Bypassing Web Authentication and Authorization with HTTP Verb Tampering:

How to inadvertently allow attackers full access to your web application

Arshan Dabirsiaghi, Director of Research, Aspect Security
arshan.dabirsiaghi@aspectsecurity.com

Introduction

Many web environments allow verb-based authentication and access control (VBAAC). The rules for these security controls involve using the HTTP verb (also called method), such as GET or POST, as part of a security decision. For example, the Java EE web.xml file can specify a security constraint as follows:

```
<security-constraint>
   <web-resource-collection>
      <url-pattern>/admin/*</url-pattern>
      <http-method>GET</http-method>
      <http-method>POST</http-method>
   </web-resource-collection>
   <auth-constraint>
      <role-name>admin</role-name>
   </auth-constraint>
 </security-constraint>
```

This rule limits access to the /admin directory to users with the "admin" role. Many tutorials and public examples of secure configurations list POST, GET (and sometimes PUT) for the HTTP methods under which a security constraint applies.

Unfortunately, almost all the implementations of this mechanism work in an unexpected and insecure way. Rather than denying methods not specified in the rule, they allow any method not listed. Ironically, by listing specific methods in their rule, developers are actually allowing more access than they intended.

We can manipulate the HTTP verb to attempt to bypass or circumvent such security controls. The use of the HEAD method is the simplest case. The attacker can also try any one of the other valid HTTP verbs, including TRACE, TRACK, PUT, DELETE, and many more. Attackers can also try sending arbitrary strings, such as "JEFF," as the HTTP verb.

An application is vulnerable to VBAAC bypass if the following conditions hold:

1) it uses a security control that lists HTTP verbs;
2) the security control fails to block verbs that are not listed; and
3) it has GET functionality that is not idempotent or will execute with an arbitrary HTTP verb

All three of these conditions are quite common. Depending on the product and rule enforcement, HTTP verb tampering could be used to get around the protections provided by any of the following security components:

- Web application firewalls
- Container-level URL authentication/authorization
- Application-layer URL authentication/authorization

Obviously, not every product could be analyzed in depth. Testing your environment for susceptibility to VBAAC bypass with HTTP verb tampering is highly recommended.

Bypassing VBAAC with HEAD

The HTTP specification, RFC 2616 [1], specifies that HEAD requests should produce the same results as a GET request but with no response body. This means that if an attacker can use HEAD to bypass many VBAAC security mechanisms and execute protected functions. If GET requests were actually idempotent as envisioned in the RFC, this would not be a problem, but many frameworks and applications treat GET and POST identically.

Imagine a URL in your application that is protected by a security constraint that restricts access to the "admin" directory and lists GET and POST.

```
http://yourcompany.com/admin/admin.jsp?fn=deleteUser
```

If an attacker tries to force browse to the URL in a normal browser, the security constraint will check to see if the request is allowed. That rule will check to see if the HTTP verb in the attacker's request is one of the HTTP methods listed. Since the request came from a browser, it probably uses GET or POST and will be stopped by the security constraint.

However, if the attacker uses an HTTP proxy like OWASP WebScarab [2] to send the request using the HEAD method, the rule will not apply and the request will not be blocked. The application will direct this request to the GET handler, the deleteUser function will be successfully invoked, and the attacker will get an empty response back in the browser.

If the GET method were idempotent, as recommended in the RFC, then being able to invoke it with a HEAD request would not be a problem, since the state of the server would not be changed and the body

of the response is removed, thus eliminating any possibility of integrity or confidentiality issues, respectively. However, many web applications on the Internet today do not follow this recommendation, and a GET request can be used to invoke many significant functions and transactions.

Bypassing VBAAC with Arbitrary HTTP Verbs

Some web platforms, including both Java EE and PHP, allow the use of arbitrary HTTP verbs. These requests execute similarly to GET requests, making it possible for attackers to use these verbs to bypass flawed VBAAC implementations. Even worse, the response is not stripped off as it is for HEAD requests, so the attacker can see the unauthorized pages as if there were no protection.

If we use the "JEFF" method instead of "HEAD" in the example described in the previous section, the request will bypass the security constraint because it is not one of the listed methods (similar to HEAD). Because the request targets a JSP, the service method of the JSP will be executed and run as normal. Because the method in the attacker's request is not one of the recognized methods, the servers default to processing the request as if it were a GET. This means that the full response is returned and the attacker will see the results of the admin.jsp page. A security proxy like WebScarab could easily be programmed to replace the actual method with "JEFF" to facilitate browsing these pages as normal.

Many custom application endpoints will require state information in order to handle a request, such as information from a session. Given that the attacker is not authorized to access the endpoint, it is possible that the proper information will be in the session and the action will fail. This should not be considered a defense against this attack, only a fortunate circumstance.

We identified a few server/language combinations that allow VBAAC bypass with arbitrary HTTP verbs. This should not be considered an exhaustive list as we have not tested a wide variety of platforms.

- Apache 2.2.6/PHP
- Tomcat, WebSphere, and WebLogic when requesting a JSP (not a servlet)
- Possibly others…?

