

Analyzing Web Traffic: Looking for the known and the unknown

Cory Scott, Lead Security Consultant – Securify Labs¹

Introduction

To have a complete and robust intrusion detection environment, analysts must include web service inspection and analysis in a production deployment. Since untrusted users often interact with web applications, failure to properly monitor the use and abuse of these applications can be detrimental. Unfortunately, there has not been significant emphasis on this field, likely because the complexity of web services makes it difficult to architect an “out-of-the-box” solution. The following paper aims to help arm an intrusion detection analyst with the knowledge necessary to perform a detailed and in-depth analysis of web services.

What are the threats to web services?

Since web services comprise a majority of the publicly available services on the Internet, a wide range of attacks have been devised against them. The threats impact multiple levels of the service and can threaten the availability, integrity, and confidentiality of the assets contained within.

Network – Simply allowing an individual to connect to a machine on the Internet presents certain risks. Denial of service attacks on the host comprise the majority of the risk at the network level. Additionally, denial of service attacks can happen at the transport, server, and application level by exhausting resources or exploiting programming errors such as unchecked input. Finally, IP spoofing can defeat access control schemes or trust models that the web service relies upon.

Transport- Attacks on the transport layer include an attacker using port 80 or 443 for traffic other than http or https. If application layer protocol inspection does not take place on firewalls that allow inbound TCP traffic on port 80 or 443 to certain hosts, then an attacker who wants to use another protocol on those ports may do so. SSL is often implemented insecurely and many clients do not handle SSL failures well. These issues may lead to security failures in the face of active attacks, especially man-in-the-middle attacks. Inspection of web traffic can be difficult, as the server may interpret a HTTP packet in one context, while the inspecting server may interpret it in another.

Protocol – The HTTP protocol itself is problematic. It has no support for data integrity, audit, or encryption. It relies on other layers (such as the Transport layer for encryption or the Server level for audit). HTTP does support authentication, but the methods used for supplying credentials is subject to eavesdropping and trivial encoding of passwords.

Server – Since the web server exports certain components of the filesystem for read and write access, improperly configured web servers can expose unintended files. In addition, since many web servers can execute code as part of a user request, improperly controlled web servers can execute unintended code as part of an attack. Since the server is a socket-

¹ The author would like to acknowledge and thank Scott Renfro, Director of Securify Labs, for his support in the development of this paper.

based application that receives uncontrolled requests, it can be vulnerable to buffer-overflows or other programming errors that result in arbitrary code execution. Often, a server is responsible for applying access control mechanisms to certain files and failures in access control mechanisms can expose files to unauthorized users. If the server fails to properly log access to the web server, it limits the ability to perform an audit. As the server runs under a certain user level on the operating system, compromise of the web server process could result in arbitrary actions taking place under that user's account. Privilege escalation could follow.

Application – One of the difficulties with web-based applications is that state must be maintained in interactive applications, while the underlying protocol is stateless. Cookies or hidden form fields with session IDs are often used to maintain state, leading to excessive reliance on the client to present persistence traits. Unchecked values in web applications (such as hidden form fields for price in a shopping cart application, for example) may lead to integrity failures. Improperly controlled administrative interfaces can allow an unauthorized user to change the server's configuration. Over-reliance on client-side limitations to user input (such as using JavaScript to enforce certain user behavior) can cause unexpected behavior when a malicious user purposefully disables these features. Reliance on weak user credentials or improper session management can result in a spoofed user identity. Often, a web application allows for parameters and instructions to be passed through to non-web-enabled applications. Improper checking of these parameters can result in unintended disclosure of information or arbitrary code execution.

Other – Vulnerabilities in network or operating systems that are not linked to the web service directly are a threat to a web service. For example, a vulnerable FTP service running on a web server could result in compromise of a server without using a web-oriented attack that could still impact the web service. Operational procedures that do not adequately protect the web services or a lack of appropriate physical security measures could also result in compromise.

Defense against threats to web-based applications

The three major defenses for securing web applications includes prevention, detection, and response. Many papers and articles have been devoted to this three-tier model, so only a quick summary will be presented here.

Prevention mechanisms include the design of secure architectures for the application deployment, access control mechanisms (such as firewalls), hardening of operating systems and network devices, use of cryptography, appropriate defense in depth, and application of the principle of least privilege, among others. Contrary to some recent statements by certain security practitioners who state that the detection and response layers are more important than prevention mechanisms, prevention mechanisms are necessary for the detection and response components to operate effectively.

Detection mechanisms, which will be the focus of this paper, are important to determine when the prevention mechanisms have failed or to identify new attacks. Network and host based intrusion detection systems, performing either misuse or

anomaly detection, may be used. Audit records generated by the operating system, network devices, and applications can also be used for detection.

