Mod-security: Zabezpieczanie aplikacji internetowych. Integracja z technologiami Oracle.

Wojciech Dworakowski – SecuRing
wojciech.dworakowski@securing.pl

Aplikacje internetowe - czyli aplikacje udostępniające interfejs użytkownika za pomocą przeglądarki internetowej - to technologia która zrewolucjonizowała sposób tworzenia aplikacji. Niestety - jak z każdą nową technologią, również i z tą wiążą się nowe niebezpieczeństwa (przegląd technik ataku był prezentowany na zeszłorocznym Seminarium). Do zagrożeń tych możemy zaliczyć m.in.:

- Brak obsługi błędów i sytuacji specjalnych
- Manipulacje parametrami
- Path traversal
- Wykonywanie komend systemu operacyjnego
- SQL injection
- Ataki na sesje (cross-site scripting i session fixation)

W niektórych wypadkach (np. skrypty CGI) aplikacje internetowe mogą być podatne na klasyczne podatności takie jak „buffer overflow" czy „format strings".
Na domiar złego źródłem zagrożeń może być samo środowisko w jakim działa aplikacja. Zarówno oprogramowanie serwera WWW, oprogramowanie serwera aplikacyjnego jak i moduły pomocnicze służące do szybkiego tworzenia aplikacji webowych (np. Oracle Portal czy PHPNuke) mogą być podatne na różne metody ataku.
Problem polega na tym że protokołem w którym są przenoszone wszystkie te typy ataków jest protokół HTTP. Ze względu na to, tradycyjne środki ochrony (np. firewalle) nie są w stanie zapewnić właściwego poziomu ochrony, gdyż zarówno atak jak i zwyczajne korzystanie z aplikacji są widziane przez nie jako ruch HTTP lub HTTPS.
Żeby skutecznie chronić się przed atakami na aplikacje, powstało zapotrzebowanie na firewalle aplikacyjne, czyli filtry działające na wysokich warstwach modelu OSI, rozumiejące protokoły aplikacyjne, w szczególności – protokół HTTP.
Jednym z takich firewalli aplikacyjnych jest mod_security. Jest to moduł integrujący się z serwerem HTTP Apache, który pozwala na szczegółowe filtrowanie w zależności od zawartości danych przekazywanych w protokole HTTP. Jest to projekt Open Source, więc jest dostępny całkowicie nieodpłatnie. Apache jest serwerem HTTP stosowanym przez Oracle Application Server (Oracle HTTP Server), tak więc mod_security można zintegrować bezpośrednio z rozwiązaniami Oracle. Alternatywnym (i chyba lepszym) rozwiązaniem jest zastosowanie tego modułu na osobnym serwerze pełniącym rolę firewalla aplikacyjnego. Obie te konfigurację opisane zostaną w dalszej części.
Projekt ten jest rozwijany od roku 2002 i jego rozwój przebiega bardzo dynamicznie. Obecnie jest przygotowywana wersja 1.8. Pozwala to ocenić ten projekt Open Source jako dojrzały i stabilny – gotowy do stosowania w instalacjach produkcyjnych.
Oprogramowanie jest dostępne na stronie: www.modsecurity.org

Ogólna idea działania mod_security

Serwer Apache posiada budowę modularną. Jego podstawowa funkcjonalność może być rozszerzana za pomocą modułów ładowanych do Apache. Moduły te są dostarczane w postaci bibliotek so (unixy) lub dll (Windows). Włączenie danego modułu polega na załadowaniu biblioteki w konfiguracji Apache. Na takiej zasadzie jest zintegrowany z Apache mod_security.

Po uruchomieniu Apache, mod_security ma możliwość przechwytywania wszystkich zleceń płynących do serwera HTTP. Każde ze zleceń może być w zależności od skonfigurowanych reguł przepuszczone dalej – do dalszej obsługi przez serwer lub zablokowane. W praktyce blokowanie polega na zwróceniu błędu (np. 404 – not found) lub przekierowaniu na inną stronę. Obsługą wygenerowanego błędu lub przekierowania zajmuje się już serwer HTTP. Serwer HTTP odpowiada też za znormalizowanie ruchu podlegającego analizie. Tak więc gdy zlecenie HTTP dochodzi do serwera, to najpierw przechodzi ono przez moduły związane np. z rozkodowaniem SSL, usunięciem kododwania hexadecymalnego czy unicode i dopiero w takiej postaci trafia do mod_security. Dzięki bliskiej integracji z Apache i wykorzystaniu jego standardowych funkcji mod_security może być oprogramowaniem stosunkowo niewielkim i nieobciążającym serwera.