JSPs behave this way because their content is placed into a generic "service" method by the JSP compiler. Unlike an HttpServlet, the JSP service method is called for all HTTP methods. The HttpServlet service method delegates to specific handlers, such as doGet() and doPost(), and throws an exception if an unknown HTTP method is received. Therefore, the use of invalid HTTP methods does not work with servlets, and the attacker has to use HEAD requests, which do not return the response body.

Note that while servlets are vulnerable to HEAD requests, they are not vulnerable to arbitrary HTTP verbs, as they implement specific methods for each method, and return an error if an unsupported method is invoked.

Vendor Specifics

The following list shows that most major web and application servers accept HEAD by default, and silently transfer all HEAD requests to the GET handler. The RFC essentially dictates that the HEAD method should be processed by the GET handler since the headers received from HEAD must match those from GET. Therefore, to have guaranteed accuracy of those headers, the GET must be executed in normal fashion.

- IIS 6.0
- WebSphere 6.1
- WebLogic 8.2
- JBoss 4.2.2
- Tomcat 6.0
- Apache 2.2.8

Again, this should not be terribly surprising as it is a requirement in the RFC. Now that we know the servers accept HEAD, we will analyze some of the more popular out-of-application security mechanisms to see if they are susceptible to VBAAC bypass.

The following table displays some popular web security mechanisms and how they are affected by verb tampering techniques.

Security Control	Allow HTTP Verbs in Policy	HEAD Can Be In Policy	Policy Bypassable?
All Java EE (web.xml)	Yes	Yes	Yes
ASP.NET Authz	Yes	Yes	Yes (if not using deny all)
SiteMinder	Yes	Yes	No (possibly if default resource protection = unprotected)
.htaccess	Yes	Yes	Yes (if not using <LimitExcept>)

Java EE Containers

Consider the following, naïve web.xml <security-constraint> policy which is actually shipped in **Tomcat 5.5.20** in the example on how to "protect" a subset of URLs:

```
<security-constraint>
   <display-name>Example Security Constraint</display-name>
   <web-resource-collection>
      <web-resource-name>Protected Area</web-resource-name>
       <!-- Define the context-relative URL(s) to be protected -->
      <url-pattern>/security/protected/*</url-pattern>
       <!-- If you list http methods, only those methods are protected -->
       <http-method>DELETE</http-method>
      <http-method>GET</http-method>
      <http-method>POST</http-method>
       <http-method>PUT</http-method>
   </web-resource-collection>
  ...
 </security-constraint>
```

The comment above the HTTP method enumeration says it all: if you list HTTP methods, only those methods will be protected. Therefore, this rule will only fire if a request for anything in the /security/protected directory uses a verb in the <http-method> list. This logic is unfortunately as common as it is backwards, as can be seen from a simple Google Code Search for <http-method> [5].

The right way to implement this control would be to block any verbs that are not listed. However, that is not the way these mechanisms currently behave, and we can see in the snippet that HEAD is not in this list. Therefore, issuing an HTTP HEAD request will bypass the <security-constraint> entirely, and the application server will pass the request to the GET handler.

The best approach to securing Java EE applications is to remove all the <http-method> elements from security constraints in web.xml. This will simply make the constraints apply to all requests. However, if you absolutely must restrict access to a single method, there is a way to use web.xml to enforce this, but you have to set up two security constraints. In the example below, the top constraint applies the constraint to GET requests, and the second constraint blocks anything else.

```
<security-constraint>
    <web-resource-collection>
      <web-resource-name>site</web-resource-name>
      <url-pattern>/*</url-pattern>
      <http-method>GET</http-method>
    </web-resource-collection>
     ...
</security-constraint>

<security-constraint>
    <web-resource-collection>
      <web-resource-name>site</web-resource-name>
      <url-pattern>/*</url-pattern>
    </web-resource-collection>
     ...
</security-constraint>
```

ASP.NET Authorization

The built-in **ASP.NET authorization** [3] mechanism can be configured in a way that is vulnerable to VBAAC bypass. Here is an example rule from MSDN:

```
<authorization>
  <allow verbs="GET" users="*"/>
  <allow verbs="POST" users="Kim"/>
  <deny verbs="POST" users="*"/>
</authorization>
```

The intention of this rule is to allow only the "Kim" user to submit POST requests. This example is not vulnerable to bypass because GET requests are allowed to everyone anyway, so there is no security to bypass by using HEAD. However, consider the following authorization policy:

```
<authorization>
  <allow verbs="GET" users="Admin"/>
  <allow verbs="POST" users="Kim"/>
  <deny verbs="POST,GET" users="*" />
</authorization>
```

This policy would be vulnerable to VBAAC bypass since HEAD could be sent in which would bypass the authorization check and result in a silent forward to the GET handler.

This happens because .NET implicitly inserts an "allow all" rule into each authorization. Therefore a .NET developer must, after listing their role entitlements appropriately, append a "deny all" rule.

