Your Client-Side Security Sucks

STOP USING IT!
(as your only method for security)

Presented by: Kurt Grutzmacher
http://grutztopia.jingojango.net

Agenda

- What is Client-Side Security?
- Why it's sometimes good and sometimes bad
- Three Examples of Gooey Badness
- Solutions
- Q&A

Dis, claim her?

What is Client-Side Security?

Specifically, what do I mean by it?

Using client-side technology such as JavaScript, Java, Flash, etc to validate data before being transmitted to the server.

"Hiding" data and performing functions within the client that logically should be performed on the server instead.

Not the W3 Client-Side Security document by Lincoln Stein (http://www.w3.org/Security/Faq/wwwsf2.html) Still a good history on what we used to fear before the days of XSS - ActiveX, Java, IE 4.01, etc.

Not talking about DOM security, same-origin policy, sandboxes, etc.

OWASP Top 10 (2007)

1. **Cross Site Scripting (XSS)**
2. **Injection Flaws**
3. **Insecure Remote File Include**
4. **Insecure Direct Object Reference**
5. **Cross Site Request Forgery (CSRF)**
6. **Information Leakage and Improper Error Handling**
7. **Broken Authentication and Session Management**
8. **Insecure Cryptographic Storage**
9. **Insecure Communications**
10. **Failure to Restrict URL Access**

http://www.owasp.org/index.php/Top_10

Is It All Bad?

- No, not really. Client-Side validation can enhance the user's experience by not allowing good people to make data entry mistakes. For example:

- BUT….You should not depend upon it for *SECURITY*
- Users can always submit requests from outside of the client, modify in-line, use a proxy, etc.

Is It All Bad?

- No, not really. Client-Side validation can enhance the user's experience by not allowing good people to make data entry mistakes. For example:

```
function validateEmpty(fld) {
    var error = "";

    if (fld.value.length == 0) {
        fld.style.background = 'Yellow';
        error = "The required field has not been filled in.\n"
    } else {
        fld.style.background = 'White';
    }
    return error;
}
```

- BUT….You should not depend upon it for *SECURITY*
- Users can always submit requests from outside of the client, modify in-line, use a proxy, etc.

The Examples of Bad

Presented by: Kurt Grutzmacher
http://grutztopia.jingojango.net

The Examples of Bad

Example 1: Protecting a page location with JavaScript (or the page itself)

The Examples of Bad

Example 1: Protecting a page location with JavaScript (or the page itself)

Example 2: Input Validation (SQL, XSS, etc)

The Examples of Bad

Example 1: Protecting a page location with JavaScript (or the page itself)

Example 2: Input Validation (SQL, XSS, etc)

Example 3: Business Logic flaw

Bad Example #1

Bad Example #1

Encrypted Password Script (from 2001 but still found in use)
http://www.dynamicdrive.com/dynamicindex9/password.htm

"JavaScript password scripts have improved substantially over time, with the latest enhancement being an encrypted password, archived using "fuzzy" algorithms. The result are password scripts that won't crumble as soon as the user views the page's source. Use this script to password protect your webpage; based on a simple yet effective encryption method, it's one of the best of its kind."

Thanks to Garrett Gee for showing me this!

Your Client-Side Security Sucks

“best of its kind”

```
function calculate(){

        passworda = document.password1.user1.value.toLowerCase()
        passwordb = document.password1.pass1.value.toLowerCase()

        var user = 1
        var pass = 1

        for(d=0;d<passwordb.length;d++) {
                pass*= passwordb.charCodeAt(d);
        }
        for(e=0;e< passworda.length; e++) {
                user *= passworda.charCodeAt(e);
        }
        document.password1.outputuser1.value = user;
        document.password1.outputpass1.value = pass;
}
```

```
function submitentry() {
        password = document.password1.password2.value.toLowerCase()
        username = document.password1.username2.value.toLowerCase()
        passcode = 1
        usercode = 1
        for(i = 0; i < password.length; i++) {
                passcode *= password.charCodeAt(i);
        }
        for(x = 0; x < username.length; x++) {
                usercode *= username.charCodeAt(x);
        }
        if(usercode==134603040&&passcode==126906300)
        {
                window.location=password+".htm"
        } else {
                alert("password/username combination wrong")
        }
}
</script>

<form name="password1">
<strong>Enter username: </strong>
<input type="text" name="username2" size="15"><br>
<strong>Enter password: </strong>
<input type="password" name="password2" size="15">
<input type="button" value="Submit" onClick="submitentry()">
</form>
```

