Protocol-Level Evasion of Web Application Firewalls

Ivan Ristic

version 1.2 (30 October 2012)

www.qualys.com

Introduction

Web application firewalls (WAFs) are security tools designed to provide an independent security layer for web applications. Implemented as appliances, network sniffers, proxies, or web server modules, they analyze inbound and outbound data and detect and protect against attacks.

At some point in the last couple of years, WAFs became an accepted best practice in security. It took a lot of time and a lot of struggle, and it was not going to happen until the PCI Council gave the WAFs a serious push when they made them an integral part of PCI compliance[1]. I remember *almost* hearing a collective sigh of relief from the WAF vendors. Certainly, their daily lives suddenly became easier. Today, we have a wide deployment of WAFs, although the doubts and controversies remain.

Anyhow, I am not here today to discuss the slow adoption of WAFs. That would take too long and distract us from more interesting things. I want to discuss something else: how good are WAFs at doing the job? More specifically, I want to focus on protocol-level evasion, which is a fairly low-level aspect of WAF operation—and one that is often forgotten.

My point is that these things need to be discussed. Because of the various realities of their business existence, vendors cannot and will not build great security products alone. It can be done only in effective collaboration with the users, with vendors building the products and users keeping them in check. Unless we make it clear that technical quality is important to us, we're not going to get it.

Thus, my main reason for being here today is my desire to expose the inner workings of web application firewalls, to increase transparency, and to improve things a notch. It all comes down to the following question.

Are WAFs Any Good?

When I am asked this question, I usually reframe the discussion away from "Are WAFs any good?" to "Are the currently available *implementations* any good?" I prefer to think about what is possible, rather than about what we have today. Even at that level of discussion, there is no easy answer. I think the answer can be *yes*, but web application firewalls are complex technology, and to use them you need to have someone pretty knowledgeable in the driver's seat. Anecdotal evidence tells us that in most deployments, problems arise from mismatched expectations, lack of expertise and time, and usability issues.

It does not help that despite the continuous vocal opposition to web application firewalls, security researchers and penetration testers aren't actually ensuring that the technical weaknesses of WAFs are discussed properly. Ask yourselves this: when did you last see a good technical com-

[1] PCI Data Security Standard (PCI DSS): Application Reviews and Web Application Firewalls Clarified (2008)

parison of WAFs? Or, rather, have you *ever* seen a good technical comparison? The answer here is a clear *no*. The concept behind WAFs is sound, but we still have a long way to go.

In terms of protection, we should accept WAFs for what they really are: a method of increasing the cost of attacks, but not necessarily one that might repel every attacker. I have a feeling that WAFs could be much more useful if more organizations stopped treating them as specialized IPS tools for HTTP. There are many other use cases with tremendous potential, such as application security monitoring (ASM), attack surface reduction, application hardening, policy enforcement, and others. Unfortunately, application security budgets are not large, and these techniques require a significant time investment.

IDS, IPS, and Deep-Inspection Firewalls

Virtually all of the information here applies not only to WAFs, but also to IDS and IPS tools and to deep-inspection firewalls. In fact, bypassing network-centric security tools is bound to be easier because in general, they perform less HTTP processing (parsing) than web application firewalls.

Prior Work

There is a decent amount of papers and research notes on HTTP evasion[2], and I've made an attempt to include in this document references to the earliest works and examples known to me. Most of the available research is focused on network-level tools, such as IDS and IPS. Many of the techniques apply to web application firewalls, but WAFs typically operate on a higher network level, and that brings a different set of challenges. Some of these challenges are discussed in subsequent sections.

Even though I believe that this paper introduces new techniques (one can never be too sure, as it's entirely possible—and likely—that others have independently discovered them, exploited them, and protected against them), my main goal was to discuss protocol-level evasion in the context of web application firewalls, which is something that has not been done so far. My other big goal was to start to enumerate all evasion techniques. I have found that, even when there is good documentation on a particular topic, there is often not enough detail about the underlying root causes.

WAF Implementation Challenges

The most important selling point of a WAF is that it fully understands and processes HTTP, as well as the many subprotocols and data formats carried over HTTP. After all, you could deploy an IPS tool to inspect the traffic byte streams, but today's web vulnerabilities are too complex for lower-level devices to handle.

[2]One of the first researchers to discuss this topic was Rain Forest Puppy in 1999, with his seminal work *A look at whisker's anti-IDS tactics.*

The job is nontrivial. As any IDS vendor will tell you, passive interpretation of traffic is fraught with traps and problems. WAFs have the luxury of more available CPU cycles to perform traffic parsing and analysis, but the smarter you get in your parsing, the easier it is for the complexity to overwhelm you, and then you become a victim of evasion.