Podstawowe zasady konfiguracji reguł filtrowania

Mod_security dodaje do konfiguracji Apache (httpd.conf) kilka dodatkowych dyrektyw. Wszystkie wpisy związane z mod_security zaczynają się literami „Sec”.
Filtrowanie zleceń HTTP włącza się za pomoca komendy:

```
SecFilterEngine On
```

Zanim przystąpimy do konfiguracji reguł filtrowania warto włączyć inspekcję zawartości zelceń POST oraz sprawdzanie kodowania URL:

```
SecFilterScanPOST On
SecFilterCheckURLEncoding On
SecFilterCheckUnicodeEncoding On
```

Do każdej z definiowanych reguł filtrowania powinna być przypisana reakcja. Mod_security posiada dość rozbudowane możliwości reakcji:

- **deny** – zablokowanie ruchu
- **allow** – przepuszczenie ruchu (nie są rozpatrywane dalsze reguły)
- **status**:nnn – odpowiedź HTTP status nnn (np. 404 – not found)
- **redirect**:url – przekierowanie do innego URL
- **exec**:cmd – wykonanie komendy systemu operacyjnego
- **log** – zapisanie zdarzenia do logów
  Zdarzenie jest traktowane przez Apache jako błąd a więc standardowo jest zapisywane w error_log.
- **nolog** – brak zapisania w logach (przydatne przy łączeniu reguł)
- **pass** – zignorowanie danej regułki (w ciągu reguł)
- **pause**:nnn – zatrzymanie zlecenia na nnn millisekund.
  Może być przydatne np. przy blokowaniu ataków DDoS ale należy tego używać bardzo ostrożnie gdyż sztucznie wprowadzone opóźnienia mogą spowodować zakłócenia w działaniu aplikacji. Nietypowym zastosowaniem tej właściwości może być symulowanie opóźnień w środowiskach testowych.

Akcje można łączyć. Tzn. do jednej regułki można przypisać wiele działań.
Dla wygody, przed zdefiniowaniem reguł filtrowania należy zdefiniować reakcję standardową, która będzie podejmowana o ile do danej regułki nie zostanie przypisana inna akcja. Przykład definicji reakcji standardowej:

```
SecFilterDefaultAction "deny,log,status:404"
```

Powyższy zapis oznacza reakcję standardową polegającą na zablokowaniu zlecenia HTTP, zalogowaniu zdarzenia oraz odpowiedzi komunikatem HTTP 404 (not found). Tutaj warto zwrócić uwagę na to że o ile przy tradycyjnych firewallach zalecane jest odrzucanie

zabronionego ruchu „po cichu” bez żadnej odpowiedzi, to w przypadku firewalli aplikacyjnych raczej należy odpowiadać jakimś komunikatem HTTP. Brak jakiejkolwiek odpowiedzi ze strony serwera mógłby zakłócać działanie Apache i aplikacji.

Mod_security ma możliwość definiowania reguł ogólnych – nie będących filtrami a jedynie narzucających pewne reguły na cały przetwarzany ruch. Przykładem takiej reguły jest:

```
SecFilterForceByteRange 32 126
```

Reguła ta powoduje że będzą dopuszczalne tylko i wyłącznie zlecenia HTTP zawierające znaki z zakresu znaków ASCII od 32 do 126. W większości przypadków nie powinno to przeszkadzać działaniu aplikacji ponieważ protokoły aplikacyjne używające jako transportu HTTP z reguły używają tylko tych znaków (o ile nie przenoszą danych binarnych – a to należy raczej do wyjątków). Reguła ta powoduje ograniczenie ryzyka związanego z przesyłaniem danych binarnych. Ruch taki jest charakterystyczny np. dla ataków typu buffer overflow i dla ataków wiążących się z nieuprawnionym użyciem protokołu WebDAV.

Podstawowe regułki filtrowania można zdefiniować za pomocą składni skróconej:

```
SecFilter SŁOWO_KLUCZOWE [ AKCJA ]
```

Regułka SecFilter działa w ten sposób, że wyszukuje SŁOWO_KLUCZOWE w pierwszej linii zlecenia HTTP (czyli np. GET /index.jsp HTTP/1.1) oraz w danych zlecenia POST. Jeżeli słowo kluczowe zostanie znalezione to zostanie wykonana przypisana AKCJA (lub akcja standardowa jeśli do regułki nie została przypisana akcja). Tak więc składnię skróconą SecFilter można zastosować do poszukiwania słów kluczowych w ciągu URI oraz parametrach zleceń GET i POST.
W przypadkach gdy te możliwości inspekcji zlecenia HTTP są niewystarczające, można użyć składni pełnej:

```
SecFilterSelective ZMIENNA SŁOWO_KLUCZOWE [ AKCJA ]
```

Składnia pełna daje dostęp nie tylko do pierwszej lini zlecenia HTTP i parametrów POST ale również do nagłówków i pozostałych części zlecenia HTTP. Pozwala też na wygodniejsze operowanie parametrami przekazywanymi w zleceniach GET i POST. W stosunku do składni podstawowej mamy tu możliwość zdefiniowania zmiennej która zostanie przeszukana.