What this script does

1. User enters USER and PASSWORD and hits "Submit" which calls the JavaScript function "submitentry"

2. This function assigns two variables the number 1 then runs loop the length of each entry string to:

    a. Obtain the ASCII character code of the letter, and

    b. Multiply the code number to the base variable

3. After completion it compares the results to the precomputed versions and redirects to the "secured" page if both are correct. The "secured" page is the cleartext password.

Your Client-Side Security Sucks

How to break it

Presented by: Kurt Grutzmacher
http://grutztopia.jingojango.net

Bay Area OWASP - Feb 21, 2008

How to break it

1. Brute force the "secured" file/password with curl, wget, your browser, etc.

How to break it

1. Brute force the "secured" file/password with curl, wget, your browser, etc.
2. Google the site looking for the "secret page"

How to break it

1. Brute force the "secured" file/password with curl, wget, your browser, etc.
2. Google the site looking for the "secret page"
3. Write a program/script to run a dictionary or incremental attack against the encryption

How to break it

1. Brute force the "secured" file/password with curl, wget, your browser, etc.
2. Google the site looking for the "secret page"
3. Write a program/script to run a dictionary or incremental attack against the encryption
4. Have a friendly math/crypto geek reverse the encryption routine (fun at parties!)

Lessons Learned

Giving the client everything thing they need to break your security is not any good “kind” to be best of.

Don’t build a castle with an entry gate and then forget to fill the moat.

Bad Example #2

Me: "Hello, I was using your system and I appear to have discovered an SQL Injection flaw with your site. Here are the details . . ."

Them: "Thank you for your assistance."

Answer: **WE FIX IN JAVASCRIPT!**

This really happened and was still an issue as of 2/20/08. Vendor will not be named but has been contacted AGAIN

Your Client-Side Security Sucks

Lets say we have a web page with some input fields

Your Client-Side Security Sucks

Lets say we have a web page with some input fields

Credit Card History

Select the period:

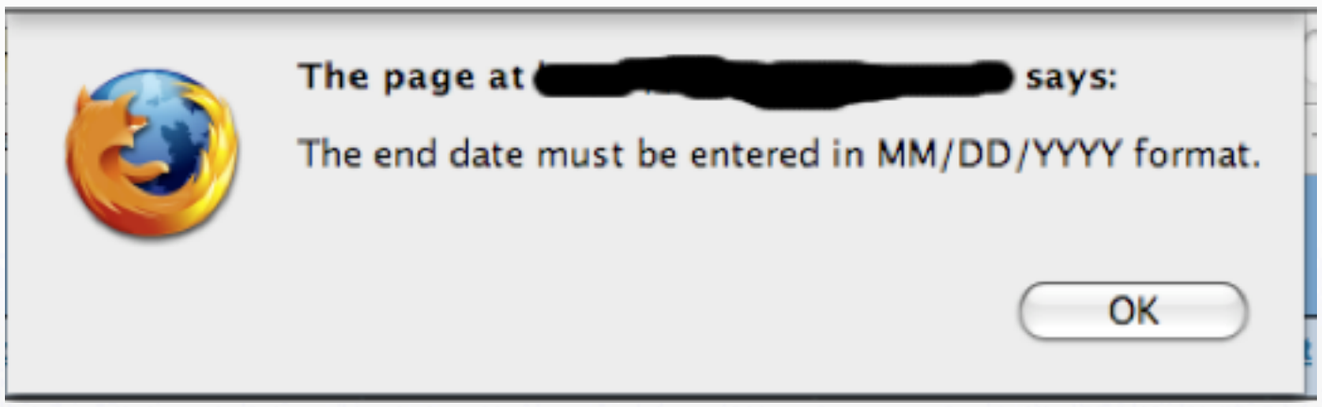
Display Payment history for the last 1 Day(s)

