アプライアンス型 UTM とホスト型 WAF の組み合わせの効果について

宇田川 暢

東海国立大学機構 情報環境部

udagawa@icts.nagoya-u.ac.jp

On the Effectiveness of Combining Appliance-Based UTM and Host-Based WAF in Operational Environments

UDAGAWA Mitsuru

Information System Operations Division, Tokai National Higher Education and Research System

概要

問い合わせ管理用 Web システムにおいて発生した情報セキュリティインシデントの後、システムの再開に伴ってセキュリティを確保するために UTM と WAF による保護を行う事となった。そのシステム運用について報告する。

1 はじめに

筆者が担当する問い合わせ管理用 Web システムが動作する Web サーバにおいて、2022 年に発生した情報セキュリティインシデントをきっかけに UTM および WAF による保護を行うこととなった。なお、当該情報セキュリティインシデントの詳細については割愛するが、ブラインド SQL インジェクションによる DB に保存された個人情報の漏洩であった。

サービスの再開にあたって、情報セキュリティインシデントへの対応として Web システムを構成の脆弱性の修正のほか、未知の脆弱性を突いた攻撃への対策として、UTM および WAF によるセキュリティ対策を導入することとなった。

その後、3 年程度の運用を経て 2025 年度中に SaaS のヘルプデスクシステムへの切り替えを行うこととなった。そのため、本態勢での運用について振り返りを行うこととした。

2 UTM によるセキュリティ保護

2.1 UTM 装置の機能

UTM（Unified Threat Management）装置は一般的に次世代ファイアウォールと呼ばれ、事前に定義された単純なルールベースの通信制御だけでなく、通信先 IP アドレスやホスト名のレピュテーション情報による評価、DPI（Deep Packet Inspection）と呼ばれる通信内容のチェック、サンドボックスによる悪性ファイルの検知といった高度なセキュリティ機能を備えている。DPI については HTTP といったアプリケーションレイヤだけでなく、ネットワークレイヤでの防御にも対応している。

2.2 フル SSL インスペクション

通常、TLS により暗号化された通信内容について、通信経路途中にある機器がその暗号化された内容をチェックすることは不可能である。例えば現在 HTTPS 通信をチェックしたい場合、SNI（Server Name Indication）により暗号化されてない宛先のホスト名（およびその IP アドレス）程度の情報しか得られず、リクエスト URI 全体やリクエストヘッダ、リクエストボディといったものを確認することは出来ない。

しかしながら、UTM では DPI の一部として暗号化された通信内容の復号および再暗号化により、通信内容のチェックを行うことが可能となる機能を有している。この機能の利用に必要な設定は通信が開始される方向によって異なるが、外部から内部の Web サーバへの通信を対象にする場合、UTM に当該 Web サーバ用の有効なサーバ証明書およびその私有鍵を設定することになる。従って、導入には当該 Web サーバ管理者の協力または当該 Web サーバ管理者に成り代わってサーバ証明書を取得出来る権限が必要となる。

2.3 UTM による保護の実施

本学では全学ネットワークを保護する UTM として、対外接続装置との間に Fortinet 社の FortiGate が導入されている。この UTM でフル SSL インスペクションを実施することとなった。

この FortiGate の UTM 機能により、不正なリクエストをブロックする WAF 相当の機能を用いて、Web サーバを不正なリクエストによる攻撃からの保護を行うこととなった。

3 WAF の導入

3.1 WAF の機能

WAF（Web Application Firewall）であり、HTTP(S)のアプリケーションレイヤで動作するセキュリティ機能となる。具体的には HTTP のプロトコル違反となるような不正なリクエストや、通常の Web ブラウザのクライアント環境ではあり得ないような異常なリクエスト、Web サーバおよびバックエンドのプログラムでの不正な処理を目的とした不正なリクエストを検出し、必要に応じてリクエスト自体をブロックすることにより、不正なリクエストによる Web サーバの誤動作を防ぐフィルタである。また、リクエストだけで無くレスポンスも対象とできるものがあり、その場合は特定のパターンの文字列出力を監視することで、情報漏洩を防ぐ事が可能になる。