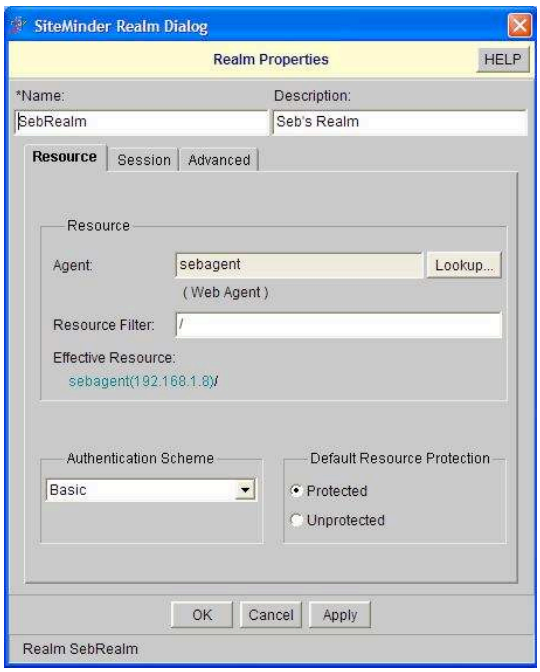
```
<authorization>
  <allow verbs="GET" users="Admin"/>
  <allow verbs="POST" users="Kim"/>
  <deny verbs="*" users="*" />
</authorization>
```

This will ensure that the only requests that pass the authorization check are those that have specified an HTTP verb that is in the authorization rule.

SiteMinder

SiteMinder [4] appears to protect against VBAAC in most configurations. Invalid methods appear to be discarded and rules that list verbs seem to appropriately limit requests to only the listed verbs.

There are still a few scenarios that we have not been able to test completely. These are related to realms that have the Default Resource Protection option set to "Unprotected" as shown below. When these realms are nested with protected realms, it may be possible to bypass protections with an unlisted HTTP method.



(Image taken from support.bea.com)

Further HTTP Method Confusion

There were a few other insecure behaviors noticed when doing the research for this attack. These quirks add up to make VBAAC bypassing more prevalent and easier to perform in some cases, but should be viewed as slightly separate from the attack in general.

Application Endpoints Don't Consider HTTP Method

Although developers may specify methods in their HTML <form> tags, many applications endpoints don't differentiate between GET and POST requests when acting on request data.

Technology	HTTP Data
Java/JSP	request.getParameter()
Struts	formBean.getValue()

PHP	$_REQUEST['value']
Python	request['foo']

All of these parameter retrieval techniques get data regardless of whether it comes from the querystring or the HTTP request body. So, although an endpoint may be intended to only receive POST requests, it is still quite possible the application will retrieve data from the querystring.

Defensive Measures to Prevent VBAAC Bypass Vulnerabilities

There are a few things an organization can do to prevent VBAAC bypass vulnerabilities.

1. Be sure to enable the "deny all" option in your VBAAC mechanism
2. Remove HTTP method restrictions from access control and authorization rules. All methods should be protected and they should not have to be listed during rule creation. For example, in Java EE, remove all <http-method> elements from web.xml files. This is unlikely to break functionality and only increases security.
3. Configure your web and application server to disallow HEAD requests entirely.
4. Make sure all functionality accessible through GET is idempotent in accordance with the specification.
5. In Java EE, never serve JSPs directly. Instead, put all of your JSP files in WEB-INF, and use a controller that uses RequestDispatcher.forward() to send a request to a JSP file. There are a number of reasons why direct access to JSPs is a bad idea, and this is just another reason.
6. If possible, developers should always invoke the parameters from the scope that they are expected to be in, and not from a dictionary collection of request information.

About Aspect Security

Aspect Security is the leading provider of application security risk management services. Millions of lines of critical application code are verified each month by Aspect's experienced penetration testing and code review specialists. Aspect teaches advanced hands-on security courses to thousands of architects, developers, and managers each year. Organizations with critical applications have gained control by implementing Aspect's Secure Development Lifecycle (SDL) program. Aspect is headquartered in Columbia MD. For information, visit www.aspectsecurity.com or call 301-604-4882.

About the Author

Arshan Dabirsiaghi is the Director of Research at Aspect Security, where in addition to his normal penetration testing and code review responsibilities, he spends his time researching Web 2.0 security and next generation attacks and defenses. Arshan is also the founder and lead author of the OWASP AntiSamy project. He has delivered tutorials at Blackhat and OWASP, and has been a featured speaker at both security and artificial intelligence conferences.

References

[1] – Hypertext Transfer Protocol – HTTP/1.1
http://www.faqs.org/rfcs/rfc2616.html

[2] – OWASP WebScarab Project
http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project

[3] – ASP.NET Authorization (URL-based)
http://msdn.microsoft.com/en-us/library/wce3kxhd(VS.80).aspx

[4] – CA SiteMinder Web Access Manager
http://ca.com/us/internet-access-control.aspx

[5] – Google Code Search for “<http-method>”
http://google.com/codesearch?hl=en&q=+%3Chttp-method%3E&start=10&sa=N