Response mechanisms are primarily human processes that act in accordance to an incident response plan, using input from detection systems. Identification, research, and categorization of the incident all are necessary in order to mitigate risks presented by the incident's occurrence. Some automated procedures, such as "self-healing" software and firewalls and routers that reconfigure themselves, are also gaining popularity.

Methodology for analyzing traffic

While detection could involve a wide variety of techniques, this paper will focus on analyzing web traffic on the network between a web client and a web server. The choice of applying this methodology is not to indicate that network monitoring is the best method to identify abuse of web services, but it is one of the most popular and easiest to deploy without impacting other areas of the deployment.

HTTP protocol/packet dissection

First, it is important to understand how the HTTP protocol works when looking at web traffic. Luckily, the protocol is very simple to understand. The following section will present an introduction to the protocol and give analysts the ability to understand everything that is going to and from the web sites they monitor.

Let's examine a typical page request. When a client connects to a web server, the traditional TCP three-way handshake takes place between the two parties. Typically, the source port of the client is above 1024 and the destination port of the server is port 80.

Once the TCP session has been established, the client initiates by sending a HTTP GET request to web server. It may look something like this:

```
GET / HTTP/1.0\r\n
Connection: Keep-Alive\r\n
User-Agent: Mozilla/4.76 [en] (X11; U; Linux 2.2.17-14 i686)\r\n
Host: www.raid-symposium.org\r\n
Accept: image/gif, image/x-xbitmap, image/jpeg, */*\r\n
Accept-Encoding: gzip\r\n
Accept-Language: en\r\n
Accept-Charset: iso-8859-1,*,utf-8\r\n
\r\n
```

The server will then respond with a status and, optionally, the requested resource. A typical response will look something like this:

```
HTTP/1.1 200 OK\r\n
Date: Sun, 22 Apr 2001 00:19:50 GMT\r\n
Server: Apache/1.3.14 (Unix)\r\n
Last-Modified: Wed, 07 Mar 2001 18:22:15 GMT\r\n
Accept-Ranges: bytes\r\n
Content-Length: 991\r\n
Keep-Alive: timeout=15, max=100\r\n
Connection: Keep-Alive\r\n
Content-Type: text/html\r\n
\r\n
```

Data (228 bytes)

```
0 3c21 646f 6374 7970 6520 6874 6d6c 2070 <!doctype html p
10 7562 6c69 6320 222d 2f2f 7733 632f 2f64 ublic "-//w3c//d
20 7464 2068 746d 6c20 342e 3020 7472 616e td html 4.0 tran
30 7369 7469 6f6e 616c 2f2f 656e 223e 0a3c sitional//en">.<
40 6874 6d6c 3e0a 3c68 6561 643e 0a20 2020 html>.<head>.
50 3c6d 6574 6120 6874 7470 2d65 7175 6976 <meta http-equiv
60 3d22 436f 6e74 656e 742d 5479 7065 2220 ="Content-Type"
70 636f 6e74 656e 743d 2274 6578 742f 6874 content="text/ht
80 6d6c 3b20 6368 6172 7365 743d 6973 6f2d ml; charset=iso-
90 3838 3539 2d31 223e 0a20 2020 3c6d 6574 8859-1">.<met
a0 6120 6e61 6d65 3d22 4175 7468 6f72 2220 a name="Author"
b0 636f 6e74 656e 743d 2248 6572 76e9 2044 content="Herv. D
c0 6562 6172 223e 0a20 2020 3c6d 6574 6120 ebar">.<meta
d0 6e61 6d65 3d22 4745 4e45 5241 544f 5222 name="GENERATOR"
e0 2063 6f6e con
```