実際にどのようなリクエストを不正と評価し、ブロックするかは WAF ルールによって規定される。WAF ルールによっては過検知（False Positive）の恐れもあり、過検知の場合は正常なリクエストが誤ってブロックされるため、Web サイトの利用者に対して不利益となる。逆に WAF ルールが十分に考慮されていない場合、検知漏れ（False Negative）を発生させ、場合によっては攻撃が成立し、不正なリクエストを処理させてしまうことになる。

3.2 WAF の分類

WAF はその提供形態によりクラウド型、アプライアンス型、ホスト型に分類することができる。

クラウド型は Cloudflare WAF や Scutum のようにクライアントからのリクエストがクラウド WAF に送信され、クラウド WAF から実際のサーバに無害化されたリクエストが送られる。基本的にマネージドサービスとなっており、新たな脅威に対応する WAF ルールの追加などが速やかに行われるという特徴もある。

アプライアンス型は FortiWeb、F5 BIG-IP Advanced WAF といった製品で、組織のネットワーク上に設置してサーバに対するリクエストを防御する。後述するホスト型と異なり、Web UI を利用して設定することができ、WAF ルールのカスタマイズなどにも対応している。また、一台のアプライアンスで複数の Web サーバを保護することもできる。

ホスト型は ModSecurity、OWASP Coraza WAF のようにサーバに導入して、直接又はプロキシとして利用する。商用製品と OSS のどちらも存在しているが、非常に柔軟な設定と運用が可能となっている。一方、導入や設定変更についての技術的な敷居は最も高いと思われる。

3.3 WAF による保護の実施

筆者が実環境での WAF の運用経験を持っていないこと、問い合わせ対応システムという対象 Web サーバの性質上、どのような内容のリクエストが送信されてくるか不明確であることから、情報を得やすく設定の自由度も高いホスト型 WAF を利用することとした。また、以前に評価目的で利用したことがあった ModSecurity を利用することとした。

4 セキュリティ対策の導入

4.1 UTM での設定

UTM への設定については、ネットワーク担当にフル SSL インスペクション（FortiGate では deep-inspection と呼称）の設定依頼をし、サーバ証明書および私有鍵の提供を行った。

4.2 ModSecurity の設定

ModSecurity を導入しただけではほぼ役に立たず、WAF ルールを別途設定することが必須となる。再公開用の Web サーバの OS が CentOS 7 であったことから、EPEL リポジトリから利用可能な OWASP CRS[1]（Core Rule Set）を導入した。OWASP CRS は OWASP が作成している WAF 用ルールセットであり、SQL インジェクションをはじめとした多様な Web アプリケーションへの攻撃に対する防御ルールを定義し提供している。

過検知による利用者の利便性悪化を考慮し、事前に想定しうるリクエストのパターンをテストし、必要に応じて WAF ルールの無効化などを行った。また、万一過検知になった際に大きな問題とならないように、WAF でブロックされた際のエラーメッセージに問い合わせ先メールアドレスを記載しておき、問い合わせの送信が上手くいかない場合はメールで連絡するように記載しておいた。

[1] https://coreruleset.org/

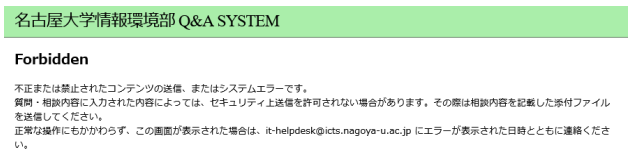

図 1　WAF ブロック時の画面表示

4.3　その他のセキュリティ対策

WAF の導入によりリクエストの処理時間の増加を原因とした DoS 攻撃の防御、および、個人情報漏洩の原因となったブラインド SQL インジェクション攻撃の検知漏れに対応することを目的とし、短時間に大量のリクエストを送信してくる接続元 IP アドレスに対して、アクセス制限を行うこととした。この制限の実現には mod_dosdetector[2] を利用している。同一 IP アドレスから短時間に多数のアクセスがあると一時的に 429 Too Many Requests のレスポンスを返し、利用者の Web ブラウザ画面には図 2 の画面が表示される。この状態を無視してリクエストが続くようであれば fail2ban[3] によりネットワークレベルでリクエストを長時間ブロックするような設定となっている。

