DAVID JOSEPHSEN

iVoyeur: permission to parse

David Josephsen is the author of *Building a Monitoring Infrastructure with Nagios* (Prentice Hall PTR, 2007) and Senior Systems Engineer at DBG, Inc., where he maintains a gaggle of geographically dispersed server farms. He won LISA '04's Best Paper award for his co-authored work on spam mitigation, and he donates his spare time to the SourceMage GNU Linux Project.

dave-usenix@skeptech.org

HAVE YOU EVER NOTICED THAT THERE is an adversarial relationship among the services we provide, the emergent security controls we put in place to protect them, and our monitoring tools? It works like this: We install a service—a Linux box, for example—and then we want to monitor it, so we use a monitoring system with ICMP echo requests (we ping it). Then, like clockwork, along comes portknocking, a clever bit of security-related trickery to muck things up.

I once had a friend whose love life worked the same way. He'd get a good thing going, and then along would come his French ex-girlfriend to mess things all up. He knew it was coming. He could see it a mile away, but she was just so cute and clever that he couldn't ever resist (and this too he knew). He even had a name for it. He called it a "malheur à trois" (doom triangle). Eventually he moved to Arkansas (a state, I'm told, that's like kryptonite to the French).

You and I both know that we can't resist portknocking no matter what state we run to (it's *that* cool), which is why we use flexible monitoring systems. We need to be able to work around things such as security-related trickery from time to time. And if it can happen to ping, it can happen to pretty much any service we run, so I thought it would make an interesting subject for a monitoring article or twelve. But rather than bore you with ICMP, I'd rather cover something a bit more complex and practically useful.

If HTTP loses the monitoring popularity contest to ICMP, it's not by much. And being a stateless protocol, with oodles of strange and intricate authentication mechanisms, it's an ideal candidate for us to take a look at. As a bonus, HTTP follows the malheur à trois pattern perfectly. Long ago we made a bunch of simple Web sites, for which we created a bunch of simple monitoring tools, and then along came single sign-on and Web services.

The safest way to make sure a Web site is functional is to request the page and parse it for some text. This accounts for pretty much everything that could go wrong, including application server trouble and even a malfunctioning database back end. But nowadays everyone is using form-based authentication, session cookies, and magically encoded URLs to handle Web site security. It's not enough that our tools support basic auth anymore, they need to act like real users, filling out forms, making multiple requests, and maintaining application state.

In this article I'll show you how to use a personal Web proxy to dissect typical modern HTTP authentication. Then I'll get you started scripting the monitoring of your Web apps with good-ol' wget. The general idea is to capture a valid authentication session with your Web site, and then extract and replay the key elements. In short, you'll perform a man-in-the-middle attack followed by a replay attack (and without ever removing your white hat).

To play along at home, you'll need to get a Web proxy, but not a proxy in the squid sense. You'll need a special-purpose proxy that will show you the content of the HTTP requests and replies between you and the site you want to monitor. Several of these exist, and I'm not particularly fond of any of them, but the one I tend to use the most often is Burp Proxy [1], which is part of a suite of tools called the Burpsuite. Launch Burpsuite, or the tool of your choosing, and point your browser at it by configuring your browser to use a proxy. For specifics on the use of Burp Proxy, check the help file [2].

Most proxies of this type, including Burp Proxy, have something akin to an "intercept" button. When intercept is "on" the proxy will intercept requests and prompt you to either allow or deny them. For our purposes, this isn't necessary, so I advise you to turn intercept off. With intercept off, all of the requests are still captured and stored, but you aren't prompted for anything. The stored requests are available in the history tab in Burp Proxy.

The Web app security I'm reverse-engineering today is actually in use by a real publicly facing entity. I simply poked around the various services-based sites I use on a regular basis for one that had a good mix of authentication-related stuff. I've anonymized the headers in the listings to avert phone conversations with angry lawyers. For the purposes of the article, assume that we need to monitor a shrubbery management app at www.mysite.com. This site is part of a larger, landscape-related management services organization, and as such they use single sign-on at www.authsite.com, so you can manage shrubbery and a little path running down the middle without having to log in twice.

HTTP conversations, as I'm sure you're already aware, are made up of a header and data section (similar to SMTP conversations). The server and client can use the headers to talk about things such as the HTTP version number and supported features. They'll also use the headers to pass cookies back and forth. The data section is for, well, data. Obviously, where authentication is concerned most of the interesting stuff is in the header section. The notable exception is when a form is used to collect the user name and password. When this happens, we'll be interested in the POST data from the client. Generally the client will make a request of the server, to which it receives a reply. In HTTP, the server can only react to what it is asked for, so the server uses things such as HTTP redirects to influence the client when it needs to. Requests take one of two forms: GET requests and POST requests. POST requests are used for submitting sensitive information such as user names and passwords.