Note that this isn't the end of the response, but the frame size limits the response, so the server continues sending the response in the next frame, after the client has sent a TCP ACK for the first packet:

```
0 7465 6e74 3d22 4d6f 7a69 6c6c 612f 342e tent="Mozilla/4.
10 3720 5b65 6e5d 432d 4343 4b2d 4d43 4420 7 [en]C-CCK-MCD
20 4e53 4350 4344 3437 2020 2857 696e 4e54 NSPCPD47 (WinNT
30 3b20 4929 205b 4e65 7473 6361 7065 5d22 ; I) [Netscape]"
40 3e0a 2020 203c 7469 746c 653e 5241 4944 >.<title>RAID
50 2048 6f6d 6520 5061 6765 3c2f 7469 746c Home Page</titl
60 653e 0a3c 2f68 6561 643e 0a3c 626f 6479 e>.</head>.<body
70 3e0a 0a3c 6365 6e74 6572 3e3c 623e 5241 >.<center><b>RA
80 4944 3c2f 623e 3c62 3e3c 2f62 3e0a 3c70 ID</b><b></b>.<p
90 3e3c 623e 5265 6365 6e74 2041 6476 616e ><b>Recent Advan
a0 6365 7320 696e 2049 6e74 7275 7369 6f6e ces in Intrusion
b0 2044 6574 6563 7469 6f6e 3c2f 623e 3c2f Detection</b></
c0 6365 6e74 6572 3e0a 0a3c 703e 0a54 6865 center>.<p>.<p>.The
d0 2052 4149 4420 2077 6f72 6b73 686f 7020 RAID workshop
e0 7365 7269 6573 2069 7320 616e 2020 616e series is an an
f0 6e75 616c 2065 7665 6e74 2064 6564 6963 nual event dedic
100 6174 6564 2074 6f20 2074 6865 2073 6861 ated to the sha
110 7269 6e67 206f 660a 696e 666f 726d 6174 ring of.informat
120 696f 6e20 7265 6c61 7465 6420 746f 2074 ion related to t
130 6865 2069 6e74 7275 7369 6f6e 2d64 6574 he intrusion-det
140 6563 7469 6f6e 2061 7265 612e 0a0a 3c70 ection area...<p
150 3e0a 3c61 2068 7265 663d 2268 7474 703a >.<a href="http:
160 2f2f 7777 772e 7261 6964 2d73 796d 706f //www.raid-sympo
170 7369 756d 2e6f 7267 2f72 6169 6439 3822 sium.org/raid98"
180 3e52 4149 4427 3938 3c2f 613e 2077 6173 >RAID'98</a> was
190 2068 656c 6420 696e 204c 6f75 7661 696e held in Louvain
1a0 206c 6120 4e65 7576 652c 2042 656c 6769 la Neuve, Belgi
1b0 756d 2e0a 0a3c 703e 0a3c 6120 6872 6566 um...<p>.<a href
1c0 3d22 6874 7470 3a2f 2f77 7777 2e72 6169 ="http://www.rai
1d0 642d 7379 6d70 6f73 6975 6d2e 6f72 672f d-symposium.org/
1e0 7261 6964 3939 223e 5241 4944 2739 393c raid99">RAID'99<
1f0 2f61 3e20 7761 7320 6865 6c64 2069 6e20 /a> was held in
```

And even yet another, again after the TCP ACK from the client acknowledging the second section of the response:

```
0 5075 7264 7565 2c20 494e 2c20 5553 412e Purdue, IN, USA.
10 0a0a 3c70 3e0a 3c61 2068 7265 663d 2268 ..<p>.<a href="h
20 7474 703a 2f2f 7777 772e 7261 6964 2d73 ttp://www.raid-s
30 796d 706f 7369 756d 2e6f 7267 2f72 6169 ymposium.org/rai
40 6432 3030 3022 3e52 4149 4427 3230 3030 d2000">RAID'2000
50 3c2f 613e 2077 6173 0a68 656c 6420 696e </a> was.held in
```

```

60 2054 6f75 6c6f 7573 652c 2046 7261 6e63      Toulouse, Franc
70 652e 0a0a 3c70 3e0a 3c61 2068 7265 663d      e...<p>.<a href=
80 2268 7474 703a 2f2f 7777 772e 7261 6964      "http://www.raid
90 2d73 796d 706f 7369 756d 2e6f 7267 2f72      -symposium.org/r
a0 6169 6432 3030 312f 4346 505f 5241 4944      aid2001/CFP_RAID
b0 3230 3031 2e68 746d 6c22 3e52 4149 4427      2001.html">RAID'
c0 3230 3031 3c2f 613e 2077 696c 6c20 6265      2001</a> will be
d0 0a68 656c 6420 696e 2044 6176 6973 2c20      .held in Davis,
e0 4341 2c20 5553 412e 0a0a 3c2f 626f 6479      CA, USA...</body
f0 3e0a 0a3c 2f68 746d 6c3e 0a                  >...</html>.

```

Note that this is a very simple page without graphics. If there were graphics on the page, the client would initiate additional HTTP requests for each graphic as it was rendering the page.

A simple GET query looks like this (note that the parameters are encoded into the requested resource following the '?' character) :

```

GET /search?q=sansfire HTTP/1.0\r\n
Referer: http://www.google.com/\r\n
Connection: Keep-Alive\r\n
User-Agent: Mozilla/4.76 [en] (X11; U; Linux 2.2.17-14 i686)\r\n
Host: www.google.com\r\n
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg,
image/png, */*\r\n
Accept-Encoding: gzip\r\n
Accept-Language: en\r\n
Accept-Charset: iso-8859-1,*,utf-8\r\n
\r\n

