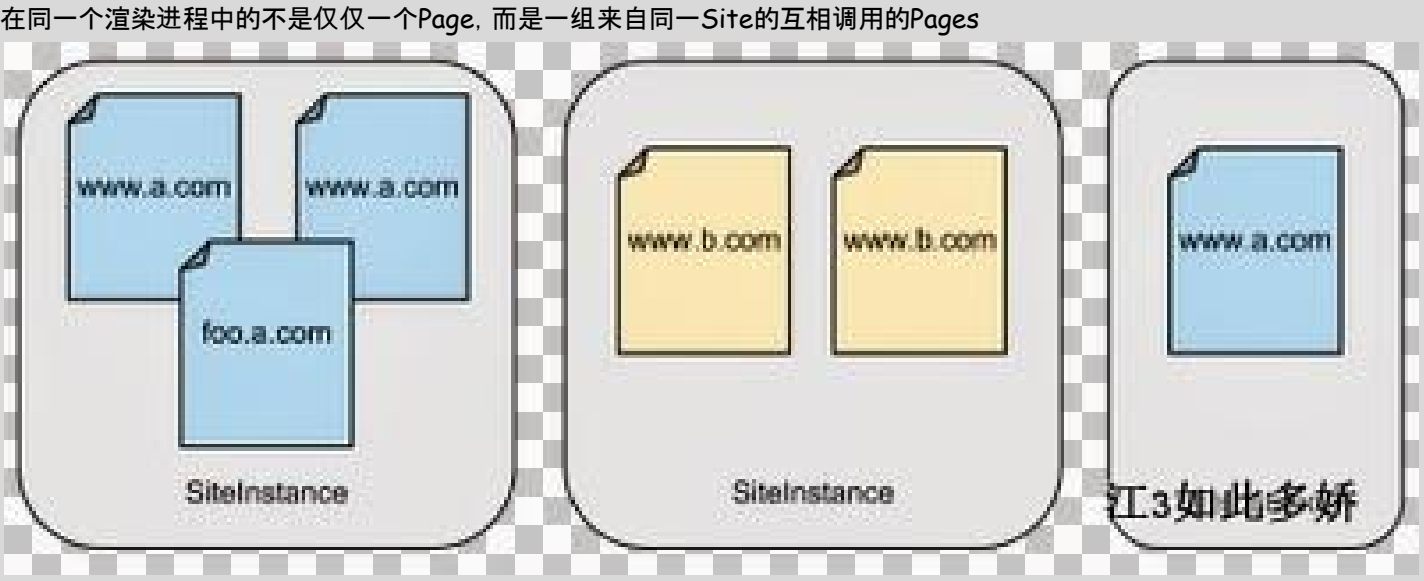
这里，我们使用一个精确度Site定义原则：一个Page的Site包含协议-sceme和主域名-registed domain name，包含公开后缀-public suffix（即顶级域名，com等），但忽略子域名（如www等）、端口、路径。我们用Sites而不是origins来避免破坏兼容性，从而允许现存的Pages可以修改document.domain 来跨子域名-subdomain通信。

我们认为该策略整体上存在如下威胁类型：

- 盗取跨站-Cross-Sites Cookie和HTML5存储型数据：
- 盗取跨站-Cross-Sites HTML, XML, and JSON文件：
- 取帐号密码：
- 滥用授权给其他Sites的许可：
- 绕开X-Frame-Opinion：

我们不指望通过该策略缓解传统的跨站Cross-Sites攻击以及仅在目标Page内就能完成的攻击，比如XSS、CSRF、XSSI、点击劫持，

要求：
在同一个渲染进程中的不是仅仅一个Page，而是一组来自同一Site的互相调用的Pages



Chrome实验性试用：
该机制默认关闭，想要开启通过命令行参数：
--enable-strict-site-isolation 严格Site隔离，强制进程所以适用于大部分Site情况
--site-per-process 仅供实验

跨站型/非同源-Cross-Sites/Cross-Origins，
多浏览器/配置与多帐号登录，多个会话，多个授权信息状态