To keep things simple in the example that follows, I've filtered out quite a bit of extraneous stuff such as requests for graphics and style sheets. I've also summarized a bunch of requests that provided me authentication-related cookies, because they weren't necessarily relevant to our automating things later. What's left are four key transactions that we'll need our monitoring script to replay to get things working. My point in telling you this is that in real life it takes a bit of time to separate the wheat from the chaff. Be patient.

So let's get started. I'm intimately familiar with this shrubbery site, as I use it quite a lot, so I already know that to monitor the page I want, I'm going to have to fill out a form, and I already know the URL of the authentication page, but I don't start my capture there. First I load a public page to see if it passes me any cookies. Many authentication setups expect you to act like a human, and when you don't they'll redirect you somewhere that suits their needs. For example, if you show up at an authentication page without certain cookies, then the authentication code may freak out because it can't figure out what you're asking for permission to see.

Freaking out will probably entail redirecting you back to some public section of the site. Automating reactions to this kind of thing can be difficult to do. Instead, act like a human and go someplace public first the way a human would. Firing up my proxy and loading the front page, I get the headers in Listings 1a and 1b. Listing 1a shows a request for the main page of mysite, and Listing 1b shows the reply. Sure enough, the server immediately hands me a session cookie. This is a pretty strong indication that our script is going to need to save and present cookies when we monitor this site in the future.

```
GET / HTTP/1.1
Host: www.mysite.com
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.8.0.2) Gecko/20060308 Firefox/1.5.0.2
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png
    ,*/*;q=0.5
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
```

LISTING 1A: HTTP HEADER FOR REQUEST IN A PUBLIC SECTION

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Cache-Control: public
pragma:
Set-Cookie: JSESSIONID=5BC21F0AC321558C088C4D13ADC35F0D;
Content-Type: text/html;charset=iso-8859-1
Date: Wed, 21 Nov 2007 17:03:22 GMT
Content-Length: 11086
```

LISTING 1B: RESPONSE TO THE REQUEST SHOWN IN LISTING 1A, WITH COOKIE

With the proxy in place, I proceed to make a request for something secure. For a few moments I'm bounced around to various pages on the site. Each of these represents some back-end application code that is attempting to determine who I am and whether I am allowed to view what I'm asking for. Along the way I pick up several more cookies and get transferred to HTTPS. One of the cookies is a monster called "s_sess," which appears to contain very specific information about what I'm asking to see. Another cookie, "s_pers," has some gobbledygook that's probably associated with who I appear to be and what level of access I currently possess. Eventually, the application decides that I can't be trusted and punts me to its parent single sign-on authority, authsite. The header of this last request, the one just before I'm redirected to authsite, is Listing 2a.

```
GET /home.jsp?cat=5 HTTP/1.1
Host: www.mysite.com
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.8.0.2) Gecko/20060308 Firefox/1.5.0.2
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Referer: https://www.mysite.com/
Cookie: JSESSIONID=5BC21F0AC321558C088C4D13ADC35F0D;s_sess=%20s_cc%3Dtrue%3B%20s_sq
    %3Dauthsiteprod%253D%252526pid%25253DUS%2525253AWelcome%2525253Emysite
    %2525253shrubberyProgram%2525253APersonalShrubbery%252526pidt%25253D1%252526oid
    %25253Dwww.mysite.com/home.jsp%2525253Fcat%2525253D5_1%252526oidt%25253D1
    %252526ot%25253DA%252526oi%25253D1%3B; s_pers=%20s_dfa%3Dauthsiteprod
    %7C1195667245382%3B
```

LISTING 2A: REQUEST FOR AUTHENTICATION, WITH COOKIES COLLECTED SO FAR