```

Using a HTTP post query, the parameters are encoded into the body of the submission rather than the resource itself. This is typically used with larger parameters:

```

POST /config/login?8er7v2hg9u24b HTTP/1.0\r\n
Referer:
http://edit.yahoo.com/config/login?.src=my&.done=http://my.yahoo.com/&pa
rtner=&.intl=\r\n
Connection: Keep-Alive\r\n
User-Agent: Mozilla/4.76 [en] (X11; U; Linux 2.2.17-14 i686)\r\n
Host: login.yahoo.com\r\n
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg,
image/png, */*\r\n
Accept-Encoding: gzip\r\n
Accept-Language: en\r\n
Accept-Charset: iso-8859-1,*,utf-8\r\n
Content-type: application/x-www-form-urlencoded\r\n
Content-length: 163\r\n
\r\n
Data (163 bytes)

```

```

0 2e74 7269 6573 3d31 262e 7372 633d 6d79      .tries=1&.src=my
10 262e 6c61 7374 3d26 7072 6f6d 6f3d 262e      &.last=&promo=&.
20 696e 746c 3d75 7326 2e62 7970 6173 733d      intl=us&.bypass=
30 262e 7061 7274 6e65 723d 262e 753d 3463      &.partner=&.u=4c
40 3630 3776 6374 6534 6663 3626 2e76 3d30      607vcte4fc6&.v=0
50 2668 6173 4d73 6772 3d30 262e 6368 6b50      &hasMsgr=0&.chkP
60 3d59 262e 646f 6e65 3d68 7474 7025 3341      =Y&.done=http%3A
70 2532 4625 3246 6d79 2e79 6168 6f6f 2e63      %2F%2Fmy.yahoo.c
80 6f6d 2532 4626 6c6f 6769 6e3d xxxx xxxx      om%2F&login=xxxx
90 xxxx xx26 7061 7373 7764 3dxx xxxx xxxx      xxx&passwd=xxxxx
a0 xxxx xx                                     xxx

```

The server response to this login includes a HTTP redirect (note the type 302 response) and a cookie set:

```
HTTP/1.0 302 Found\r\n
Date: Sun, 22 Apr 2001 02:10:35 GMT\r\n
Location: http://login.yahoo.com/config/verify /\r\n
Connection: close\r\n
Content-Type: text/html; charset=iso-8859-1\r\n
Set-Cookie: Y=v=1&n=18mok07tbn&l=a8c1e7e/o&
p=m292rq8491j202&r=2m&lg=us&intl=us&np=1; path=/;
domain=.yahoo.com\r\n
\r\n
Data (112 bytes)
```

```
 0 5468 6520 646f 6375 6d65 6e74 2068 6173 The document has
10 206d 6f76 6564 203c 4120 4852 4546 3d22 moved <A HREF="
20 6874 7470 3a2f 2f6c 6f67 696e 2e79 6168 http://login.yah
30 6f6f 2e63 6f6d 2f63 6f6e 6669 672f 7665 oo.com/config/ve
40 7269 6679 3f2e 646f 6e65 3d68 7474 7025 rify?.done=http%
50 3361 2f2f 6d79 2e79 6168 6f6f 2e63 6f6d 3a//my.yahoo.com
60 2f22 3e68 6572 653c 2f41 3e2e 3c50 3e0a /">here</A>.<P>.
```

Future accesses to the server's domain would include this line in the HTTP headers originating from the client:

```
Cookie: Y=v=1&n=18mo7tbgp9n&l=a8c1e7e/o&p=m292rq8491j202&r=2m&lg=us
&intl=us&np=1
\r\n
```

Other client commands besides GET and POST include:

HEAD – Often used by web vulnerability scanners, HEAD is the same as GET, except the message-body is not returned. A scanner will perform a HEAD request for a vulnerable CGI and if a 200 OK message is returned, then it knows it has a positive hit. The scanner doesn't want to bother actually posting data to the CGI or parsing the results.

PUT – Used to upload files to the web server, some commercial web services support this option for file attachments for web-based e-mail, file repositories, and collaboration environments.

OPTIONS – Only some web-servers support this command. It is used for enumerating configuration information about the web server.

Here's an example, using netcat to directly connect to the web server port:

```
C:\tools>nc xxx.xxx.xxx.xxx 80
OPTIONS *
/r/n

HTTP/1.1 200 OK
Server: Microsoft-IIS/4.0
Date: Sun, 22 Apr 2001 03:28:55 GMT
Public: OPTIONS, TRACE, GET, HEAD, POST, PUT, DELETE
Content-Length: 0
```

DELETE – Rather self-explanatory, you can delete a resource on the web server with this command. Hopefully, most people configure their webservers to not accept this command, or only after successful authentication.

CONNECT – Rarely seen on the Internet itself, clients use this command on their internal web proxies that can support dynamically switching to SSL.

TRACE – A rare command that is used for troubleshooting proxy communications by requiring the end server that receives a web request to send a copy of that request back to the client.