DoS 対策は ModSecurity および OWASP CRS においても設定可能ではあったが、細かい制御を行う目的から別モジュールでの対応とした。

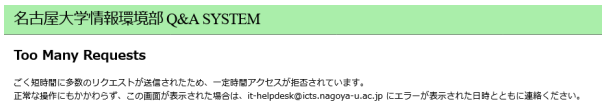

図 2　DoS 検知時の画面表示

5　ログの監視による過検知への対応

運用開始後すぐに事前に想定していた以上に過検知が発生していることがログから判明し、ModSecurity の Audit ログを毎日チェックすることになった。しかしながら、ブロックされたリクエストが真陽性（True Positive）なのか過検知なのかを判断することが困難であり、また複数のリクエストを総合して評価することも作業の難易度に拍車をかけており、多大な労力を割くこととなっていた。

この問題について、Audit ログを定期的に DB にインポートし、Abuse IPDB[4] の IP レピュテーション情報を突き合わせて Web UI 上で確認出来るシステム「WAF ログアナライザー」を作成した。この仕組みにより、毎日 1 時間程度かかっていた Audit ログの確認作業が数分で完了できるようになった。図 3 および図 4 の状況は WAF でブロックされたものではなく DoS として記録されたものではあるが、一連の検知結果についてセキュリティスキャナによるリクエストであると容易に判別できる。

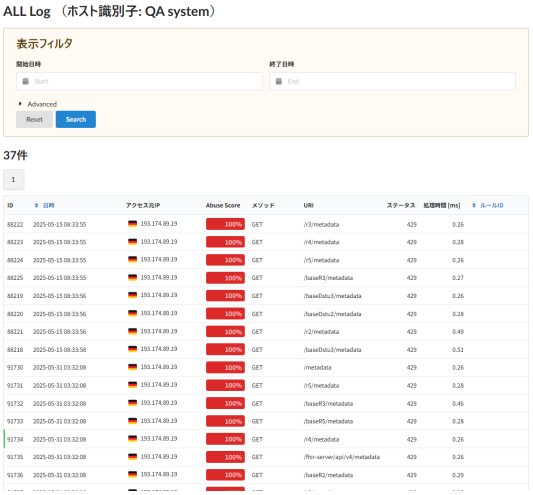

図 3　WAF ログアナライザーの画面

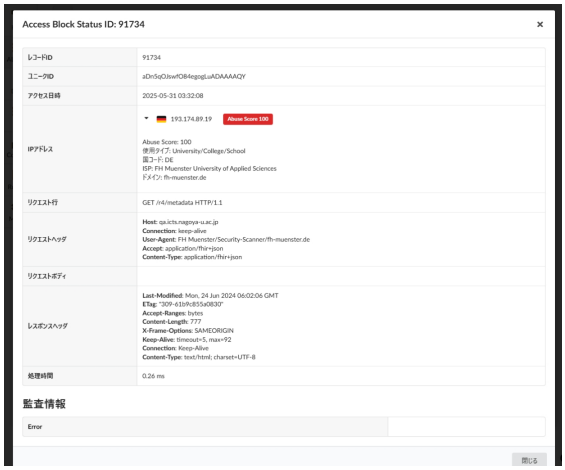

図 4　Audit ログの詳細表示

この確認作業により、過検知であると判断した場合には、必要に応じて WAF ルールの緩和や無効化といった対応を行った。例えば図 5 では問い合わせ文中の”ssh”を含む文字列を不正なコマンド実行の試みであると検知している。しかし、システムの仕様上、この入力値が直接シェルコマンドとして実行される危険性は無いため、過検知と

[2] https://github.com/stanaka/mod_dosdetector
[3] https://github.com/fail2ban/fail2ban
[4] https://www.abuseipdb.com/

なる。このように、過検知であるかどうかの判定には、Web サーバ上で動作する各種システムに対する知見が必要となり、WAF の導入や運用を困難にする要素となっている。

図 5　過検知の例（一部文字を加工）

過検知自体は何度も発生したが、事前にエラーメッセージを工夫しておいたためか、問い合わせや苦情はあっても大きな問題とはならなかった。しかし、ログのインポートが完了していないタイミングでの問い合わせが時々発生し、手動でログのインポート作業を行う必要があった。