Display Payment history for the period from 1/20/2008 (mm/dd/yyyy) to 2/20/2008' (mm/dd/yyyy)

Run Report

Your Client-Side Security Sucks

Lets say we have a web page with some input fields



Your Client-Side Security Sucks

Lets say we have a web page with some input fields

```
<input value="1" name="Period" checked="checked" type="radio">
Display Payment history for the last
<input size="4" value="1" name="NumberOfDays">
Day(s)<br>
<input value="2" name="Period" type="radio">
Display Payment history for the period from
<input size="10" value="1/20/2008" name="StartDate">
(mm/dd/yyyy) to 
<input size="10" value="2/20/2008" name="EndDate">
 (mm/dd/yyyy)</font></p>
<p><font face="Arial, Helvetica, sans-serif" size="2">
<input value="Run Report" name="Submit" onclick="validate(document.form1)" type="button">
</font></p>
```

Your Client-Side Security Sucks

```
function validate(theForm) {

  if (!ValidateInt(theForm.NumberOfDays.value))  {
    alert("The number of days must be an integer value.");
    theForm.NumberOfDays.focus();
  }

  else if (!ValidateDate(theForm.StartDate.value))  {
    alert("The start date must be entered in MM/DD/YYYY format.");
    theForm.StartDate.focus();
  }

  else if (!ValidateDate(theForm.EndDate.value))  {
    alert("The end date must be entered in MM/DD/YYYY format.");
    theForm.EndDate.focus();
  }

  else {
    theForm.submit();
  }
```

Your Client-Side Security Sucks

```
function ValidateDate(z) {

  var x = new Boolean(true);
  if (z != "") {
    var DatePattern = /^(\d{1,2})(\/|-)(\d{1,2})\2(\d{4})$/; // MM/DD/YYYY // Date Pattern
    var TempString = z.match(DatePattern);

    if (TempString == null) {
      x = false;
    }
    else {
      var dayLengths = [31,29,31,30,31,30,31,31,30,31,30,31];
      var m = TempString[1], d = TempString[3], y = TempString[4];

      if(!((y % 4 == 0 && y % 100 != 0) || y % 400 == 0)) {
      dayLengths [1] = 28;
      }

      if (m <= 0 || m > 12 || d <= 0 || d > 31 || y <= 0 || dayLengths[m-1] < d) {
        x = false;
      }
    }
  }
  return x;
}
```

What they've done

- The vendor knows they're expecting a date in a specific format. The JavaScript function "validate" checks to ensure the fields are in that format.

- If they're not valid, send a dialog box telling the user to enter a correct date.

- This is all well and good if you forget that browsers are not the only place users can enter data.

Your Client-Side Security Sucks

Using a proxy (like WebScarab, TamperData, etc) the attacker can bypass any client-side validation steps:

Your Client-Side Security Sucks

Using a proxy (like WebScarab, TamperData, etc) the attacker can bypass any client-side validation steps:

Request Header Value
www.*********.com
Mozilla/5.0 (Macintosh; U; Intel
HTTP Accept=text/xml,applicat
en-us,en;q=0.5
gzip,deflate

Post Parameter Name	Post Parameter Value
NumberOfDays	1
Period	2
StartDate	1%2F20%2F2008
EndDate	2%2F20%2F2008'

Your Client-Side Security Sucks

Using a proxy (like WebScarab, TamperData, etc) the attacker can bypass any client-side validation steps:

Microsoft VBScript runtime error '800a000d'

Type mismatch: '[string: "2/20/2008""]'

/youraccount/ConnectHistoryReport.asp, line 268

Your Client-Side Security Sucks

Using a proxy (like WebScarab, TamperData, etc) the attacker can bypass any client-side validation steps:

ame	Request Header Value
	www.*********.com
	Mozilla/5.0 (Macintosh; U; Intel
	HTTP Accept=text/xml,applicat
	en-us,en;q=0.5
	gzip,deflate
	ISO-8859-1,utf-8;q=0.7,*;q=0