The server returns a status code with every response to a HTTP message. Typically, you will see a 200 OK for successful client requests. Here are some other common status codes:

301 Moved Permanently / 302 Found / 307 Temporary Redirect – This is your typical redirect event from the webserver. In the header, there will be a Location: field that would redirect the web client to another URI. It should not be confused with a page-level redirect which has a meta-refresh tag to another site – that would be considered a 200 OK and the client would be forced to move to the next page when the refresh time ended. The status code will differ depending on what web server is generating the code and what techniques the webmaster is using for his or her redirects.

304 Not Modified – GET requests can be conditional. Usually, there will be an If-Modified-Since: field in the GET request when the browser has a page contents stored in its cache. The value of the field will be the last time the page was downloaded. If the page hasn't changed, the server sends back a 304 and nothing else happens.

400 Bad Request – The general “syntax error” message from the web server. The client did something wrong. A client that generates a 400 could be malicious or just broken – or the server doesn't support the request.

401 Unauthorized – Authentication is required, so then a browser that supports HTTP Authentication will present a dialog for the user to enter credentials, and send them in an Authorization: field to the server.

403 Forbidden – The server refused to answer the request. This could be due to IP address restrictions on the resource as applied by the server, file system permissions on a file, or other security oriented behavior by the server.

404 Not found – The resource does not exist – it could be a page or cgi program. A web vulnerability scanner will always pop up a lot of these, but so will web spiders, old bookmark lists, and webmasters with a proclivity for typographical errors.

500 Internal Server Error – The server encountered an error attempting to fulfill a client request. When this error happens on most web servers, it is something that a webmaster should check out.

501 Not Implemented – The client has asked for a method or special functionality that the server does not have. A web vulnerability scanner may generate this message.

One area that this section will not cover is HTTP authentication. However, it is important to understand how it works if your site supports it. Most major web sites on the Internet have eschewed HTTP authentication in favor of a customized cookie and session ID authentication scheme.

Much more information on the HTTP protocol is available from RFC 2616. State management (cookies) information is available from RFC 2965.

All web traffic that an analyst will monitor should be able to be decoded at the protocol level using the above information. However, analysis at the protocol level is not enough for most web sites. In order to detect web service abuse, an analyst needs to look at all levels, including the network and transport layer, the protocol level, and the application level.

Understanding the web service

In order to properly monitor your website, you need to understand what it is used for. Does it just serve up static pages or is most of the content dynamically generated? Are cookies used for session management, ad tracking, user preferences, or all of the above? Are some of the more rare client commands supported – such as PUT and DELETE? Should all of the traffic be SSL-encrypted or authenticated with HTTP authentication? Should a user always enter at a certain page or can they enter the application at any point? Are referrer fields used for session control or are they evidence of a potential cross-site scripting attack? What is the maximum field length supported for a form entry? The list of questions goes on and on.

As someone looking from the outside in (a pretty common relationship between network/operations folk and developers), it is important to understand the expected application behavior. Ask your developers to walk you through the application and ask lots of questions. Areas to cover include:

1. What are the names of all the CGIs and directory paths that an end user is expected to access? If you can capture a HTTP access request to other paths or executables, you might have an attack in progress. What's surprising about the Unicode exploit is not so much the details of the vulnerability but that no one thought to monitor that end users were accessing cmd.exe (or cmd1.exe)! Categorization of known bad paths (such as traversals (../), /sbin/, /system32, etc.) could be set at the highest level of severity and anomalous paths (either malicious or simply erroneous) at another severity level. Monitoring at this level will pick up the web vulnerability scanners easily.
2. What is the expected length of a user visit on the site? Is it open-ended (such as a content site) or should a user only be making a few transactions on the application server and the rest of the user interaction should be on the static web pages? If the latter is the case, you might want to monitor for a single host making many calls

to an application on the web server – they may be trying to figure out how it works (and how to break it) or have found a way to have the application give up interesting data or execute arbitrary commands.