On one end is a simple *byte stream inspection*, in which you treat a TCP stream or some major part of a HTTP transaction as a series of bytes and try to match your signatures against that. This approach is powerful, as it can treat any data (no protocol parsing is necessary), but it can be easily evaded. For example, think about the support for header folding in HTTP. The mechanism can be used to split a single header value across several lines. Further, simple inspection does not enable advanced features.

On the other end is smart *in-context inspection,* with full protocol parsing and evaluation of data in the appropriate context. For example, you know that a piece of data is a value of a request header, and you treat the data as such. This approach is very powerful, but very difficult to implement successfully because you have to deal with dozens of different backend implementations and their parsing quirks.

I experienced this problem firsthand, in the years of working on ModSecurity, which is a popular open source web application firewall. I started ModSecurity in 2002 and worked on it until 2009. I always wanted ModSecurity to be very smart, but every time I pushed it into that direction, I discovered that being smart is not always the best approach. What I eventually realized is that you need to be smart and dumb at the same time.

During my research on this topic, I used ModSecurity and ModSecurity Core Rule Set (a separate distribution of security rules) to test against. In the process, I discovered two previously unaddressed issues. They were disclosed to Trustwave in June 2012[3] and consequently fixed in ModSecurity 2.6.6 and ModSecurity Core Rule Set 2.2.5. The details about the problems are addressed later in this text.

Impedance Mismatch

There's a term for what I am talking about; it's called *impedance mismatch*[4]. This is a very important concept for security tools: you're interpreting the stream of data in one way, but there's a danger that whatever you are protecting is interpreting the same data differently.

I'll give you one example: years ago I was testing a passive web application firewall. Back then, passive WAFs were easy to sell because people wanted security but didn't want the risk of downtime. To configure this particular WAF to monitor a web site, you had to put it on a network somewhere, where it could see all traffic, and input a web site's hostname.

[3] ModSecurity and ModSecurity Core Rule Set Bypasses, by Ivan Ristic (2012)
[4] External Web Application Protection: Impedance Mismatch, by Ivan Ristic (2005)

Deciding whether a connection should be expected is a *decision point*. Decisions points are critical for WAFs; every time they make a decision, there's a risk that they'll do the wrong thing. And it's a very tricky problem, too, because if you make a mistake, you become completely blind to attacks.

I succeeded in bypassing this device on my first attempt; I did it by adding a single character to the request payload. Here's the request I sent:

```
GET /index.php?p=SOME_SQLI_PAYLOAD_HERE HTTP/1.0
Host: www.example.com.
User-Agent: Whatever/1.0
```

To make this request bypass the WAF, I added a single dot character to the end of the `Host` request header. Yes, really. Apparently, the WAF developers had not considered the various alternative representations of the hostname—*they implemented only what worked for them*.

There are a bunch of other things that I could have tried here. Omitting the `Host` request header or using a nonexistent hostname often works. A WAF may be configured to select sites and policies based on the hostname, whereas the backend server may always simply fall back to the default site when the hostname is not recognized. In addition, for performance reasons, WAFs may stop monitoring a connection after determining that the first request is not intended for the site, but you can often continue to submit further requests (with the correct hostname) when persistent connections are enabled.

The main lesson here is that security products must be designed to use their most restrictive policy by default and relax policy only when there is a good reason to do so. In other words, they must be designed to fail securely.

Decision Point Exploitation

I want to go back to decision points, because it's a very important concept to understand. A decision point occurs at any place in the code where the implementation logic has to branch. In the previous evasion example, the key decision point occurred when the WAF examined the hostname and determined that it did not match the site that is being protected.

Any decision point can be potentially turned into an evasion point by performing the following steps:

1. Pick a target technology stack.
2. Identify the processing decision points involved in the processing of some data.
3. Generate a number of request variations, each differing in some small detail, and designed to exercise an individual decision point.
4. Note which variations are ignored.

5. Try the same variations when a WAF is present, without any attack payload (you want to see whether the WAF will pick up the anomaly itself and not any other payload you might have in the request).
6. If the variations are not blocked and detected outright, attempt to develop an exploit.

By now you have probably figured out what the rest of this talk is going to be like. We're going to slice and dice our way through HTTP looking for important decision points, and I am going to show you how bad decisions can be used to evade detection for any attack payload. I don't want to even try to enumerate all possible techniques, in part because doing so would make a terribly long and boring talk. I am focusing on only the principles and a few selected interesting approaches that worked reliably.

However, as part of this talk, I am releasing a catalog of evasion techniques, and there the goal is to enumerate everything. In addition, I am also releasing a number of tests along with a simple tool that you can use to test these things yourselves. All these things are now part of the IronBee WAF Research Project, on GitHub., so feel free to try them out. I'd be delighted if some of you find the topic so interesting that you're compelled to join the project and contribute refinements, new tests, and tools.