Przykładowe zmienne:

REMOTE_ADDR – adres IP klienta
REMOTE_HOST – nazwa klienta
REQUEST_METHOD – metoda HTTP (np. GET, POST, HEAD)
REQUEST_URI – ciąg URI
AUTH_TYPE – sposób uwierzytelnienia HTTP
SERVER_NAME – nazwa serwera (przydatne w przypadku serwerów wirtualnych gdy jeden Apache obsługuje wiele serwerów)
TIME, TIME_YEAR, TIME_MON, TIME_DAY, itd – czas
POST_PAYLOAD – zawartość zlecenia POST (parametry)
ARGS – wszystkie parametry (ze zlecenia POST i po znaku ? w zleceniu GET)

Można również używać zmiennych specjalnych:

HTTP_nazwa – nagłówek HTTP „nazwa” np. HTTP_USER_AGENT – nagłówek User-agent (nazwa przeglądarki)
ARG_nazwa – konkretny argument zlecenia GET/POST. Np. ARG_UZYTKOWNIK – parametr „uzytkownik” przekazywany z formularza

COOKIE_nazwa – zmienna przekazywana w cookie. Np. COOKIE_USER_ID – zmienna user_id przekazana z cookie użytkownika.

Zmienne można łączyć za pomocą operatrora | oraz negować (wprowadzać wyjątki) za pomocą operatora **!** Przykładowo: ARGS|!ARG_nazwa spowoduje przeszukanie wszystkich argumentów z wyjątkiem argumentu „nazwa".

Bardzo dużą zaletą mod_security jest możliwość definiowania słów kluczowych za pomocą wyrażeń regularnych w składni PCRE (Perl Compatible Regular Expressions). Daje to bardzo rozbudowane możliwości definiowania słów kluczowych. Wyrażenia regularne są standardem wywodzącym się ze świata unixów. Są stosowane jako standard opisu słów kluczowych m.in. w takich narzędziach systemowych jak *grep*, *sed*, *awk* czy *perl*. Również Oracle w wersji 10g wprowadziło wyrażenia regularne do składni SQL i PL/SQL [1].
Omówienie wszystkich możliwości wyrażeń regularnych to temat na całą książkę. Jednak dla lepszego zrozumienia dalszych przykładów w ramce podaje podstawowe operatory wyrażeń regularnych. Więcej informacji o wyrażeniach regularnych można znaleźć np pod adresami:
http://www.pcre.org
http://etext.lib.virginia.edu/helpsheets/regex.html
lub w dokumentacji Oracle 10g.

Wyrażenia regularne (PCRE):

Metaznaki:

.	(kropka) dowolny znak
[...]	dowolny znak z listy między nawiasami (lista może być przedstawiana jako zakres np.: **[A-Z]**)
[^...]	dowolny znak z wyjątkiem znaków z listy między nawiasami

Powtórzenia:

?	dowolny znak zero lub jeden raz
*	poprzedni element zero lub więcej razy (np.: **.*** oznacza dowolny ciąg znaków)
+	poprzedni element raz lub więcej razy
{n}	poprzedni element n razy
{n,m}	poprzedni element od n do m razy

Modyfikatory pozycyjne:

^	na początku linii
$	na końcu linii
\<	na początku słowa
\>	na końcu słowa
\b	na początku lub na końcu słowa

Operatory logiczne:

|	logiczny operator OR (lub)
!	zaprzeczenie

Operacje na buforze danych

(wyrażenie)	Znaleziony w wyniku działania wyrażenia ciąg jest ładowany do bufora
\1	Odwołanie do zawartości bufora

Znaki specjalne muszą być chronione przed interpretacją za pomoca znaku ****

[1] Writing Better SQL Using Regular Expressions:
http://otn.oracle.com/oramag/webcolumns/2003/techarticles/rischert_regexp_pt1.html
Using Regular Expressions:
http://otn.oracle.com/obe/obe10gdb/develop/regexp/regexp.htm

Na koniec tego wprowadzenia do pisania reguł warto jeszcze wspomnieć że reguły mod_security można łączyć w ciągi. Robi się to za pomocą słówka chain:

```
SecFilterSelective reguła1 chain SecFilterSelective reguła2
```

Przykładowe reguły filtrowania

Możliwości mod_security najlepiej poznać na przykładach. Poniżej przedstawiam kilka przykładów reguł podnoszących bezpieczeństwo serwera WWW i chroniących aplikacje.
W poniższych przykładach założono że standardową akcją jest odrzucenie zlecenia, a więc dopasowanie wyrażenia powoduje zablokowanie zlecenia.