6　OS 更新による対応

運用を再開して 2 年程度で CentOS 7 の EoL（End of Life）となったため、Rocky Linux 8 への移行を行うこととなった。当初から Rocky Linux 8 などのより長期のサポートが提供されるバージョンを利用しなかった理由として、この頃既に短期間での SaaS への移行を予定していたためである。

OWASP CRS のバージョンは CentOS 7 では 2.2.9 だったが、Rocky Linux 8 では 3.3.4 とメジャーバージョンが変更となっている。その際に大幅に構成が見直されており、WAF ルールの ID なども全く異なるものとなってしまっていた。これにより、以前の WAF ルールの例外設定や WAF ルールの上書きによる動作変更についても再度確認と調整が必要となった。

また、このタイミングで WAF ログアナライザーについても見直しを行い、ModSecurity の Audit ログの出力形式をシリアルからコンカレントに変更し、Audit ログがリクエスト毎に個別のファイルで出力されるようにした。この Audit ログを ModSecurity Log Collector を使用してリアルタイムに WAF ログアナライザーに取り込ませることで、ModSecurity による検知結果を即座に確認出来るように改修をおこなった。

7　ログの分析

7.1　ログ分析の対象と目的

OS 更新と WAF ログアナライザーの改修が完了した後の 2024 年 9 月 1 日から 2025 年 6 月 30 日までの UTM ログと WAF ログについて分析を行った。これは UTM と WAF の両方を設置することが、サイバー攻撃への防御にどの程度有用かであることを明らかとすることを目的としている。

7.2　UTM ログの分析

同期間中の UTM のトラフィックログには 3,852,282 件が記録されていた。また、UTM のイベントログは 13,369 件であった。UTM で攻撃を検知し、ブロックしたイベントログに記録されていた攻撃種別は 344 パターンであったが、そのうち件数上位 20 件を表 1 として示す。最も件数が多かったのは CVE-2021-44228 の識別子を持つ Apache Log4j の脆弱性 [5]を利用した攻撃となっていた。その他も SQL インジェクションやパストラバーサル、SSRF（Server Side Request Forgery）といった Web サイトへの攻撃が並んでいるが、18 位に位置している OpenSSL に存在し Heartbleed と名付けられた脆弱性 [6]（CVE-2014-0160）への攻撃は Web サイトそのものを対象としたものではない。UTM ではこのように多様な攻撃を検知して防ぐことが可能となる。誤解が無いように念のため言及しておくが、本 Web サーバにこれらの脆弱性が存在していたわけでは無く、UTM が攻撃パターンとして検知し、ブロックしている。

なお、この UTM は学外とのネットワーク境界に設置されているため、学内ネットワークからのアクセスについては基本的に処理対象外となる。

表 1　UTM によりブロックした攻撃

順位	攻撃パターン	件数
1	Apache.Log4j.Error.Log.Remote.Code.Execution	2,578
2	Apache.HTTP.Server.cgi-bin.Path.Traversal	2,068
3	HTTP.URI.SQL.Injection	1,068
4	PHPUnit.Eval-stdin.PHP.Remote.Code.Execution	638
5	PHP.CGI.Argument.Injection	572
6	Generic.XXE.Detection	458
7	TP-Link.Archer.AX21.luci.stok.Command.Injection	438
8	Spring.Boot.Actuator.Unauthorized.Access	438
9	Web.Server.Password.File.Access	352
10	Multiple.Routers.GPON.formLogin.Remote.Command.Injection	316
11	AndroxGh0st.Malware	298
12	Cross.Site.Scripting	297
13	Apache.HTTP.Server.mod_proxy.SSRF	274
14	SystemBC.Botnet	225
15	Remote.CMD.Shell	196
16	ALFA.TEaM.Web.Shell	183
17	Bash.Function.Definitions.Remote.Code.Execution	156
18	OpenSSL.Heartbleed.Attack	132
19	MS.Exchange.Server.Autodiscover.Remote.Code.Execution	128
20	Mirai.Botnet	100
21	MS.Exchange.Server.ProxyRequestHandler.Remote.Code.Execution	94
22	Generic.Path.Traversal.Detection	91
23	Bladabindi.Botnet	85
24	Gh0st.Rat.Botnet	71
25	D-Link.Devices.HNAP.SOAPAction-Header.Command.Execution	59
26	Apache.Struts.2.OGNL.Script.Injection	57