Virtual Patching

Virtual patching is the process of addressing security issues in web applications without making changes to application code. For various reasons, organizations are often unable to address security issues in a timely manner (e.g., no development resources, not allowed to make modification to the source code). In such situations, virtual patching can mitigate the problem to reduce the window of opportunity for the attack. Further, because the application itself is left alone, virtual patching can be used even with closed source applications.

Virtual patching is probably one of the most loved WAF features because of its narrow focus (breakage potential is limited) and potential high value (vulnerability is mitigated).

However, the same aspect that makes virtual patches so useful (precision, and the ability to control exactly what is allowed through) also makes them prone to bypasses. A large number of decision points are required to deliver the precision, but the more decision points there are, the more opportunities for evasion there are.

Before we proceed, I want to be clear what I mean when I say "virtual patching," because there are multiple definitions of this term. Some people have low expectations for this technique; they might say that they're using virtual patching even when they do things such as *enable* blocking in the part of the application that is vulnerable (they might be deploying in monitoring-only mode elsewhere). Or they might be increasing the aggressiveness of their blocking in the vulnerable spot. These approaches, although they are no doubt useful, are not what I have in mind.

My definition is much stricter. For me, a virtual patch is what you produce when you take the time to understand the part of the application that is vulnerable and the time to understand the flaw, and in the end produce a patch in which you accept only data that you know to be valid.

This approach is also known as *whitelisting*, or a positive security model. It is powerful because you don't need to know what attacks look like; you only need to know what good data looks like. The catch is that good virtual patches require a great deal of knowledge. They are tricky to implement correctly, as you shall soon see.

Broadly speaking, virtual patches generally consist of two steps:

1. **Activation**, in which you examine the request path to determine whether the patch should be enforced. The site is vulnerable in only one location, so you need to ensure that your patch runs only there and does not interfere with the rest of your site.
2. **Inspection**, in which you examine the vulnerable parameter(s) to determine whether they are safe to allow through.

Looking at these two steps, ask yourselves:

- What if I manipulate the path so that the patch is not activated but the request is still delivered to the correct location in the backend?
- What if I manipulate parameters so that my attack payload is missed by the WAF, but the request is still normally processed by the backend?

Let's take a look.

Attacking Patch Activation

To activate a patch, the WAF needs to examine the request path and match it against the path of the virtual patch. We are assuming that the hostname/site selection was successful. We're going to use Apache and ModSecurity for examples, but the approach is the same regardless of which WAF you're dealing with.

Let's suppose that we have an application vulnerable to SQL injection. Normally, you would exploit such an application by sending something like this:

```
/myapp/admin.php?userid=1PAYLOAD
```

To address this problem, I might write a patch like this:

```
<Location /myapp/admin.php>
    # Allow only numbers in userid
    SecRule ARGS:userid "!^\d+$"
</Location>
```

ModSecurity is very good at giving you near-complete control over what is and isn't allowed. Not all WAFs are able to do this. Those that have fewer controls might be easier to evade. Those that have more controls are better in hands of an expert but also offer more room for mistakes.

Warming up with PATH_INFO and path parameters

Surprisingly, some WAFs are still missing the concept of extra path contents (`PATH_INFO`). Against such WAFs, the following simple path change works:

```
/myapp/admin.php/xyz?userid=1PAYLOAD
```

Simply by appending some random content to the path, you completely evade the virtual patch. The attack does not work against the <Location> tag in Apache because the value supplied there is treated as a prefix.

Not all web servers support `PATH_INFO`, or at least not in all situations. In such cases, a similar feature called *path parameters* (there's only a vague mention of this feature in section 3.3 of the URI RFC) may come in handy. Consider the following URL:

```
/myapp/admin.php;param=value?userid=1PAYLOAD
```

In both cases, the operation of this evasion technique is the same: we alter the path so that it is not matched correctly by the WAF but still works in the backend.

Attacking self-contained ModSecurity rules

ModSecurity patches are sometimes written to be self-contained, which makes them more useful. You simply include the virtual patches at the site level, and you don't have to worry about them messing up your Apache configuration. Take a look at the following example, which is representative of many real-life rules:

```
SecRule REQUEST_FILENAME "@streq /myapp/admin.php" \
    "chain,phase:2,deny"
SecRule ARGS:user "!^[a-zA-Z0-9]+$"
```

These lines are what is known in ModSecurity as a chain of rules. The first rule looks at the path to determine if we're in the right place. The second rule, which runs only if the first rule was successful, performs parameter inspection.

On the surface, the example chain works as expected, but sadly, it suffers from more than one problem. First, it is vulnerable to the *PATH_INFO* attack. Append anything to the path, and the rule is bypassed. The mistake here is in using the `@streq` operator, which requires a complete match.