```
HTTP/1.1 302 Moved Temporarily
Server: Apache-Coyote/1.1
Set-Cookie: StaticTrackingCookie=dzGTdukyUUTcrTcOGzUd; Expires=Mon, 09-Dec-2075 20:17:50 GMT
Set-Cookie: TrackingCookie=24Od2TmMzzdhvdh8O4z2; Path=/
Location: https://www.authsite.com/shrubbery/us/action?request_type=authreg_ssologin&target=https
    %3A%2F%2Fwww.mysite.com%2Fhome.jsp%3Fcat%3D5
Content-Length: 0
Date: Wed, 21 Nov 2007 17:03:42 GMT
```

LISTING 2B: RESPONSE TO THE REQUEST IN LISTING 2A

As you can see, I've presented the various cookies I received in my interaction with mysite. The reply in Listing 2b is a redirect to the authsite. Before we go, we're given a few tracking cookies for good measure. So our monitoring scripts are certainly going to need to handle cookies if they expect to play well with this shrubbery management site. We could use our proxy to withhold some of these cookies, just to see which of them are actually required by the site and which are just nice to have, but the safest thing to do would probably be to make sure our script gets all of them. This appears to be a JSP back end after all, and one never knows what those Java guys are thinking.

At authsite, we're ping-ponged around for a while, picking up more cookies in the process. Finally, we're presented with a simple form asking us for our user name and password. Listing 3a displays the POST header and data that I send to authsite. Our new cookies are presented to the form processor as well as my user name and password, which can be seen toward the end of the POST URL. The server responds with some more cookies and a 302 redirect, as seen in Listing 3b. This redirect is to another URL on the authsite, and it appears to be related to requesting SSO-related credentials to access our originally requested shrubbery-related content.

```
POST /myshrubberybbage/logon/us/action?request_type=LogLogonHandler&location=us_logon2 HTTP/1.1
Host: www.authsite.com
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.8.0.2) Gecko/20060308 Firefox/1.5.0.2
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
```

```
Connection: keep-alive
Referer: https://www.authsite.com/myshrubbery/logon/us/en/en_US/logon/LogLogon.jsp?DestPage=https%3A%2F
      %2Fwww.authsite.com%2Fmyshrubbery%2Fus%2Faction%3Frequest_type%3Dauthreg_ssologin%26target
      %3Dhttps%253A%252F%252Fwww.mysite.com%252Fhome.jsp%253Fcat%253D5
Cookie: s_vi=[CS]v1|474464E90000173B-A170C2800002396[CE]; SaneID=67.88.91.16-1195664628842678; s_sess=
      %20s_cc%3Dtrue%3B%20s_sq%3Dauthsiteprod%253D%252526pid%25253DUS%2525253AMYCA-Login-
      LightVersion%2525253EUserManagement%2525253Aauthsite%252526pidt%25253D1%252526oid
      %25253Dfunctiononclick%25252528event%25252529%2525257Bjavascript%2525253Aif%25252528
      %25252521checkBeforeSumbit%25252528%25252529%25252529%2525257Breturnfalse%2525253B
      %2525257Ddocument.frmLogon.submit%25252528%25252529%2525253B%2525257D%252526oidt
      %25253D2%252526ot%25253DIMAGE%3B; s_pers=%20s_dfa%3Dauthsiteprod%7C1195667265665
      %3B; s_cc=true
Content-Type: application/x-www-form-urlencoded
Content-Length: 337

DestPage=https%3A%2F%2Fwww.authsite.com%2Fmyshrubbery%2Fus%2Faction%3Frequest_type
      %3Dauthreg_ssologin%26target%3Dhttps%253A%252F%252Fwww.mysite.com%252Fhome.jsp%253Fcat
      %253D5&Face=en_US&Logon=Logon&b_hour=11&b_minute=17&b_second=32&b_dayNumber=21&b_
      month=11&b_year=2007&b_timeZone=-6&UserID=dave&Password=iheartshrubbery&x=0&y=0
```

LISTING 3A: A POST TO THE AUTHSITE

```
HTTP/1.1 302 Found
Date: Wed, 21 Nov 2007 17:04:06 GMT
Server: IBM_HTTP_Server/2.0.42.2-PK29827 Apache/2.0.47 (Unix) DAV/2
Set-Cookie: shrubberyboxvalue=d9ad1ab0-02271d96-5153a860-770139b1;Domain=.authsite.com;Path=/; Secure
Cache-Control: no-cache="set-cookie,set-cookie2"
Expires: Thu, 01 Dec 1994 16:00:00 GMT
Set-Cookie: shrubberyboxpub=7d38d1a8936edc29f58b2859d260885e;Domain=.authsite.com;Expires=
      Fri, 13-Nov-2037 17:04:06 GMT;Path=/
location: https://www.authsite.com/myshrubbery/us/action?request_type=authreg_ssologin&target=https
      %3A%2F%2Fwww.mysite.com%2Fhome.jsp%3Fcat%3D5
Vary: Accept-Encoding
Keep-Alive: timeout=30, max=100
Connection: Keep-Alive
Content-Type: text/html
Content-Language: en
Content-Length: 0
```

LISTING 3B: RESPONSE TO THE POST IN LISTING 3A