[5] https://www.jpcert.or.jp/at/2021/at210050.html

7.3 WAF ログの分析

まず、WAF でのリクエストを処理することで、どの程度パフォーマンスに影響するかを確認した。

表 2 はリクエストに対する HTTP レスポンスステータスコード（以下、ステータスコード）別の記録件数と WAF での処理時間である。WAF でリクエスト内容に何らかの問題があるとされたものの、リクエストのブロックには至らなかったものがステータスコード 200 として返され、WAF でブロックされたリクエストはステータスコード 403 となっている。この結果を見るとほとんどのリクエストが数ミリ秒以下で処理されていることが分かる。主観ではあるが本 Web サーバではこの程度の処理時間は問題とはならず、利用者に違和感を覚えさせることもない。

興味深いのはリクエストタイムアウトを示すステータスコード 408 の多さである。リクエストの途中で UTM により通信がブロックされた場合、Web サーバ側からはリクエストが途中で届かなくなったように見えるため、このようにログに記録される。運用開始当初は原因不明の不具合ではないかと混乱することがあった。

表 2　ステータスコード別の件数と処理時間

		処理時間［マイクロ秒］						
ステータスコード	件数	最大	最小	平均	25%	50%	75%	95%
200	20	5,087	478	2,407	2,001	4,571	4,613	4,897
301	5	592	518	538	525	525	530	579
400	29	1,891	472	1,092	875	1,228	1,381	1,703
403	2,275	83,267	461	2,300	483	502	521	565
404	11	582	479	524	540	554	568	579
405	35	604	432	502	478	505	527	557
408	1,167	1,025	279	522	474	499	530	800
419	14	8,592	1,950	4,328	2,188	4,717	5,611	6,818
429	27,892	10,275	255	577	393	458	519	1,388
500	11	3,173	390	1,232	517	570	1,866	3,107

表 3 は WAF ルール ID ごとの Audit ログの記録件数であるが、一つのリクエストに対して複数のログが記録されることに注意が必要となっている。また、特殊なルールが存在しており、例えばルール ID 949110 は他の WAF ルールによりスコアを評価し、ブロックするかどうかの判定を行い、ルール ID 980130 は Audit ログに評価の詳細を記録するためのものであり、リクエスト内容そのものの評価は行っていない。各 WAF ルール ID がどのような意味を持っているかは OWASP CRS のルール定義 [7]を参照頂きたい。

[6] https://www.jpcert.or.jp/at/2014/at140013.html
[7] https://github.com/coreruleset/coreruleset/tree/v3.3.4/rules

なお、ルール ID が該当無しになっているものについては、WAF ルールには該当しなかったものの、何らかの問題によりステータスコード 200 となり、Audit ログに記録されたリクエストである。このうち、ステータスコード 429 は 27,892 件記録されており、DoS 対策がいかに有用であるかを示していると考える。実際には有効になっている WAF ルールは全部で 150 個程度だが、攻撃を検知しログに記録されることがなかった WAF ルールについては掲載を省略している。

UTM での検知をすり抜けて、WAF によりブロックされたリクエストが複数存在していることから、UTM だけでは網羅的に防御することができないことが見て取れる。

表 3　WAF ルール ID とステータスコード毎のログ記録件数

	ステータスコード										
ルールID	200	301	400	403	404	405	408	419	429	500	合計
200002			1								1
911100				4							4
913100				21							21
920120				4							4
920160			4								4
920170				4							4
920210	9										9
920340	9										9
920420				1							1
920470					2		2				4
920600	2	5		175	9		15				206
921110				10							10
921120				15							15
921150				4							4
930100				42							42
930110				42							42
930120				26							26
931100				4							4
932105				14							14
932110				13							13
932140				7							7
932150				80							80
932160				7							7
933100				3							3
933110				3							3
933120				3							3
933140				2							2
933160				18							18
933170				1							1
933180				6							6
933210				8							8
941100				2							2
941120				1							1
941160				1							1
941170				1							1
941180				2							2
941210				1							1
942100				246							246
942160				4							4
942170				5							5
942190				2							2
942360				8							8
944100				9							9
944110				10							10
944130				20							20
949110				700							700
950140				9							9
951120				2							2
959100				11							11
980130				700							700
980140				11							11
該当なし			24	13		35	1,150	14	27,892	11	29,139
合計	20	5	29	2,275	11	35	1,167	14	27,892	11	31,459