Second, the rule does not anticipate the use of any path obfuscation techniques. For example, consider the following functionally equivalent paths:

```
/myapp//admin.php
/myapp/./admin.php
/myapp/xyz/../admin.php
```

When the *Location* tag is used, Apache handles path normalization and thus the example attacks. But with self-contained ModSecurity rules, rule writers are on their own. There's a great feature in ModSecurity to address this issue: *transformation pipelines*[5]. The idea is that before a matching operation is attempted, input data is converted from the raw representation into something that, ideally, abstracts away all the evasion issues.

In this case, all we need to do is apply the normalizePath transformation function to take care of path evasion issues for us:

```
SecRule REQUEST_FILENAME "@beginsWith /myapp/admin.php" \
    "chain,phase:2,t:none,t:normalizePath,deny"
SecRule ARGS:user "!^[a-zA-Z0-9]+$"
```

In addition, the updated version of the rule uses `@beginsWith`, which is safer (`@contains` is a good choice, too) to counter `PATH_INFO` attacks.

Variations in underlying web server platforms

Let's complicate things further. Imagine that your Apache installation is actually a reverse proxy designed to protect applications running on a Windows-based web server. Several further evasion techniques come to mind, such as these:

```
/myapp\admin.php
/myapp\ADMIN.PHP
```

These lines will actually work against either the Apache `Location` approach or the self-contained ModSecurity rule used previously. (You may not necessarily be able to execute these attacks from a browser because they sometimes normalize the request URI and in the process convert backslashes to forward slashes.) The first attack will work because Windows accepts the backslash character as a path separator. The second attack will work because the backend filesystem on Windows is not case sensitive. This attack is not Windows specific and will work on any filesystem that is not case sensitive (e.g., HFS).

To counter these new evasion techniques in Apache, we have to ditch the comfort of the default `Location` tag and use the more powerful and more difficult to use pattern-based location matching. After some trial and error, this is what I came up with:

```
<Location ~ (?i)^[\x5c/]+myapp[\x5c/]+admin\.php>
    SecRule ARGS:userid "!^\d+$"
</Location>
```

[5] ModSecurity 2 Reference Manual: Transformation Functions (2006)

To deal with the problem, we have to:

1. Resort to using the special regex-based `Location` syntax
2. Escape all meta-characters in the path (e.g., `\.`)
3. Activate lowercase pattern matching with `(?i)`
4. Treat backslashes as path separators
5. Manually handle multiple consecutive path separators

Frankly, I don't have much faith in the solution. It may or may not work, but it's too convoluted for my liking. Something that complicated is difficult to secure reliably.

ModSecurity has a much better way to handle these attacks: in the transformation pipeline, instead of `normalizePath`, which was designed for Unix platforms, you use `normalizePathWin`, which was designed for Windows platforms. In addition, we are going to convert the path to lowercase:

```
SecRule REQUEST_FILENAME "@beginsWith /myapp/admin.php" \
    "chain,phase:2,t:none,t:lowercase,t:normalizePathWin,deny"
SecRule ARGS:userid "!^[0-9]+$"
```

The first transformation converts input to lowercase. The second normalizes the path, first converting backslashes to forward slashes and then performing path normalization as `normalizePath` does.

Path parameters again

Path parameters are actually *path segment* parameters, which means that any segment of a path can have them. The following works against Tomcat:

```
/myapp;param=value/admin.php?userid=1PAYLOAD
```

And even a single semicolon is enough to break up the path:

```
/myapp;/admin.php?userid=1PAYLOAD
```

Let's try to counter this evasion in Apache:

```
<Location ~ (?i)^[\x5c/]+myapp(;[^\x5c/]*)?[\x5c/]+admin\.php(;[^\x5c/]*)?>
    SecRule ARGS:userid "!^\d+$"
</Location>
```

This approach seems to work, but can we reasonably expect anyone to write and maintain such rules?

ModSecurity does not currently have a transformation function that removes path segment parameters, which would be the ideal approach to keep virtual patches relatively simple. In the absence of such transformation function, the possible defenses are to use the example pattern

with an `@rx` operator or detect presence of invalid path segment parameters (which may not be easy, because there are legitimate uses for them, such as session management) and reject the transaction on that basis alone.

Short names Apache running on Windows

The Windows operating system supports long filenames today, but this feature was introduced about halfway through its evolution. To support legacy applications, even today, every file with a long name also has a short name associated with it. These short names are great for evasion, because they are like a passage to your application hidden from the WAF but known to the web server.

This evasion technique cannot be used against IIS, but it works well against Apache when it is running on Windows, presumably because Apache does not implement any Windows-specific countermeasures. (Thanks to Johannes Dahse, who pointed out this out to me when he was reviewing this document.)