Post Parameter Name	Post Parameter Value
NumberOfDays	1
Period	2
StartDate	1%2F20%2F2008
EndDate	2%2F20%2F2008'%20UNION%2(

Your Client-Side Security Sucks

Using a proxy (like WebScarab, TamperData, etc) the attacker can bypass any client-side validation steps:

Invoice #	Date/Time	Package Purchased	Price	Taxes	Total	Card Type	Card Number
0000000042	1/10/1900	[redacted]	$9.00	$9.00	$9.00	mi[redacted]	*9
0000000070	1/10/1900	v[redacted]	$9.00	$9.00	$9.00	vl[redacted]	*9
0000000073	1/10/1900	J[redacted]	$9.00	$9.00	$9.00	z[redacted]	*9
0000000113	1/10/1900	j[redacted]	$9.00	$9.00	$9.00	ja[redacted]	*9
0000000150	1/10/1900	t[redacted]	$9.00	$9.00	$9.00	t[redacted]	*9
0000000198	1/10/1900	y[redacted]	$9.00	$9.00	$9.00	[redacted]	*9
0000000227	1/10/1900	fletcher	$0.00	$0.00	$0.00	hilton	*0

Your Client-Side Security Sucks

Using a proxy (like WebScarab, TamperData, etc) the attacker can bypass any client-side validation steps:

Invoice #	Date/Time	Package Purchased	Price	Taxes	Total	Card Type	Card Number
0000000042	1/10/1900	[redacted]	$9.00	$9.00	$9.00	mi[redacted]	*9
0000000070	1/10/1900						*9
0000000073	1/10/1900						*9
0000000113	1/10/1900						*9
0000000150	1/10/1900						*9
0000000198	1/10/1900						*9
0000000227	1/10/1900						*0

Great Job!

Lessons Learned

Javascript, Java, Flash, etc are not good enough to stop Injection, XSS, or other attacks

Validate the data on the server!

When you have an SQL Injection and are told about it, fix it in the *server code*, don't just mask it.

Bad Example #5

Macworld Conference & Expo®

+

```
G:\passwd>john

John the Ripper  Version 1.6  Copyright (c) 1996-98 by Solar Designer

Usage: //G/PASSWD/john [OPTIONS] [PASSWORD-FILES]
-single                  "single crack" mode
-wordfile:FILE -stdin    wordlist mode, read words from FILE or stdin
-rules                   enable rules for wordlist mode
-incremental[:MODE]      incremental mode [using section MODE]
-external:MODE           external mode or word filter
-stdout[:LENGTH]         no cracking, just write words to stdout
-restore[:FILE]          restore an interrupted session [from FILE]
-session:FILE            set session file name to FILE
-status[:FILE]           print status of a session [from FILE]
-makechars:FILE          make a charset, FILE will be overwritten
-show                    show cracked passwords
-test                    perform a benchmark
-users:[-]LOGIN|UID[,..] load this (these) user(s) only
-groups:[-]GID[,..]      load users of this (these) group(s) only
-shells:[-]SHELL[,..]    load users with this (these) shell(s) only
-salts:[-]COUNT          load salts with at least COUNT passwords only
-format:NAME             force ciphertext format NAME (DES/BSDI/MD5/BF/AFS/LM)
-savemem:LEVEL           enable memory saving, at LEVEL 1..3

G:\passwd>
```

=

Your Client-Side Security Sucks

I've got a blue ticket

Presented by: Kurt Grutzmacher
http://grutztopia.jingojango.net

Bay Area OWASP - Feb 21, 2008

I've got a blue ticket

The flaw: Storing critical data on the client side for validation purposes

I've got a blue ticket

The flaw: Storing critical data on the client side for validation purposes

The tools: Browser, text editor, password cracker, a little research, and time

I've got a blue ticket

The flaw: Storing critical data on the client side for validation purposes

The tools: Browser, text editor, password cracker, a little research, and time

The result: A free pass to MacWorld!

Where to start?

From the beginning of course!

In early 2006 MacWorld was just around the corner. Expo passes are usually free but I waited too long and didn't have a vendor code. Since I like free vendor schwag I became determined.