Zapobieganie atakom na aplikacje

Zabezpieczenia przed typowymi metodami wykorzystania podatności w aplikacjach internetowych:

- Zabezpieczenie przed próbami uruchamiania komend systemu operacyjnego

  ```
  SecFilter "/bin/.*sh"
  SecFilter cmd.exe
  ```

- Zabezpieczenie przed atakami typu path-traversal. Są to ataki polegające na próbach wyjścia z podstawowego katalogu przez doklejenie do zmiennej interpretowanej jako ścieżka katalogu, ciągu typu /../../../ (wyjście w drzewie katalogu o jeden lub więcej poziomów wyżej)

  ```
  SecFilter "\.\./"
  ```

- Zabezpieczenie przed atakami cross-site scripting. Ataki te polegają na użyciu kodu JavaScript w zmiennej. Regułka odrzuca wszystko co zawiera ciąg <script>. Prostą regułkę intruz mógłby obejść podając np. ciąg < script>. Postaramy się stworzyć regułkę bardziej uniwersalną :

  ```
  SecFilter "<[[:space:]]*script"
  ```

  [[:space:]] w wyrażeniu regularnym oznacza klasę znaków typu „odstęp” tzn.: znak spacji lub inny znak oznaczający odstęp (np. TAB, CR, LF, ASCII-255).

- Zabezpieczenie przed podstawowymi atakami SQL-injection. Ataki te polegają na doklejaniu kodu SQL do zapytań, więc należy odfiltrować wszystkie ciągi znaków zawierające komendy SQL:

  ```
  SecFilter "delete[[:space:]]+from"
  SecFilter "insert[[:space:]]+into"
  SecFilter "select[[:space:]]+from"
  SecFilter "or[[:space:]]+(.+)=\1"
  ```

  Ostatnia reguła zabezpiecza przed doklejeniem wyrażenia logicznego które jest zawsze prawdziwe (np.: `or 1=1` , lub: `or nnn=nnn`)

  Uwaga: Skonstruowanie uniwersalnych reguł które chroniłyaby przed atakami SQL-injection jest bardzo trudne albo wręcz niemożliwe. Problem polega na tym, że fragmenty doklejanych zapytań będą parsowane przez interpreter SQL a my mamy do dyspozycji wyrażenia regularne. SQL jest językiem bardzo rozbudowanym, przez co ten sam skutek można osiągnąć konstruują inne zapytanie. Przykładowo w ciągu:

  or ‘abcde’=’abcde’

  Można zastosować operator sklejania ciągów znaków:

or ‘abcde’=’abc’+’de’

Warunek zawsze prawdziwy można też skonstruować w SQL na wiele różnych sposobów. A więc należy sobie zdawać sprawę, że w niektórych wypadkach, nawet filtrowanie na wyższych warstwach daje się obejść.

Ogranicznanie podejrzanej aktywności

Dobrą praktyką jest również ograniczanie ruchu HTTP tylko do zleceń charakterystycznych dla działania przeglądarek internetowych i zleceń charakterystycznych dla danej aplikacji, czyli normalizacja zleceń. Znacznie utrudnia to intruzom wykorzystanie niektórych technik poszukiwania podatności w aplikacjach.

- Przepuszczenie tylko i wyłącznie metod GET i POST a zablokowanie wszystkich innych (np. OPTIONS czy HEAD):

  ```
  SecFilterSelective REQUEST_METHOD "!(GET|POST)"
  ```

- Zablokowanie zleceń w których brak nagłówków User-Agent, Host, Accept, Keep-Alive:

  ```
  SecFilterSelective “HTTP_USER_AGENT|HTTP_HOST” “^$”
  ```

  Nagłówki te są ustawiane w typowych zleceniach wysyłanych przez przeglądarkę. Intruzi podczas rekonesansu często używają prostych narzędzi (np. telnet) i nie ustawiają wszystkich nagłówków tak jak przeglądarka.