Further problems with older IIS versions

The point at which come close to admitting defeat, at least when it comes to performing virtual patching in an *elegant* way, is when IIS 5.1 gets involved. The rich path handling and normalization features of IIS 5.1 allow many other evasion techniques:

- Overlong two- or three-byte UTF-8 characters[6] representing either `/` (`%c0%af` and `%e0%80%af`) or `\` (`%c1%9c` and `%e0%81%9c`); in fact, any overlong UTF-8 character facilitates evasion
- Best-fit mapping[7] of UTF-8; for example, U+0107 becomes `c`
- Best-fit mapping of %u-encoded characters
- Full-width mapping[8] with UTF-8 encoded characters; for example, U+FF0F becomes `/`
- Full-width mapping of %u encoding
- Terminate URL path using a URL-encoded NUL byte (`%00`)

Against both IIS 5 and IIS 6:

- Encode slashes using %u encoding

Against some very old IIS installations:

- Bypass patch activation using *Alternate Data Streams*[9] (e.g., append `::$DATA` to the path)

[6] Crypto-Gram: Security Risks of Unicode (Bruce Schneier, 2000)
[7] README.http_inspect (Snort project, 2005)
[8] GS07-01 Full-Width and Half-Width Unicode Encoding IDS/IPS/WAF Bypass Vulnerability (GamaTEAM, april 2007)
[9] CVE-1999-0278 (1999)

Fortunately, most of these issues were addressed in newer IIS releases (IIS7+). Unfortunately, WAFs are often deployed to protect legacy systems that no one knows how to upgrade, so chances are good that the problems still exist. A good WAF should be able to handle all of these problems within its default protection rules.

Attacking Parameter Verification

Even after a patch is activated, it still needs to apply correct checks to the correct parameters. In my experience, this check is generally easy to bypass with one of the techniques presented in this section.

Multiple parameters and parameter name case sensitivity

The simplest evasion technique of this type to try is to submit more than one parameter, with some containing innocent data and some containing attack payloads. A badly written defense mechanism may screen the first or the last parameter, missing the others. For example:

```
/myapp/admin.php?userid=1&userid=1PAYLOAD
```

Another similar approach is to vary the case of the parameter name in the expectation that some mechanisms may implement case-sensitive name detection.

PHP's treatment of cookies as parameters

In PHP, configuration can dictate that request parameters are extracted not only from the query string and request body, but also from cookies. This setting used to be the default, actually, but it changed at some point (probably when 5.3.0 was introduced).

Consider the following code:

```
$_REQUEST['userid']
```

and the following request:

```
GET /myapp/admin.php
Cookie: userid=1PAYLOAD
```

With a vulnerable PHP application, this code will bypass the WAF that does not treat cookies as parameters. Further, even though the Cookie specification defines cookie values as opaque, PHP will URL-decode the supplied values.

HTTP Parameter Pollution

HTTP Parameter Pollution (HPP), documented by Luca Carettoni and Stefano di Paola in 2009[10], exploits the fact that there is no standard way to handle multiple occurrences of same-name parameters. Some platforms will process the first value, some the last, and some will—depending on the code—even combine multiple parameter values into a single string. This is a problem for WAFs because they see two separate parameters, whereas the application sees only one. But which one, and what is the contents?

Assuming the following request with two same-name parameters:

```
/myapp/admin.php?userid=1&userid=2
```

the following table demonstrates the behavior of major web application platforms (much more information is available in Luca's and Stefano's presentation):

Table 1. HPP on major web server platforms

Technology	Behavior	Example
ASP	Concatenate	`userid=1,2`
PHP	Last occurrence	`userid=2`
Java	First occurrence	`userid=1`

This technique is especially useful for SQL injection because the comma that is the byproduct of concatenation in the case of ASP applications can be arranged to work as part of SQL.

Because same-name parameters are often used by applications (selecting more than one option in a list will produce them), HPP is difficult to detect without some knowledge of the application.

To counter HPP, virtual patches should be written to restrict the number of same-name parameters submitted to the application (there's an example later in this text). Assuming individual parameter values are properly validated, it is difficult to use HPP for protocol-level evasion. It's far more useful for the evasion of the signatures that attempt to detect specific issues (e.g., SQL injection).

Tricks with parameter names

Speaking of tricks, my favorite one is the transformation of parameter names done by PHP. This problem was documented long ago in the *ModSecurity Reference Manual*, and I wrote about again on my blog in 2007[11]. If you submit parameter names with funny characters, PHP will tidy up for you. For example, whitespace from the edges is removed. Whitespace and a few other characters inside parameter names are converted to underscores, which is very handy for evasion. Simply use a + character in front of parameter name and you're done.

