










- 1  **HTTP / HTTPS / URL'S**
The Hacking Protocols
and
The Hackers Sword
- 2  *"The light-saber is a Jedi's weapon – not as clumsy or random as a blaster."*

Obi-Wan Kenobi, "Star Wars: Episode IV"
- 3  **Introduction**
 - ⊙ Why are firewalls basically meaningless for Web security?
 - People might not realize: if you have a Web site on the www, you have 1 (or 2) hole(s) in your firewall – ports 80 (and 443)!
 - Why? HTTP communicates over TCP port 80 and HTTPS communicates over TCP port 443
 - A firewall must let traffic through ports 80 and 443!
- 4  **Protocols of the Web**
 - ⊙ HTTP = Hypertext Transfer Protocol
 - "Language" spoken over the Web
 - Textual based
 - Request/Response mechanism
 - How can we make it secure?!?
 - HTTP over TCP Port 80
 - ⊙ HTTPS = HTTP over SSL
 - Same as HTTP but text is encrypted
 - HTTPS over TCP Port 443
- 5  **Protocols of the Web**
 - ⊙ No matter what, all Web browsers/servers must use HTTP
 - ⊙ Request/Response stateless protocol
 - Client sends an HTTP request to communicate to the server its intentions
 - Server returns an HTTP response to communicate its desires
- 6  **HTTP/0.9 & 1.0**
 - ⊙ HTTP/0.9
 - First official version of HTTP
 - Somewhat limited
 - ⊙ HTTP/1.0
 - Even though 1.0 is "old" (released in 1996), it is still used
 - Most are using to 1.1...
 - Again, request made by client, response made by server
 - ⊙
- 7  **HTTP/1.0**
 - ⊙ The URL (Universal Resource Locator) initiates the request from the client
 - ⊙ The request contains (among other things) a *Request method*
 - ⊙ The method is like a command to the server..
 - GET
 - HEAD
 - POST
- 8  **The HTTP/1.0 Request**
- 9  **HTTP/1.0 Response**
 - ⊙ To respond, the server sends back a response message

- ⊙ The message contains:
 - Response code: a numeric code that has some meaning
 - Header fields: additional info about the response
 - Data: the content or body of the response. If requesting HTML docs, this is the actual HTML page

10 **The HTTP/1.0 Response**

11 **HTTP/1.1**

- ⊙ Official spec as of 2001
- ⊙ Widely used by popular browsers
- ⊙ HTTP/1.1
 - Extends HTTP/1.0 by adding some interesting new features
 - Examples:
 - New methods
 - New response codes
 - New header fields

12 **Query String Support**

- ⊙ The HTTP/1.1 URL supports a query string & script parameter passing
 - The query string is at the heart of any Web application
 - The query string is the primary way attackers get access to a Web server
 - Anything after the "?" can be processed by a server script (PHP, ASP, etc...)
 - Ex: `.../myscript.php?id=112233&name=myname`

13 **An HTTP Req/Resp Example...**

- ⊙ The message that the browser sends to the server might be:

14

15 **HTTPS – Secure HTTP**

- ⊙ HTTPS is a protocol used to encrypt an HTTP stream
- ⊙ In essence, it is HTTP over SSL (Secure Socket Layer)
- ⊙ HTTPS is used when sensitive info is being sent via HTTP (typically through an HTML form)
 - Credit card transactions
 - Transactions involving your SSN
- ⊙ This is not foolproof!!!

16 **The Universal Resource Locator**

- ⊙ Most Web attacks today are quite elegant
- ⊙ The attack usually starts with small steps and each proceeding step takes them deeper
- ⊙ The elegance is in the fact that only a browser is used or needed
- ⊙ The carrier of the attack payload?
- ⊙ The URL!

17 **URL Structure**

- ⊙ A URL (Universal Resource Locator) is a mechanism for uniquely identifying a Web resource.
- ⊙ Generic form:
 - `protocol://server/path/resource?parameters`
- ⊙ Protocol: http, https, ftp, etc.
- ⊙ Server: www.uwp.edu

- ⊙ Path: /staff/knautz
- ⊙ Resource: res.php?id=12345&name=tim

18 **URL Structure**

- ⊙ Some URL Examples:

19 **URL's & Parameter Passing**

- ⊙ The last example in the previous slide shows parameters being passed to a server-side resource
- ⊙ The *query string* is used to pass info from the browser to a Web server
- ⊙ When a form is submitted, the form elements are gathered up and sent to the server as a query string
- ⊙ Example
 - http://www.uwp.edu/query.php?
 - id=123&name=tim
 - id=123&name=tim is the query string

⊙

20 **URL Encoding**

- ⊙ URL's are made up of letters, numbers and symbols
- ⊙ Some symbols have a special meaning within the confines of a URL and some do not
- ⊙ However, some symbols have a meaning for the Web server receiving the URL...