- Czasami zamiast blokowania nietypowych zleceń lepiej jest je zapisać do późniejszej analizy. Poniżej przykład reguły która zapisuje do logów ale przepuszcza zlecenia HTTP bez ustawionych nagłówków Accept i Keep-Alive (nagłówki takie są ustawiane przez typowe przeglądarki):

  ```
  SecFilterSelective “HTTP_ACCEPT|HTTP_KEEP_ALIVE” “^$” log,pass
  ```

- Jeśli przeglądarka przedstawia się za pomocą cookie, to przepuszczenie wyłącznie tych zleceń w których parametr „SessionID” składa się tylko z cyfr:

  ```
  SecFilterSelective COOKIE_sessionid "!^(|[0-9]{1,9})$"
  ```

  Sesyjność w aplikacjach internetowych jest obsługiwana przez ustawienie identyfikatora sesji. Identyfikator ten ma format charakterystyczny dla danego serwera aplikacji. Identyfikator sesji jest zwykle przekazywany przez zmienną w cookie a czasami przez zmienną wysyłaną jako parametr zlecenia GET/POST. Reguła takiego typu jak powyżej wymusza stosowanie identyfikatora sesji w takim formacie jak spodziewa się tego serwer aplikacji. Ogranicza to możliwość manipulacji tym parametrem.

  Warto zaznaczyć że analogiczne reguły można stosować do sprawdzania formatu praktycznie każdego parametru przekazywanego ze środowiska użytkownika. Przykładowo – sprawdzenie czy zmienna ID (przekazywana w parametrach GET lub POST) istnieje i jest złożona z od 1 do 6 cyfr :

  ```
  SecFilterSelective ARG_id "!^[0-9]{1,6}$"
  ```

  O potencjalnej skuteczności takiego zabezpieczenia można się przekonać wpisując wartość nieliczbową (np. „aaaa”) w miejsce w którym aplikacja internetowa spodziewa się wartości liczbowej (np. zmienna „id” przekazywana jako parametr GET po znaku ?). Zachęcam do eksperymentów na niektórych znanych polskich serwisach internetowych ;)

- Przepuszczenie całego ruchu dla jednego adresu IP (np. stacji administratora):

  ```
  SecFilterSelective REMOTE_ADDR "^10.1.1.3$" nolog,allow
  ```

  Tego typu reguła musi pojawić się przed wszystkimi innymi.

Reguły wymuszające znormalizowany ruch warto stosować również wtedy jeśli klientem aplikacji nie jest przeglądarka internetowa tylko aplikacja działająca po stronie klienta. Wtedy można zastosować dodatkowe zabezpieczenie polegające np. na ustawianiu w wysyłanych przez klienta zlecenia HTTP dodatkowego nagłówka lub parametru i sprawdzaniu czy ten parametr istnieje po stronie serwera.
Warto zauważyć że tego reguły sprawdzające normalizację zeleceń, zabezpieczają zarówno przed obecnie znanymi metodami ataku na aplikacje jak również (w pewnym stopniu) przed metodami które mogą ujrzeć światło dzienne w przyszłości.

Mod_security można wykorzystać również do pewnych zautomatyzowanych działań administracyjnych. W jednym z artykułów autor mod_security – Ivan Ristic, podaje przykład reguły którą wysyła do niego mail każdorazowo gdy jego szef zapomni swego hasła ;)

```
SecFilterSelective REQUEST_URI "login_failed\.php" chain
    SecFilterSelective ARG_username "^ceo$"
    log,exec:/home/apache/bin/notagain.pl
```

Jest to przykład połączenia dwóch reguł za pomoca składni „chain”. Pierwsza reguła uruchamia się tylko wtedy gdy pobierany jest określony plik z serwera (strona z informacją o błędzie logowania). W takim wypadku uruchamiana jest druga reguła sprawdzająca czy nazwa użytkownika brzmi „ceo”. Jeśli tak jest to uruchamiany jest skrypt.

Zapobieganie atakom na oprogramowanie Oracle

Niektóre wersje oprogramowania Oracle (Apache/Oracle HTTP Server i towarzyszące mu moduły) zawierały błędy lub niebezpieczne ustawienia standardowe, które można wykorzystać do skutecznego ataku na dane lub system operacyjny serwera. Mod_security można wykorzystać również do ograniczenia możliwości ataków na oprogramowanie Oracle wykorzystujących jako transport protokół HTTP.