[10] HTTP Parameter Pollution (Luca Carettoni and Stefano di Paola, 2009)
[11] PHP Peculiarities for ModSecurity Users (Ivan Ristic, 2007)

```
/myapp/admin.php?+userid=1PAYLOAD
```

This problem emphasizes the often-used (and wrong) approach in virtual patching in which you inspect only parts of the request. For example, you may remember this rule from our earlier patch:

```
SecRule ARGS:userid "!^\d+$"
```

Clearly, if the WAF does not see the userid parameter, it won't do anything. To address this problem, virtual patches should be designed to reject any unknown parameters.

Invalid URL encoding

There's an interesting behavior in the ASP platform (and potentially elsewhere) that applies here. If you supply invalid URL encoding, ASP will remove the `%` character from the string:

```
/myapp/admin.php?%userid=1PAYLOAD
```

This reminds me of a very old problem[12] from the time when many web applications were still developed in C and had their own custom URL decoders. Some applications would not check the range of characters used in the encoding and would proceed to decode payloads that are not hexadecimal numbers. For example, normally you would encode i as `%69`:

```
/myapp/admin.php?user%69d=1PAYLOAD
```

But with an incorrectly implemented decode function, `%}9` might work just the same:

```
/myapp/admin.php?user%}9d=1PAYLOAD
```

In this section, I am assuming the WAF knows how to handle the non-standard %u encoding, which was reported as an evasion technique back in 2001[13]. For a good coverage of URL encoding evasion attacks, I recommend Daniel J. Roelker's paper[14].

Writing Good Virtual Patches

Most tricks that target parameter verification can be mitigated with a few easy-to-follow rules:

1. Enumerate all parameters.
2. For each parameter, determine how many times it can appear in the request.
3. For each parameter, confirm that the value conforms to the desired format.
4. Reject requests that contain unknown parameters.

[12] URL Embedded Attacks (Gunter Ollmann, 2002)
[13] %u encoding IDS bypass vulnerability (eEye Digital Security, 2001)
[14] HTTP IDS Evasions Revisited (Daniel J. Roelker, 2003)

5. Reject requests that use invalid encoding (not as part of the patch itself, but as part of the global WAF configuration).

Here's an example of these rules using ModSecurity:

```
<Location /index.php>
    SecDefaultAction phase:2,t:none,log,deny

    # Validate parameter names
    SecRule ARGS_NAMES "!^(articleid)$" \
        "msg:'Unknown parameter: %{MATCHED_VAR_NAME}'"

    # Expecting articleid only once
    SecRule &ARGS:articleid "!@eq 1" \
        "msg:'Parameter articleid seen more than once'"

    # Validate parameter articleid
    SecRule ARGS:articleid "!^[0-9]{1,10}$" \
        "msg:'Invalid parameter articleid'"
</Location>
```

Attacking Parameter Parsing

In requests that use the `GET` method, you know that any supplied parameters are in `application/x-www-form-urlencoded` format. However, when the `POST` method is used, you have to first correctly identify the type of encoding used. As it turns out, this is a nice opportunity for evasion.

If you cannot identify the encoding, the best you can do is treat the request body as a stream of bytes, and inspect that. That approach is actually an excellent second line of defense, but it does not work very well in all cases—particularly virtual patching.

Omitting Content-Type or supplying an arbitrary value

First of all, you can try to confuse the WAF by not providing a `Content-Type` header. That approach may cause it to skip processing the body, whereas the backend application may proceed because its processing is hard-coded.

The WAF may be designed to reject transactions that do not have `Content-Type` set, in which case you may attempt to submit some random value that does not match the type of the payload. With such a `Content-Type`, the transaction will be valid, strictly speaking, yet the WAF will not be able to process it correctly. The backend application, with its hard-coded functionality, will happily handle the data.

In the most difficult case, that of a WAF rejecting unknown MIME types, the best approach is to submit `multipart/form-data` in the `Content-Type` header, along with a request body that

can be parsed as both urlencoded and multipart format. The former will be designed for the application to consume, and the latter for the WAF.

Attacks against format detection

Not all applications hard-code request body processing, but some are pretty lax in determining the correct encoding, as became apparent to me when I was reading the source code of Apache Commons FileUpload, a very popular Java library that handles `multipart/form-data` parsing. When detecting `multipart/form-data`, it will examine the `Content-Type` request header and treat any MIME type that begins with `multipart/` as `multipart/form-data`. I wonder what we can do with that?

One approach might be to send `multipart/whatever` as the MIME type, with the body in `multipart/form-data`. The application will decide that the request body does indeed contain `multipart/form-data` and process it accordingly. But what will the WAF do?