3. If your application utilizes cookies, is it session-based or persistent? If it is session-based, a new client should not be presenting a session cookie immediately after the handshake unless they have connected before and failed to close their browser. Otherwise, it could be a case of session cookie theft or someone probing your application. Does the cookie value change throughout the interaction or is it static? Monitoring cookie changes without a corresponding Set-Cookie occurring previous to the presentment could definitely be evidence of a problem. Of course, proper application architecture and session management techniques can mitigate this risk, but monitoring these types of manipulations can provide invaluable evidence as a precursor to a successful hack.
4. What mechanism is used for user input – GET, POST, or both? How long should a GET or POST request be? What’s the minimum, expected, and maximum length of a cookie? Should there always be x number of fields filled out in a form submittal? How are hidden fields used?
5. What type of output will the web server be sending back to the client? Should it only be html, gif, and jpeg? If so, examination of the Content-Type: field for generic binary blobs or text files could be useful to determine when the web server has been compromised and the attacker is using it to get files off the server.
6. Can the user upload files or delete files? Is it through the HTTP PUT or DELETE command?
7. How is file access provided? Is there a common CGI used to display content? Is there syntax for accessing this CGI?
8. Is there an application-level administrative interface for developers, webmasters, or site operators?
9. Are there use cases (including expected and unexpected behavior) that are used for QA? Looking through the expected paths that an end-user is supposed to follow (ie. catalog page to shopping cart to checkout) can help identify potential attack vectors. Don’t be afraid to ask a good amount of “What if...?” questions.
10. Are there other application level security concerns that your developers or business owners have? For some websites, the content is the value – if a sitesucker comes to troll all of the content, is that a problem? If the site is supposed to be in “stealth” mode or not directly accessed without some passthrough mechanism, looking at the Referrer: fields may be very important.

Baselining your web server

Once you understand the dynamics of the web service you are trying to protect, it is important to baseline the traffic to understand what it should look like under normal use.

When the baseline is complete, you should have some of the necessary data points to start monitoring the service. Some data points, all of which are in the HTTP headers, to look for include:

- Use of HTTP and HTTPS – what resources are protected with HTTPS?
- Expected syntax of GET requests, including required fields, legal and illegal characters, size of fields
- Resource directories, extensions, or objects that are expected to be requested by a client
- Expected syntax of cookies, including size, originating servers, and expected resources that require cookie access
- Expected or unexpected Referer: fields
- Size of requests and responses
- Average number of transactions per client
- Expected HTTP commands (ie. GET and POST only?)
- Expected Content-types (ie. text/html, image/gif, etc.)
- Other expected headers, including Etag, Host:, and Agent:

If you are willing to look at the body of the message, other data points to look for are:

- Expected syntax of POST requests, including required fields, legal and illegal characters, size of fields
- Form fields presented by the server with expected client return values

If you are lucky, your QA or development organization may have some automated testing tools customized for your application that you can monitor the output from. Sophisticated organizations may present multiple types of web clients on different platforms, either directly or through emulation.

If such tools are not available, there are other ways of capturing a baseline of expected web traffic. The most obvious technique is to start capturing traffic on a production web server and examine how the service is used by untrusted end-users. Of course, one of the major problems with this technique is the same problem that learning anomaly detection systems have – how to ensure that the behavior observed is actually acceptable behavior. Therefore, analysis of live traffic for baselining needs to contain a certain level of statistical evaluation for the extremes identified.

Other techniques include using the `wget`² utility to retrieve web pages either in a single-page or recursive fashion. `Wget` is best for simple transactions and is good to simulate sitesuckers and unauthorized mirrors.

² `wget` is available at <http://www.gnu.org>.

Curl³ is an excellent utility with support for HTTP authentication, SSL, cookie support, and form data support. If there is expected path that a user is supposed to follow, you can script the entire transaction with curl. The ELZA project⁴ is another scripting tool that can be used. The scripts can include cookie presentment, form submission, and file downloads. To understand how the service reacts to errors, add abnormal conditions to your scripts, including incorrect passwords, expired or missing cookies, odd entry points into the application, invocation of redirects, and other common problems that end-users may present to the service.

Capturing and examining baseline traffic for manual analysis

Once you have determined how you are going to generate your traffic, you should determine how you are going to capture it. In order to have a reference set of traffic, using tcpdump to record traffic is recommended. In particular, setting a high snaplen (-s 4096), turning off name resolution (-n), with a filter for web traffic (port 80). Putting it all together, the command would be:

```
tcpdump -n -s 4096 -w capture port 80
```

Since most popular parsers can read tcpdump capture files, you can then use your capture file with a variety of utilities. Ethereal⁵ (using either the GUI version or command-line version, tethereal) is a good protocol analyzer to evaluate a capture file for non-automated analysis. Ethereal will inspect each layer, both at the Ethernet, IP, TCP, and the application layer of HTTP. Tcpreplay⁶ can be used to replay traffic in order to test your IDS system and signatures.

To examine the HTTP headers in a streamlined fashion, you need to discard the TCP and IP header information and examine the application layer payload directly. Tepparse⁷, a small utility available from Securify Labs, performs this extraction on a tcpdump capture file. A sample perl script is included in the distribution that uses tepparse to extract HTTP headers. The script and some sample output are displayed in Example 1.

Capturing and examining traffic for intrusion detection

Any network intrusion detection system that understands HTTP and is flexible enough to handle the granularity of inspection that is proposed in this paper should be adequate for capturing and examining web traffic for real-time intrusion detection. This paper's scope does not extend to evaluating different systems for their applicability for this method.

