WAS Protect Discussion

v1, 16/03/2004

Introduction

Our first task it to determine the scope of the WAS Protect element. Some of the things we need it to do are:

- monitor and interpret HTTP traffic
- detect attacks
- prevent attacks
- protect specific vulnerabilities
- protect from classes of attack
- alter HTTP traffic in real-time

However, it should be noted that although defining an intrusion prevention language is our primary goal, a generic server side programming model could be, in fact, used for other purposes such as authentication, and url rewriting.

It is envisioned that the future standard will be used by software applications known as application gateways. These generally come in two forms:

- embedded in web servers
- standalone proxies (possibly appliances)

Standalone application gateways are easier to program for because they have full control over the HTTP stream. Gateways embedded in web servers are more difficult to design because they have to confirm to a web server's application programming interface, which come with significant constraints.Not surprisingly, none of the APIs are completely adequate for our purposes. The following APIs have been considered:

- Apache 1.x API
- Apache 2.x API
- IIS (ISAPI)
- Zeus (also ISAPI but an older version)
- Java Servlet specification 2.4

Also, the following related efforts were also considered:

- VulnXML
- ADVL, http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=avdl
- OVAL, http://oval.mitre.org/oval/
- mod_ssl, http://www.modssl.org/docs/2.8/ssl_reference.html
- mod_rewrite, http://httpd.apache.org/docs/mod/mod_rewrite.html
- mod_security, http://www.modsecurity.org/documentation/modsecurity-manual-1.8dev1.pdf
- The Common Gateway Interface, http://hoohoo.ncsa.uiuc.edu/cgi/

Architecture

- **The engine**; the purpose of the engine is to get access to the HTTP traffic, parse it to create the object model, and execute the scripts. Each request is validated as it is parsed.
- **The object model**; providing rules access to request and response data.
- **Context**; to allow some rules to apply to some requests and not the others.
- **Rules/Tests**; each rule consist of a number of statements.
- **Events**; for each request/response pair a zero, one or more events can be generated.
- **Central console**; gateways can operate on their own but, quite likely, some gateways will communicate with a central console. This communication is out of the scope of this document.

The object model

A HTTP server side programming model is needed as a basis to build everything else. Actually it already exists - in the form of the CGI specification. All server side progamming technologies have their own programming models and they mostly rely on the concepts established by the CGI spec. The problem, however, is that the CGI specification is not adequate for sophisticated requirements.

One approach is to try to extend the CGI specification, adding the missing features. The benefit of this approach is that most server side programmers are already familiar with the CGI specification, therefore the adoption will be easier.

On the other hand, one could argue that trying to extend the CGI spec is not the way to go, and that it cannot tolerate the required changes. The other approach is to build a completely new server side programming model using XML. Of course, a combination of both approaches is possible, to bring the best of two worlds together.

The idea is that each HTTP request will be parsed by the engine and presented to the script in a form of objects. An object can contain variables (or properties) or other objects. It is likely that some variables will be read-only. The following top-level objects are identified as follows:

- request (request_line, headers[], body, method, uri, version, query string, content_length, content_type, parameters, cookies[])
- response (status_line, headers[], body, status, content type, content_length)
- server
- connection (number of requests, transfer speed)
- session (id, creation time, number of requests served, last time of access, expiry time, original ip address)
- context

Possible properties are given in brackets. Access to raw data should always be given (for example request.request_line or request.headers[]). Where appropriate, raw data will be parsed into more meaningful properties (such as request.parameters).

The appearance of the session object implies the adoption of a stateful model because it requests client data to persist in spite of the stateless nature of HTTP. This model is more difficult to implement but more rewarding at the same time.

Built-in Functions

- strlen
- count (the number of elements in a collection)
- exists
- url_validate_decode
- lowercase
- unicode_validate
- remove_double_slashes
- compress whitespace
- unescape_characters
- remove_directory_selfreferences
- detect_remove_directory_traversal
- byte range validation
- ...possible other normalisation functions
- a special function to quantify the dynamics of change (needed to enforce limits and detect brute hacking attacks)

Blocks

A block is a series of statement.

Statements

- if
- foreach (member of a collection)
- foreach/except
- execute a named block (gosub)
- goto (label)

Operators

- not
- equal
- greater than
- less than
- regex match
- parenthesses
- IP address arithmetics? (eg is the IP address in the range)

Actions

- dispatch event (to the log, or to a remote location using syslog, snmp trap, any other method)
- delete/unset variable
- create/set variable
- create collection
- add a member to a collection
- add a member to a collection, to be automatically removed after a period of time
- stop request processing (with a status)
- read response from a file
- stop rule processing and allow the current request to proceed
- send a redirect
- skip a number rules
- execute external script
- create a timer?

Events

- unique event id
- message
- severity
- impact
- WAS Core reference id