If you're lucky, your WAF will block when upon encountering an invalid or unknown MIME type. If it does not, anything you put in the request body will potentially evade evasion.

Apache Commons FileUpload is not the only library with lax MIME type detection. A quick inspection of other popular libraries reveals that they have equally bad or worse implementations.

Evading ModSecurity

By default, ModSecurity has only a small number of rules in its default configuration. The purpose of these rules is to check whether processing has been performed correctly. These rules do not check whether the supplied MIME type is valid, or known, so using `multipart/whatever` against ModSecurity will work when it is used to protect applications relying on the FileUpload library.

If you are using ModSecurity, but not the separate Core Rule Set package (see below), you need to add custom rules to your configuration to ensure that only known MIME types are allowed through. ModSecurity does not perform byte-stream inspection of unknown content, which means that it will pass through whatever it does not recognize.

It is difficult for WAFs to be secure by default because different applications use different MIME types. If you start with strict configuration, usability suffers. If you start with a lax configuration, it is the security that suffers. WAFs can't win here.

Evading ModSecurity Core Rule Set

The ModSecurity Core Rule Set (CRS), distributed separately from ModSecurity, is stricter—it allows only requests using known MIME types. However, when I examined the implementation, I found that it was flawed. This is not the same rule, but a rule that emulates the approach:

```
SecRule REQUEST_CONTENT_TYPE "!@within \
     application/x-www-form-urlencoded multipart/form-data"
```

The `@within` operator will look for a substring match, which means that any substring of the above value would work—even a single character. So now we know that we can bypass this control using invalid MIME types, which may be useful.

One such invalid MIME type is very useful in combination with Apache Commons FileUpload. If you recall, its check is lax. If we supply `multipart/` as the MIME type, the library will accept that as `multipart/form-data`. At the same time, ModSecurity will not know what to do with the payload and will allow it through without any processing.

If you are a CRS user, you should upgrade to version 2.2.5, which should address this problem.

Multipart Evasion

Multipart parsing has always been my favorite area of evasion. Not only is the specification very shallow and ambiguous, but the `multipart/form-data` encoding is often forgotten because it's needed only by the sites that need file uploads. But people tend to forget that even applications that do not use file uploads can be forced into processing request bodies in `multipart/form-data` format.

Multipart processing flow works as follows:

1. Recognize presence of `multipart/form-data` request body
2. Extract boundary
3. Process request body data
    a. Extract parts
    b. Determine part type
    c. Extract part name
    d. Extract part value

Multipart evasion in the context of HTTP has not had much research focus, but there is an interesting early work by 3APA3A focused on evasion of content filtering software[15].

Common programming mistakes

There are two ways in which you can discover what evasion techniques work. One is to read the source code of the library or framework used by the target website. This is the preferred initial approach, because you are going to expose yourself to a number of common programming mistakes and corner-cutting errors.

[15] Bypassing Content Filtering Software (3APA3A, 2002)

There are several very common types of problem:

- Partial implementations that cover only what is commonly used by major browsers, but leave the edge cases unimplemented (e.g., HTTP request header folding)
- Lack of proper parsing, relying instead on crude substring matching or regular expressions
- Not detecting invalid or ambiguous submissions
- Trying to recover from serious problems, in the name of interoperability

Let's now look at all processing steps, one at a time.

Content-Type evasion

As already discussed, the goal with this approach is to trick the WAF into treating a `multipart/form-data` MIME type as something else while the backend is operating normally. There's a number of approaches that we can try here, such as:

```
Content-Type: multipart/form-data ; boundary=0000
Content-Type: mUltiPart/ForM-dATa; boundary=0000
Content-Type: multipart/form-datax; boundary=0000
Content-Type: multipart/form-data, boundary=0000
Content-Type: multipart/form-data boundary=0000
Content-Type: multipart/whatever; boundary=0000
Content-Type: multipart/; boundary=0000
```

The final example is what bypassed the ModSecurity CRS.

Boundary evasion

The goal of this evasion technique is to get the backend application to use one value for the boundary while tricking the WAF to use another value. Once that's possible, you can craft an attack payload in such a way that it can be correctly processed no matter what boundary value is used. The end result is that the WAF misses the attack payload.

Examples include the following:

```
Content-Type: multipart/form-data; boundary =0000; boundary=1111
Content-Type: multipart/form-data; boundary=0000; boundary=1111
Content-Type: multipart/form-data; boundary=0000; BOUNDARY=1111
Content-Type: multipart/form-data; BOUNDARY=1111; boundary=0000
Content-Type: multipart/form-data; boundary=ABCD
Content-Type: multipart/form-data; boundary="0000"
Content-Type: multipart/form-data; boundary=0000'1111
```

In 2009, Stefan Esser reported[16] that the last example was effective against F5 ASM[17].