³ Curl is available at <http://curl.haxx.se>.

⁴ The ELZA is available at <http://www.stoev.org>.

⁵ Ethereal is available at <http://www.ethereal.com>.

⁶ Tcpreplay is part of NIDSbench, available at <http://www.anzen.com>.

⁷ Tepparse is available at <http://www.securify.com/labs/>.

With that in mind, let us set out some data capture requirements for web services:

1. All inbound web requests should be examined for known attack signatures. This is your typical misuse detection mechanism. As with standard misuse detection systems, this method of detection will provide primarily attack detection, not intrusion detection, if you are as vigilant about updating your servers with the latest patches as you are with updating your intrusion detection system with the latest signatures.
 - a. Strings to use include names of vulnerable CGIs, odd path structures, and system commands.
 - b. In addition, unsupported HTTP commands such as OPTIONS, PUT, or DELETE, might be important to examine.
 - c. SQL commands, often inserted in form fields with improperly check for SQL termination or append characters and pass the variable directly to a SQL query, resulting in database manipulation.
2. All server responses to web clients should be examined for unexpected data.
 - a. Content-types that are not expected, such as binary objects.
 - b. Known strings that indicate read or execute access of the file system, such as “command completed successfully” or directory listing artifacts such as “Directory of”, which is one of the lines in the header of a directory listing in Windows.
 - c. Red herrings inserted in application code, such as a standard detection comment in all web CGIs. If you can, have your developers add an innocuous comment line in each CGI. When detected in outbound traffic, then there is an indication that a showcode vulnerability was exploited.
 - d. Certain server responses such as 500 or 501, and potentially 400.
 - e. Error messages generated by an application processor or middleware components. Few successful SQL insertion attacks take place without the telltale error messages generated multiple times by the application or an ODBC driver, giving helpful information such as database, table, and column names. Application servers such as Weblogic also give up significant debug information that attackers use for reconnaissance.
3. Other inbound requests and traffic flows should be captured for more heuristic analysis.
 - a. Inbound client requests can be examined for syntactical compliance, for example, the length of a GET string, the structure of a HTTP request line, the size of a cookie, and the required elements of a GET query.
 - b. Transactional details, such as the number of requests to a certain CGI by a single client or the amount data transferred in a session, are also beneficial.
 - c. Repeated use of the HEAD command may indicate a web vulnerability scanner.
 - d. Application specific details, such as entry into an application dataflow halfway without a cookie, a cookie changing without a previous Set-

- Cookie command, or access to resource protected with HTTP authentication or SSL where no such mechanisms are used.
- e. POST examination, where submitted form values should be in an expected range and where hidden form fields should remain unmolested.
 - f. Inbound requests by netcraft.com, a web service used for identifying web servers and their underlying operating systems, is generally not a good situation. Attrition.org, a defacement mirroring site, is also not a good visitor to have, and both parties should raise suspicion.
 - g. Connection properties should be examined. Except in cases of fragmentation (which is suspicious in itself) or large form submission, a complete client request should fit into a single packet, multiple packets for a single request may indicate a telnet client attempting to connect to port 80 and manually enumerating the HTTP client request and looking for the HTTP server response.
 - h. Outbound SYN requests from a webserver are also something to look for. Normally, webserver shouldn't be connecting to the outside world, particularly to a client that has initiated an inbound request on port 80 shortly before. Such activity may be indicative of a trojan or a reverse shell.
 - i. Of course, traffic that does not conform to the HTTP specification may indicate that is not HTTP traffic at all. If a firewall is in place that allows inbound web connections, but does not inspect the traffic at an application level, there needs to be some mechanism to verify that only web traffic is being passed. An attacker who has compromised a webserver via a web service exploit may stop the web service and bind a rootshell to port 80 instead.

Example implementation

With the requirements in place, let's look at sample deployment of this level of inspection with a network IDS. For demonstration purposes, we will be using snort with tcpdump.

Since snort is able to analyze payloads by performing pattern-matching operations, standard snort content rules will work well for the first and second set of requirements. Snort comes with a good number of misuse detection signatures for web server vulnerabilities and new updates are published on a regular basis. Adding customized content rules specific to your application, both for inbound connections (for example, unsupported HTTP commands or SQL commands inserted into form fields) and server responses (such as content-types, error messages, and red herrings).

As an added bonus, content rules benefit from the advanced packet processing features available in snort or in supplemental plugins that deal with some level of IDS evasion techniques including stream reassembly, defragmentation, and HTTP preprocessing that deals with Unicode characters.

See Example 2 for a sample set of snort rules for detection of web service attacks.