```
HTTP/1.1 200 OK
Date: Wed, 21 Nov 2007 17:04:08 GMT
Server: IBM_HTTP_Server/2.0.42.2-PK29827 Apache/2.0.47 (Unix) DAV/2
Set-Cookie: MR=4;Domain=.authsite.com;Expires=Sat, 30-Jul-2039 18:50:49 GMT;Path=/
Cache-Control: no-cache="set-cookie,set-cookie2"
Expires: Thu, 01 Dec 1994 16:00:00 GMT
Set-Cookie: Domain=.authsite.com;Expires=Sat, 30-Jul-2039 18:50:49 GMT;Path=/
Vary: Accept-Encoding
Keep-Alive: timeout=30, max=100
Connection: Keep-Alive
Content-Type: text/html;charset=ISO8859-1
Content-Language: en
Content-Length: 363

<html>
<head>
```

```
<meta http-equiv="Refresh" content="1;
    url=https://www.mysite.com/home.jsp?cat=5&ctoken=2608C5DB4EFAAEE2B9B4BA4A0245C025062C70F042D494
    4F1AD94166EFBD3497A24EE95ADEBEE0E0&crIndex=0&crk=60387FA24B7E7BBBF7A54A08D48AC048&tier=CA&
    sid=67.88.91.16-1195664628842678">
</head>
<body>
</body>
</html>
```

LISTING 4: REPLY

When we follow the redirect, we're presented with the reply in Listing 4. This reply links us back to the shrubbery site by way of a Meta Refresh Tag. The URL in the tag is what I refer to as a "Magic URL." As you probably already know, authsite cannot give us a "yeah, he's good" cookie, since cookies can only be read by the domain that wrote them. Mysite can't read cookies authsite gave us. Instead, authsite gives us an authentication token in the URL. The magic URL should be cryptographically verifiable by mysite, should work only for us, and should be robust against replay attacks by folks pretending to be us (hence the magic). In practice it is rarely any of these things.

So how in heck do we automate all of this? In fact, it turns out to be pretty simple with the old standby, wget. This great piece of software handles cookies (if you tell it to), automatically follows redirects, and generally just does the right thing. With wget we can get from public site to SSL-enabled, protected content in three commands:

```
wget —no-check-certificate —delete-after —keep-session-cookies \
    —save-cookies mmmcookies http://www.mysite.com

wget —no-check-certificate —delete-after -keep-session-cookies \
    —save-cookies mmmcookies —load-cookies mmmcookies \
        https://www.mysite.com/home.jsp?cat=5

wget —no-check-certificate —keep-session-cookies \
    —save-cookies mmmcookies —load-cookies mmmcookies \
        -O parseme.html —post-data='request_type=authreg_ssologin&
        target=https://www.mysite.com/home.jsp?cat=5&Face=en_US&Logon=Logon&
        b_hour=12&b_minute=17&b_second=32&b_dayNumber=21&b_month=11&
        b_year=2007&b_timeZone=-6&UserID=dave&Password=iheartshrubbery
                &x=0&y=0' https://www.authsite.com/myshrubbery/us/action
```

The key cookie-related options are —keep-session-cookie, —save-cookies, and —load-cookies. They're all pretty self-explanatory. The save and load options take a filename as an argument and save cookies to, or load them from, the given file. The option —keep-session-cookies is necessary when you're dealing with JSP-style session cookies, since they won't be saved by default.

The first two commands use —delete-after to get rid of the file once it's downloaded, since we're not really interested in parsing any but the final content for errors. The last command uses —post-data to post the data we captured in Listing 3a. Once the data is posted, wget will automatically follow the redirects and meta-refresh, providing and saving cookies as necessary, finally providing a file called parseme.html. This file is the content we originally wanted, and it may be parsed to discover the state of the site.

This works great, and even lends itself to code reuse if you think ahead a little bit. The only caveat is perhaps that, because this particular POST data contains dates and times, you may have to programmatically generate them every time you run the script. This is pretty simple to do in any language you happen to fancy. More complicated authentication schemes may force you to parse tidbits out manually in interim steps, but I rarely run into something that wget doesn't just handle. If you're finding yourself doing a lot of parsing through interim HTML files for this or that, you might want to have a look at webInject [3]. It's another great tool which handles most of the error checking for you and even has a Nagios Plugin mode (but it doesn't automatically follow redirects, which is a bit of a drag).

Take it easy.

REFERENCES

[1] http://portswigger.net/suite/.

[2] http://portswigger.net/proxy/help.html.

[3] http://www.webinject.org/.