- Zabronienie dostępu do skryptów demo standardowo udostępnianych w instalacjach Oracle HTTP Server:

  ```
  SecFilterSelective REQUEST_URI “demo”
  ```

- Wcześniejsze wersje Oracle HTTP Server posiadały błąd pozwalający na pobranie kodu źródłowego stron JSP. Atak taki polegał na pobraniu źródła strony z katalogu /_pages. Katalog ten nie jest potrzebny dla użytkownika więc dostęp do niego można zablokować:

  ```
  SecFilterSelective REQUEST_URI “_pages\/”
  ```

- Zablokowanie dostępu do stron administracyjnych różnych modułów:

  ```
  SecFilterSelective REQUEST_URI “pls\/.+\/admin_”
  SecFilterSelective REQUEST_URI “dms0|dms\/DMSDump”
  SecFilterSelective REQUEST_URI “servlet\/Spy”
  SecFilterSelective REQUEST_URI “oprocmgr”
  ```

- Zablokowanie prób pobrania plików konfiguracyjnych np: plsql.conf, XSQLConfig.xml, soapConfig.xml. W niektórych wersjach było to możliwe dzięki wykorzystaniu technik *double-encoding* i *path traversal*.:

```
SecFilterSelective REQUEST_URI
    “plsql.conf|XSQLConfig.xml|soapConfig.xml”
```

- Starsze wersje Oracle HTTP Server udostępniały przez interfejs mod_plsql wszystkie procedury z pakietu owa_util. Umożliwiało to wiele ciekawych ataków na serwer i na aplikacje (np. pobranie kodu źródłowego dowolnej procedury za pomoca owa_util.showsource). Zabezpieczenie może polegać na wyblokowaniu zleceń charakterystycznych dla wywołań procedur z pakietu owa_util:

  ```
  SecFilterSelective REQUEST_URI “pls\/.+\/owa_util\.”
  ```

Powyżej pokazałem tylko niektóre z przykładów zastosowań mod_security. Chodziło mi raczej o zaprezentowanie na przykładach sposobu konstrukcji reguł a nie o stworzenie kompletnego zestawu sygnatur do wykorzystania. W przypadku wdrożeń mod_security zestaw reguł należy dostosować do specyfiki konkretnego serwera aplikacji i konkretnej aplikacji.

Możliwości dodatkowe

Oprócz wyżej zaprezentowanych możliwości związanych z filtrowaniem zleceń HTTP, mod_security posiada wiele możliwości dodatkowych. Na przykład pozwala nie tylko na inspekcje zleceń płynących do serwera, ale również na inspekcje odpowiedzi serwera. Pozwala to np. na zablokowanie informacji o błędzie serwera.

```
SecFilterScanOutput On

SecFilterSelective OUTPUT “error|Error|ERROR” deny,status:404
```

Informacje o wystąpieniu błędu powinny być obsługiwane w sposób świadomy przez serwer i aplikację. W przeciwnym wypadku mogą ujawniać dość istotne informacje. Np. błąd składni SQL powoduje często wypisanie w komunikacie błędu części zapytania SQL które wywołało błąd. Znacznie ułatwia to zadanie atakującemu aplikację.

Inną ciekawą możliwością jest podmienianie w locie sygnatury serwera zwracanej w nagłówkach HTTP. W ten sposób możemy wprowadzić potencjalnego intruza (a jeszcze skuteczniej – programy zautomatyzowane) w błąd:

```
SecServerSignature "Microsoft-IIS/5.0"
```

Standardowo mod_security zapisuje w logach tylko podstawowe informacje o błędzie. Możliwe jest włączenie zapisywania do osobnego logu pełnej informacji o zleceniach które zostały wyłapane przez regułki mod_security. Robi się to w następujący sposób:

```
SecAuditEngine RelevantOnly
SecAuditLog logs\audit_log
```

Mod_security a Oracle

Mod_security można zintegrować bezpośrednio z Oracle HTTP Server. Sprawdziłem jego współpracę z OHS zarówno będącym elementem bazy jaki i 9i Application Server.
Dokumentacja mod_security opisuje instalację mod_security ze źródeł lub z wykorzystaniem dostępnych wersji binarnych. Instalacja ze źródeł (oprócz kompilatora gcc) wymaga obecności skryptu $ORACLE_HOME/Apache/Apache/bin/apxs , który nie jest obecny we wszystkich instalacjach OHS.
Instalacja z binariów udostępnionych na stronach projektu (www.modsecurity.org) jest dość prosta. W przypadku OHS polega ona na:

1. Skopiowaniu pliku mod_security.so (w przypadku Linuxa) lub mod_security.dll (w przypadku Windows) do katalogu $ORACLE_HOME/Apache/Apache/modules
2. Dopisaniu do $ORACLE_HOME/Apache/Apache/conf/httpd.conf linni:

   ```
   LoadModule security_module modules/mod_security.dll
   AddModule mod_security.c
   ```

3. Skonfigurowaniu mod_security. Dla wygody można to zrobić w osobnym pliku, który dołączy się za pomocą polecenia *include* na końcu *httpd.conf*

   ```
   include "C:\ora9ias\Apache\Apache\conf\modsecurity.conf"
   ```

W przypadku integrowania mod_security z Oracle Application Server warto przepuścić wszystkie zlecenia płynące bezpośrednio z adresu IP naszego serwera.:

```
SecFilterSelective REMOTE_ADDR "^10.1.1.100$" nolog,allow
```

Jest to spowodowane tym, że niektóre z modułów Application Server-a komunikują się z Oracle HTTP Server w sposób inny niż przeglądarki HTTP. Np. zauważyłem że raz na minutę serwer sprawdza czy serwis HTTP działa poprawnie przez wywołanie metody HEAD. Zlecenie te nie zawiera żadnych dodatkowych nagłówków HTTP.

Oracle HTTP Server nawet w najnowszej wersji 10gR1 nadal wykorzystuje oprogramowanie Apache 1.3.x. Należy o tym pamiętać konfigurując mod_security. Największym ograniczeniem dla mod_security współpracującego z Apache 1.3 jest brak możliwości analizy odpowiedzi serwera.

Jak zbudować firewall aplikacyjny na osobnym serwerze?

Apache 1.3.x nie jest dobrze przygotowany na dołączanie modułów filtrujących ruch. W związku z tym mod_security dla Apache 1.3 musi wykorzystywać pewne sztuczki programistyczne. Sam autor mod_security pisze w dokumentacji:

> *"Apache 1.x does not offer proper infrastructure to intercept requests. Since what mod_security is doing is a hack,"*

W związku z powyższym nie zalecałbym bezpośredniego integrowania mod_security z Oracle HTTP Server w instalacjach produkcyjnych (Apache 1.3 jest stosowany we wszystkich obecnych wersjach Oracle HTTP Server). Nawet gdyby nie byłoby żadnych przeciwskazań do integrowania mod_security z Apache 1.3 to wydaje mi się, że taka ingerencja w oprogramowanie Oracle mogłaby być zbyt ryzykowna.
W związku z tym zalecałbym instalację mod_security na osobnym serwerze lub osobnym serwisie HTTP działającym na tym samym serwerze (na innym porcie). Taki serwer HTTP powinien działać jako reverse proxy. Funkcjonalność ta polega na tym że serwer obsługuje wszystkie zlecenia płynące od klientów i przekazuje je do właściwego serwera. W analogiczny sposób są przesyłane odpowiedzi serwera. Na podobnej zasadzie działa m.in. Oracle WebCache.
Do zbudowania serwisu HTTP typu reverse proxy z filtrowaniem mod_security najlepiej wykorzystać Apache 2.x. W standardowej konfiguracji Apache należy wprowadzić następujące zmiany:

1. Załadować mod_security i moduły niezbędne do obsługi funkcjonalności reverse_proxy:

   ```
   LoadModule proxy_module modules/mod_proxy.so
   LoadModule proxy_http_module modules/mod_proxy_http.so
   LoadModule security_module modules/mod_security.dll
   ```

2. Wyłączyć obsługę zleceń typu proxy.

   ```
   ProxyRequests Off
   ```

   Zlecenia typu proxy są potrzebne do realizacji funkcjonalności typowego proxy HTTP a nie funkcjonalności *reverse proxy*. Jeślibyśmy nie wyłączyli obsługi tych zleceń, to nasz firewall aplikacyjny mógłby być wykorzystywany jako *proxy* przez intruzów, do ukrywania swojej tożsamości przy atakach przy użyciu protokołu HTTP.

3. Spowodować przekierowanie wszystkich zleceń HTTP do chronionego serwera (analogicznie dla odpowiedzi serwera):

   ```
   ProxyPass / http://10.1.1.100/
   ProxyPassReverse / http://10.0.0.100/
   ```

4. Skonfigurować mod_security

Naturalnie serwer reverse proxy powinien być dostępny pod adresem IP (i na porcie tcp) na którym chcemy udostępniać chronioną aplikację.