In the source code I noticed a JavaScript MD5 routine and what appeared to be many hashes. I ran them through John The Ripper and immediately was given the code "CREDIT" which provided a no-cost Platinum Pass. Whoa.

I didn't believe it would work until I picked up my badge during lunch time the first day of the conference. That afternoon I immediately contacted IDG and the codes were removed. I met with them the next day to say hi and that I wasn't a bad person, I'm here to help, etc.

Problem solved. They wouldn't do it again, would they?

...those who forget the past

Of course they would! I wouldn't be here ranting about client-side security if that were the case.

This year I started a little earlier after a few people asked me to take a look. Lo and behold -- *THEY WERE USING THE SAME SOURCE CODE!*

B U T … Running the MD5 hashes in JTR had yet to reveal any of the "good" codes.

Password Cracking 101

Keyspace	Set of all possible keys that can be used to initialize a crypto algorithm
Key length	Size of a key used in a cryptographic algorithm
Brute force attack	Method of defeating a cryptographic scheme by trying a large number of possibilities. Sometimes known as "Incremental"
Dictionary Attack	Method of defeating a cryptographic scheme by using a list of words
Rainbow Tables	A pre-computed lookup table of a keyspace and key length offering a time-memory tradeoff for recovering plaintext

Step 1

Get the MD5 hashes from the source code and format for processing (John The Ripper and RainbowCrack both can use "userid:password" format:

```
553:77c34ddea4adf4aa79c69ab471539847
554:2476fee59de2c14f3bcc305f84c32209
555:1d2863778fb0fe89c9e4c2929e437c14
556:90fd53a2967995804bfb3ab639c9f6d0
557:d6fdf20e7995d08c2ce75fe2dd943af0
558:c47cbb4b92b68d4b9fe85fc0ea4e0042
559:d31830730fd84233bdd1bfe1969cb24e
560:eac8780bdd7c8d39bda71bb854425b21
561:ac910361ffec9261802b907788d446a4
562:852c6738e01803f64ac785abe3ae6659
563:6e5d4f697d7aa4901460cd0257484176
564:fcc66c568b7fd1f7cdde953628238ee1
565:cf0c737b854ce6e97654542f200e0f42
566:df2fe494621ae661d93e52190086c794
567:3c65bb39ee7b2e8106e9cc375fac804a
568:b61818555bc3740a368aa32b5c35a5e6
```

Step 2a - Crackin'

Run John The Ripper!

Run RainbowCrack!

Wait some amount time. . .

Nothing useful cracking? Drat, time to get smarter.

Maths Break

The size of your keyspace (k) and the maximum word length (l) determine the total number of permutations that have to be encrypted to check every instance (P). P=k^l.

Take the benchmark cracks-per-second your machines do (Cs), run the math (P/Cs) and you have the number of seconds it takes to run an Incremental.

This is a **total** time required to exhaust the keyspace and length. Randomness and chaos play a big part to achieve a successful crack.

k = 69 l = 8 Cs = 30M	$\frac{69^8 / 30M}{60}$	285,443.54 minutes (3.68 months)
k = 69 l = 7 Cs = 30M	$\frac{69^7 / 30M}{60}$	4,136.86 minutes (69 hours)
k = 69 l = 6 Cs = 30M	$\frac{69^6 / 30M}{60}$	**59.95 minutes**

Step 2b: Lernin'

Brute force password cracking is tedious. We have no idea the size of the keyspace or the length we're looking for. If only we could crack a billion MD5 hashes per second. . . .

We can crack smarter however. We know a general format of the codes that are given out for free Expo passes by searching Google:

http://www.google.com/search?hl=en&q=macworld+priority+code+2008

08-E-VF01 08-G-PC260 08-G-PC189 . . . . Pattern?