運用においていくつかの WAF ルールを無効化しているため、その影響を考慮する必要がある。

削除または更新した WAF ルールについては、表 4 のとおりとなる。削除している WAF ルールについて検知記録されているものもあるが、これは運用途中で設定を変更したため、削除前の件数となる。

WAF ルールの更新については特定のパラメータを除外する設定となっており、例えば「リクエストヘッダのうち、Cookie を検知対象から除外」や「問い合わせ本文はシェルコマンド実行攻撃の検知対象外」といった変更を指定した WAF ルールに対して適用し、過検知を回避している。

表 4　変更した WAF ルール

ルールID	メッセージ	件数	削除	更新
920450	HTTP header is restricted by policy (%{MATCHED_VAR})	0	✓	
921110	HTTP Request Smuggling Attack	10		✓
921130	HTTP Response Splitting Attack	0	✓	
930100	Path Traversal Attack (/../)	42		✓
930110	Path Traversal Attack (/../)	42		✓
930120	OS File Access Attempt	26		✓
932100	Remote Command Execution: Unix Command Injection	0	✓	
932105	Remote Command Execution: Unix Command Injection	14		✓
932110	Remote Command Execution: Windows Command Injection	13		✓
932115	Remote Command Execution: Windows Command Injection	0	✓	
932130	Remote Command Execution: Unix Shell Expression Found	0	✓	
932150	Remote Command Execution: Direct Unix Command Execution	80		✓
932160	Remote Command Execution: Unix Shell Code Found	7		✓
933160	PHP Injection Attack: High-Risk PHP Function Call Found	18		✓
933180	PHP Injection Attack: Variable Function Call Found	6		✓
933210	PHP Injection Attack: Variable Function Call Found	8		✓
942100	SQL Injection Attack Detected via libinjection	246		✓
943110	Possible Session Fixation Attack: SessionID Parameter Name with Off-Domain Referer	0	✓	
950140	CGI source code leakage	9	✓	
953110	PHP source code leakage	0		✓

前述のとおり、本 Web サーバ上で動作するシステムは問い合わせシステムであるため、利用者から想定していない入力が不正な意図無く入力される場合がある。そのため、通常の Web サイトでは有効にしておいても問題ない WAF ルールについても、無効化するなどの対策の必要があった。

8　まとめ

本稿では UTM と WAF を併用した環境で、それぞれでどのような脅威が検出されるかを集計し確認した。環境構築時は UTM のフル SSL インスペクション機能を利用することで、WAF を導入しなくても十分な効果が得られるかと想定していた。しかし、UTM は細かな設定による柔軟性がないことから、過検知の恐れがある検知ルールは導入しづらいと考えられ、OWASP CRS と比較して積極的な防御がされていないように見える。

しかしながら、ネットワークレイヤでの防御やサブスクリプションによる速やかなパターン更新による新たな脅威への対応といった UTM の機能は魅力的であり、今回評価した OSS のホスト型 WAF の運用においては追加で利用することは有効であると考える。費用や運用負荷の問題はあるが、UTM と WAF の長所をそれぞれ生かしながら、併用して活用することがセキュリティ対策として望ましいと考える。

また、本稿では触れていないが、検知漏れに対する考慮も必要となる。ModSecurity の設定の都合上、攻撃として検知されなかったリクエストのログは保存されていないが、全てのリクエストログを取得し、検知された結果と合わせて機械学習や LLM を用いて処理することで、何らかの洞察を得ることができるように努力したい。既に Kee Hock Tan ら[1]により、LLM を利用してログから WAF ルールを作成する OSS[8]も公開されているため、技術というよりも運用態勢の問題となると考える。

想定外だったのが DoS 対策の効果であり、非常に有効だったと感じる。この DoS 対策では何らかの脆弱性スキャンツールを利用したアクセスや Web クローラなどのリクエストについてブロックに役立っていた。

参考文献

[1] Kee Hock Tan, Sean Yeoh、”WAFSmith - LLM-based Rule Management Framework to Create Rules that Simply Work”、Black Hat USA 2025、2025.

[8] https://github.com/tankeehock/wafsmith