Wydajność i obciążenie serwera

Mod_security nie obciąża zbytnio serwera i nie wprowadza dużych opóźnień w nadzorowanym ruchu HTTP. Według autorów oprogramowania - prędkość przetwarzania zleceń HTTP spada po dodaniu mod_security maksymalnie o 10%. Prędkość przetwarzania można znacznie zwiększyć włączając filtrowanie wyłącznie dla stron generowanych dynamicznie.

```
SecFilter DynamicOnly
```

Pojedyncze zlecenie HTTP GET/POST powoduje wiele wywołań elementów statycznych takich jak grafika, arkusze stylów (CSS), biblioteki JavaScript itd. Włączenie filtru w powyższy sposób powoduje, że mod_security będzie nadzorował tylko i wyłącznie strony generowane dynamicznie, ignorując elementy statyczne. Wiąże się z tym pewna pułapka: Mod_security traktuje jako dynamiczne tylko te elementy, które są generowane przez odpowiedni „*handler*" skojarzony w konfiguracji Apache z typem plików, za pomocą składni *AddHandler*. W konfiguracji Apache, wskazanie że dany typ pliku ma być generowany dynamicznie może się odbywać również za pomocą typów MIME. W takim wypadku mod_security potraktowałby te pliki jako zawartość statyczną. Jeśli chcemy skorzystać z powyższego sposobu zwiększania wydajności, to wszystkie tego typu skojarzenia należy zmienić na skojarzenia typu *AddHandler*. Przykładowo – skojarzenie:

```
AddType application/x-httpd-php .php
```

Należy zmienić na:

```
AddHandler application/x-httpd-php .php
```

W instalacjach gdzie kluczowa jest maksymalna wydajność, należy również unikać trybu „audit", w którym do osobnego logu są zapisywane całe zlecenia, które wywołały reakcję mod_security. Powoduje to obniżenie wydajności ze względu na dodatkowe operacje I/O.

Na zakończenie

Twórcy mod_security pracują w tej chwili nad wersją mod_security/Java, która będzie integrować się z serwerami HTTP napisanymi bezpośrednio w Javie. Ciekawą opcją, która jest zapowiadana w kolejnej wersji będzie również możliwość kontrolowania plików ładowanych do serwera, łącznie z możliwością uruchamiania zewnętrznych programów

sprawdzających pliki. Może to być bardzo przydatne w przypadku serwerów HTTP w aplikacjach korzystających z WebDAV.
Na koniec warto zaznaczyć, że również autorzy mod_security nie uniknęli błędów programistycznych. W lutym 2004 odkryto możliwość ataku metodą buffer-overflow na mod_security. Tak więc – paradoksalnie – mod_security, który jest oprogramowaniem zwiększającym bezpieczeństwo, mógł stać się boczną furtką do systemu. Na tym przykładzie jasno widać, że stopień skomplikowania systemu, przeważnie podnosi stopień ryzyka. Dlatego zawsze należy pamiętać o jednej z podstawowych zasad zabezpieczania systemów – zasadzie stosowania wielu zabezpieczeń, na wielu warstwach. Przykładowo – dobre wdrożenie mod_security powinno polegać na uruchomieniu Apache z minimalnymi przywilejami w systemie, w okrojonym środowisku (chroot, jail). Jeśli zachowamy wszystkie środki ostrożności, to ryzyko zostanie znacznie ograniczone.
Sam autor mod_security podkreśla, że jego oprogramowanie nie jest żadnym panaceum na wszystkie niebezpieczeństwa związane z aplikacjami internetowymi. Jest to kolejna broń w arsenale jakim dysponuje administrator bezpieczeństwa. Od niego zależy jak zostanie ona wykorzystana. Mod_security to jedynie „silnik” pozwalający na inspekcję ruchu. W jaki sposób ta inspekcja będzie realizowana zależy od zestawu reguł w jakie wyposażymy mod_security.
Podsumowując – mod_security wydaje się być doskonałym narzędziem, zwłaszcza dla administratorów serwerów i administratorów bezpieczeństwa, którzy muszą traktować aplikacje udostępniane przez ich serwery jako „black box”. Nie mogą ingerować w kod aplikacji, a jednak chcieliby ustrzec się przed zagrożeniami jakie może wprowadzać źle napisany kod lub zastosować dodatkowy środek ochronny.
Stosując analogię to tradycyjnych firewalli: uzasadnienie dla stosowania mod_security pomimo tego, że kod aplikacji jest pisany przy zachowaniu wszelkich zasad bezpiecznego programowania, jest takie jak uzasadnienie stosowania firewalla, pomio tego że wszystkie serwery i stacje robocze są skonfigurowane w sposób bezpieczny. Nikogo nie trzeba dziś przekonywać że stosowanie firewalli jest koniecznością. Podobny rozwój wróżę technologii filtrów aplikacyjnych, takich jak mod_security. Zwłaszcza w dobie ogromnego rozwoju aplikacji bazujących na technologiach webowych.