21 **URL Encoding**

- ⊙ ? = Query string separator
- ⊙ & = parameter delimiter. Separates name=value pairs
- ⊙ = = Name/value separator
- ⊙ + = Translates into a space
- ⊙ : = protocol separator
- ⊙ % = escape character for specifying hex, i.e. %20 is a space

22 **URL Special Characters**

- ⊙ What if you really need to put a "&" in the URL?
- ⊙ I.e., book=pride&prejudice&payment=visa
- ⊙ Most Web servers will break the QS into 3 parameters: book=pride, prejudice=, payment=visa
- ⊙ The URL spec allows us to pass special chars by using two-digit hex encoded ASCII prefixed with a % symbol: ...?book=pride%26prejudice&pmt=visa







23 **Meta-Characters**

- ⊙ Some characters do not have a special meaning in a URL
- ⊙ But, if these characters make it into an application, they may have meaning there
- ⊙ For example:
 - * is a "wildcard" character in shell scripts
 - | (pipe) is the pipe character in shell scripts.
 - This is especially lethal in Perl scripts: can cause commands to be executed on the server

24 **Meta-Characters**

- ⊙ For example (cont):
 - ` (back-tick) means "execute" in shell scripts or command output substitution. The output of the command inside the ` is executed and the output is placed in a variable

```
files=`ls -al`
```

- files contains the output of the ls command
- 25  **Unicode Encoding**
- ⊙ ASCII works pretty well...
 - English needs only a single byte to store characters
 - Not robust enough for other languages like Chinese
 - ⊙ Most OS's support multi-byte character sets: Unicode
 - 2 byte character set
 - Encoded %uXXXX where XX is the high-order byte & YY is the low-order byte
 - ASCII %00 - %FF = Unicode %0000 - %00FF
- 26  **Abusing URL Encoding**
- ⊙ When Unicode is used incorrectly, certain Web servers can be fooled exposing vulnerabilities
 - ⊙ Unicode Vulnerabilities
 - In 2000, Microsoft IIS vulnerable to the "Unicode Bug":
 - Illegal Unicode encoding of "/"
 - Allowed users to make URLs that could navigate outside of the document root
 - Allowed users to call the command shell (cmd.exe)
- 27  **Abusing URL Encoding**
- ⊙ Unicode Vulnerabilities (cont)
 - "Unicode Bug" (cont)
 - Example:
 http://192.168.7.21/scripts/..%c0%af.
 /winnt/system32/cmd.exe?/c+dir+d:\
 - How does it work?
 - Hacker knows the system is IIS and that the scripts are in a scripts directory at c:\inetpub
 - %c0%af translates into "/" making the directory equivalent to ../..
 - The whole thing now executes cmd.exe
- 28  **Abusing URL Encoding**
- ⊙ Unicode Vulnerabilities (cont)
 - "Unicode Bug" (cont)
 - How does %c0%af translate into "/"?
 - Unicode (specifically UTF-8) encoding allows for multi-byte character encoding up to 3 bytes
 -
- 29  **Abusing URL Encoding**
- ⊙ Unicode Vulnerabilities (cont)
 - "Unicode Bug" (cont)
 - But, according to the spec., a UTF-8 decoder is not supposed to accept encoding of a character longer than necessary
 - The "/" is encodable in 1 byte so the 2 or 3 byte encoding should have been rejected by IIS!!!
 -
- 30  **Abusing URL Encoding**
- ⊙ Unicode Vulnerabilities (cont)
 - "Double Decode" or "Superfluous Decode"
 - May of 2001 another bug was discovered in IIS
 - Very similar to the "Unicode bug"; same result
 http://192.168.7.21/scripts/..%25%32%66..

- /winnt/system32/cmd.exe?/c+dir+d:\
 - %25 = "%", %32 = "2", %66 = "f"
 - But we still have to decode again as %2f is not encoded
 - The result: ../../

◎ Summary: URL encoding can be a huge security risk!

31 HTML Forms

- ◎ We've all seen forms before...
- ◎ There are 2 aspects to handling forms:
 - Client-side processing
 - Server-side processing
- ◎ Browsers tasks:
 - Display the form
 - Allow input to the fields
 - Build a properly encoded URL
 - Send the URL to the server

32 HTML Forms

- ◎ Servers tasks:
 - Separate the query string
 - Process the input elements
- ◎ How do we know that the input elements are "proper"?
- ◎ What is secure form handling?

33 HTML Forms

- ◎ Anatomy of an HTML Form:
 - *Method*: The HTTP method used in the form submission. Usually GET or POST
 - *Action*: The server-side application that is run to process the forms input elements
 - *Input elements*: The form data. Every element must have a name
 - *Submit button*: Special button used to send the data
- ◎ Let's look at an example...

34 HTML Forms

35 Parameter Passing Via GET & POST

- ◎ See http://www.cs.uwp.edu/staff/knautz/form_elements.html
 - 2 forms, each with the same controls
 - Method is different for each
 - Notes:
 - The URL is different for each
 - POST → no form data passed in the URL
 - GET → form data passed in the URL including the password in plain text!
 - The HTTP request is different
 - POST → parameters passed as the content of the HTTP request
 - GET → parameters passed in the URL

36 Summary

- ◎ Web traffic is primarily HTTP or HTTPS
- ◎ HTTP and HTTPS are subjected to nearly 100% of all Web attacks
- ◎ Firewalls cannot stop these attacks!
- ◎ The URL is a tiny portal into your Web servers inner workings
- ◎ Firewalls, intrusion detection systems and proxy security technologies are useless
- ◎ Your ports 80 and 443 are open
- ◎ The URL is the sword of a Web attacker

© Understand the URL!