Step 2c: Filterin'

A word filter helps reduce the keyspace and length to a manageable size. Since we have a general knowledge of the plaintext (*08-x-y*(z)*), restricting the incremental mode to that format will lessen the amount of time we need to process.

```
[Incremental:MW]
File = $JOHN/lanman.chr
MinLen = 7
MaxLen = 7
CharCount = 69

[List.External:MW]
void filter()
{
    int i, c;
    i = 0;
    while (c = word[i]) {
        // If character is lower case, convert to upper
        if (c >= 'a' && c <= 'z') word[i] &= 0xDF;
        i++;
    }

    // We know the static filter 08-?-????
    // Add or remove word[]s to fit the length
    word[10] = word[6];
    word[9] = word[5];
    word[8] = word[4];
    word[7] = word[3];
    word[6] = word[2];
    word[5] = word[1];
    word[4] = '-';
    word[3] = word[0];
    word[2] = '-';
    word[1] = '8';
    word[0] = '0';
}
```

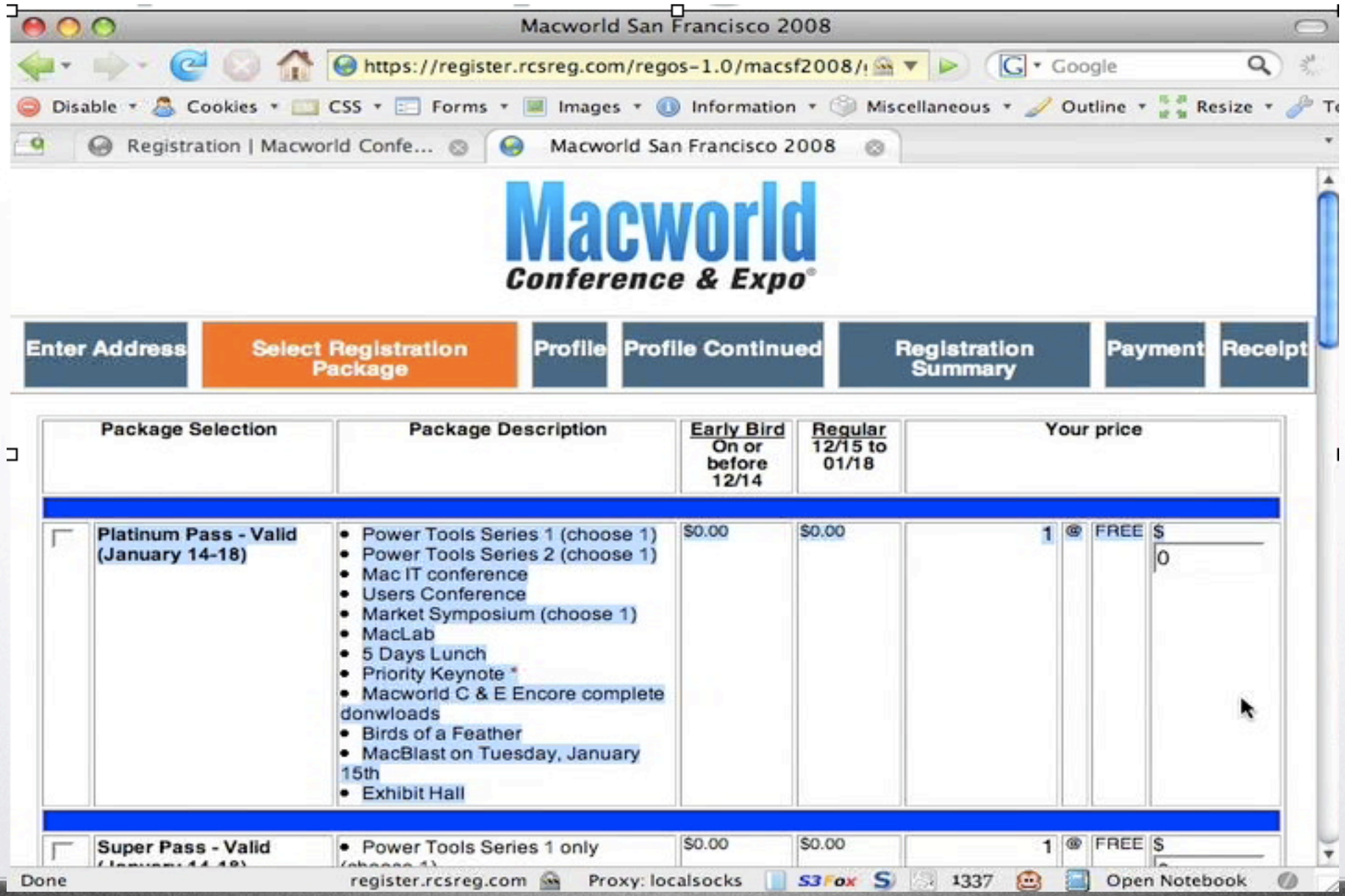
Your Client-Side Security Sucks

Step 3: Smart Crackin'