For the third set of requirements, some pattern matching content rules (or in the case of the outbound SYN connection, a standard snort alert rule based on TCP flags) will suffice, but for others, detailed examination of the HTTP headers or payload is necessary. Snort currently cannot perform this level of inspection outside of basic pattern matching, and arguably, it shouldn't, as the resources required for advanced analysis combined with packet capture and decode may be too much.

As a result, there is a feature (`log_tcpdump`) in snort to dump particular packets that meet a rule criterion into a packet capture file. Other utilities can then perform additional inspection as a post-processing step. For example, an analyst could use snort to log all web traffic, `tcpdump` to extract the necessary headers, and a perl script to examine them for anomalies. An example would be to check for header values that exceed a specified maximum value. The IIS 5 .printer buffer overflow could be detected by this example, even without a specific misuse signature, as a successful overflow requires approximately 420 characters in the `Host:` field of the HTTP header.

Additional analysis could check for an abnormal number of HEAD requests from a single client, check for odd Content-Types, evaluate the syntax of a form submission, keep a counter of a particular client access to a CGI, or check for cookie manipulation. The variety of detection vectors and custom signatures is great, and a baseline created beforehand should give good direction on what analysis is most beneficial.

See Example 2 for a sample snort rule entry to log http packets to a file.

Conclusion

Analyzing web traffic, looking both for the known and unknown, is a necessary process for a complete intrusion detection capability. After understanding the threats that web services introduce, a security-conscious architect can implement the necessary defensive mechanisms, including those that deal with prevention and detection. An understanding of the HTTP protocol and the applications deployed can make detection more effective and comprehensive. Hopefully, the suggested techniques presented in this paper can assist information security staff along that road.

Examples

Example 1. Perl script to extract HTTP headers from `tcpparse`⁸

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

$/=""; # suck in paragraphs, not lines, at a time

$http_words = # these start an http paragraph
  "GET|POST|HEAD|PUT|OPTIONS|DELETE|CONNECT|TRACE";

my (@header_groups, $header_group);

while (<>) {

  # massive regex to match groups of http paragraphs
  #
  # here's the logic: match
  #
  # - any group of lines starting with a Src: ... Dst: line
  #
  # - followed by a line starting with either a http_word
  #   followed by a space or HTTP/
  #
  # - followed by the minimal set of any characters (typically
  #   multiple lines) that lead up to a line that is either empty
  #   or only contains a \r character
  #
  # '(?:' means group, but don't capture
  # '\A' means start of string; needed since there may not be a
  # preceding \n
  # '*' is minimal match rather than maximal match
  # we need '[\r]?' since HTTP has \r\n so blank line may have \r in it
  # /m modifier means that ^ and $ match embedded newlines
  # /s modifier means that . matches newlines

  @header_groups = m/(?:^\A)Src: [0-9] [\^\n]*?\n^(?:($http_words)
|HTTP\|/).*?^\[\r]?$/msg;

  foreach $header_group (@header_groups) {
    $header_group =~ s/\r//g; # nix the \r characters!
    print $header_group, "\n";
  }
}
}
```

Output of `tcpparse -ir capturefile | perl http-headers.pl`:

```
Src: 192.168.101.5 Dst: 209.73.57.168
GET / HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/4.76 [en] (X11; U; Linux 2.2.17-14 i686)
Host: www.plastic.com
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png,
*/*
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8

Src: 209.73.57.168 Dst: 192.168.101.5
HTTP/1.1 200 OK
Date: Fri, 04 May 2001 02:51:46 GMT
```

⁸ Tcparse and the following perl script were written by Scott Renfro, Director of Securify Labs.

```
Server: Apache/1.3.12 (Unix) mod_perl/1.24
Connection: close
Content-Type: text/html
```

Example 2. Sample snort configuration file

```
preprocessor http_decode: 80 8080

ruletype cli_webinspect
{
type alert
output alert_full: /var/log/snort/alert.cli
}

ruletype srv_webinspect
{
type alert
output alert_full: /var/log/snort/alert.srv
}

ruletype cli_webdump
{
type log
output log_tcpdump: clientweb.pcap
}

ruletype srv_webdump
{
type log
output log_tcpdump: serverweb.pcap
}

# Dumping web content for further analysis

cli_webdump tcp $EXTERNAL_NET any -> $HTTP_SERVERS 80
srv_webdump tcp $HTTP_SERVERS 80 -> $EXTERNAL_NET any

# Example Client -> Server rule - there should be plenty more of these
# available in the snort distribution

cli_webinspect tcp $EXTERNAL_NET any -> $HTTP_SERVERS 80 (msg:"Attempted
HTTP PUT command"; flags: A+; content:"PUT";)

# Example Server -> Client rule

srv_webinspect tcp $HTTP_SERVERS 80 -> $EXTERNAL_NET any
(msg:"Application showcode found"; flags: A+;
content:"INSERT_DETECTION_COMMENT_HERE";)
```