root@:/ | Default | grutz@jumpbox:~

```
1949:4deaf98d2c54f0c15811c14602fdfd8a
1950:9492d0a7f9a144795063c18a286ea26b
1951:1045be4dcd12b63b6b17c531bcaf1fbd
1952:4e10ab8f7d5b02710d5eff925729ba4d
1957:932b176b6d9a884f56b8facb392a9fec
1958:d0a7c20f5286ab77c6f6223aff11d257
1959:f868ee1e43f0dececd6358b56514fa03
1960:16883b74fa21a74606c13ae2d16ade75
1961:bae924cbf831f492c69a312c186e972f
1962:bc4519493c6a35bea5378de112a90303
1963:9d8526ebb4c080b58fa0aba3443de0e4
1964:cb08b3bf0dea89aceb244196f4a36176
1965:091bec09a449e6094ce3fdf41d5bfe91
1966:dd0404e61ec6a5b0124cc93af76dd691
1967:822785def257c66f63ca6e2ce4a4249b
1968:5e5b1feefa1f14f35cc46df36408c312
1969:0451a3336984703bfd4297df3115b7f3
1970:d80fb8735f58b83e9f6a3aa088ede981
node0 run # mpirun -np 9 ./john -i=MW --session=MW2K8 --format=raw-MD5 --external=MW macworld-2008.codes
Loaded 1341 password hashes with no different salts (Raw MD5 [raw-md5 SSE2])
Loaded 1341 password hashes with no different salts (Raw MD5 [raw-md5 SSE2])
Loaded 1341 password hashes with no different salts (Raw MD5 [raw-md5 SSE2])
Loaded 1341 password hashes with no different salts (Raw MD5 [raw-md5 SSE2])
Loaded 1341 password hashes with no different salts (Raw MD5 [raw-md5 SSE2])
Loaded 1341 password hashes with no different salts (Raw MD5 [raw-md5 SSE2])
Loaded 1341 password hashes with no different salts (Raw MD5 [raw-md5 SSE2])
Loaded 1341 password hashes with no different salts (Raw MD5 [raw-md5 SSE2])
Loaded 1341 password hashes with no different salts (Raw MD5 [raw-md5 SSE2])
08-S-STAFF       (1188)
```

Step 4: Trying The Code



Your Client-Side Security Sucks

The full video and additional details can be viewed here:

http://grutztopia.jingojango.net/2008/01/another-free-macworld-platinum-pass-yes.html

Lessons Learned

This could have been mitigated if all of the codes listed didn't provide any large discounts.

Just because you have a longer password doesn't mean it's not going to be broken.

Business logic flaws can be discovered if you think about the process flow.

Session States

C#/.NET:VIEWSTATE - Can be modified and accepted by the app if not MAC signed or encrypted:

```
<%@Page EnableViewStateMAC=true %>
<machineKey validation="3DES" />
```

http://weblogs.asp.net/varad/archive/2005/02/04/367056.aspx

Ruby on Rails 2.0: CookieStore session object will be encrypted and stored in the browser by default. Add this to your environment.rb script:

```
config.action_controller.session = {
    :session_key => '_cookies2_session',
    :secret      => 'secret-secret',
}
```

http://www.rorsecurity.info/2007/11/20/rails-20-cookies/

Solutions!

- Do Not Trust The Client, EVER
- Use client-side validation to improve the customer experience
- Verify all data on the SERVER before processing
- Beware of business logic flaws! They can't be caught by scanning tools (as far as I am aware) and can usually be found by reviewing your workflow and asking yourself "Are we doing this in a secure way? What are my risks?"

THANK YOU!