Part handling evasion

Each multipart/form-data payloads consists of one or more individual parts. Differences in how parts are handled can sometimes lead to evasion opportunities.

Data after the last part

The `multipart/form-data` format allows arbitrary data to appear before the first part, as well as after the last part. In some cases, this feature can be exploited. For example, the earlier versions of PHP used to process all parts, even those that appeared after the part marked as being last.

In 2009, Stefan Esser reported[16] that this problem could be used to bypass ModSecurity running in front of a PHP application.

Invalid part handling

In 2012, SEC Consult discovered[18] an evasion technique against ModSecurity that can be used when PHP is in the backend. As it turned out, ModSecurity would ignore an invalid multipart part whose headers were not terminated with an empty line. PHP, on the other hand, would handle such a part by ignoring the next boundary and proceeding to parse the following part.

Parameter name evasion

Each part in a multipart submission has a name. If you can get the WAF to use an incorrect name, you succeed with evasion. The tricks here are similar to those used when attacking boundary detection.

```
Content-Disposition: form-data; name="one"; name="two"
Content-Disposition: form-data; name="one"; name ="two"
```

For example, PHP will always use the last parameter value (**two** in the first example), but it's also picky when it comes to parsing, and may "miss" the value if it is not in exactly the right format (this is why the name of the field is **one** in the second example).

ModSecurity will detect this attack because it does not allow multiple same-name parameters in a `Content-Disposition` part header. Similarly, it does not allow unknown parameters to be used.

[16] Shocking News in PHP Exploitation (by Stefan Esser, 2009)
[17] BIG-IP Application Security Manager
[18] SEC Consult: ModSecurity multipart/invalid part ruleset bypass (by Bernhard Mueller, 2012)

Parameter type evasion

There are two types of parameter. Normal parameters are equivalent to those in the urlencoded format. File parameters are files. Some WAFs treat the contents of the files differently. In ModSecurity, for example, the file content is not inspected. So, if you can trick ModSecurity into believing something is a file when it is not, you can evade it.

This is what the Content-Disposition header of a file part looks like:

```
Content-Disposition: form-data; name="f"; filename="test.exe"
```

If the filename parameter is present, the part is considered to contain a file. If the filename parameter is not present, the part is considered to contain a normal parameter.

There was already a bypass in this area, which Stefan Esser discovered[16] in 2009 (that was after I had left the project, and I was not involved in the patching effort). It applied to a deployment of ModSecurity protecting a PHP application. Stefan determined that PHP supported single quotes for escaping in the `Content-Disposition` header, whereas ModSecurity supported only double quotes. ModSecurity was subsequently patched to add support for single quote–escaping, and all was well.

Earlier this year, when I started researching evasion, I decided to revisit all historic evasion issues and double-check them. The logic is simple: where you discover one unusual behavior, you may discover another. After reading the PHP source code, I understood that the original vulnerability was not in the code that parses key-value pairs but in the code that executes before that, which breaks the `Content-Disposition` header into key-value pairs:

```
Content-Disposition: form-data; name=1"2;3"4; filename=5
```

In the example, the double quotes (single quotes can be used, too) effectively shield the semicolon, which is the termination character for a key-value pair. So, as far as PHP is concerned, the semicolon is data. For everyone else, it's part of the `Content-Disposition` syntax.

Once you understand the core issue, it's clear that ModSecurity's patch was insufficient. All you need to exploit the problem is add a single nonquote character as the first character in a parameter value:

```
Content-Disposition: form-data; name=x';filename="';name=userid"
```

PHP allows you to use quote characters anywhere on the line, but ModSecurity's parser is quite strict, so the only place where the exploit can be used is in the parameter value.

Once the evasion payload is removed, this is what PHP sees:

```
name = userid
```

Although there are two name parameters, the first one is overwritten by the second one.

On the other hand, ModSecurity sees this:

```
name = x'
filename = ';name=userid
```

The evasion messes up the parameter names as observed by ModSecurity. That issue alone can interfere with any virtual patches that may be deployed. More important, the part is treated as a file and bypasses all inspection.

Future Work

This document is not a complete guide to protocol-level evasion of web application firewalls. It's only a starting point. There's a lot of work that I could not fit into a single document, manifested as a pile of notes that are still waiting to be processed. Further, a serious effort is needed to examine the behavior of the many libraries and frameworks that are used to build web applications.

Further topics include:

- HTTP message parsing
- Request line parsing
- Request header parsing
- Hostname parsing
- Cookie parsing
- Unicode issues
- Character encoding handling

Acknowledgments

Thanks to Johannes Dahse, Mario Heiderich, Krzysztof Kotowicz, Marc Stern, and Josh Zlatin for providing feedback on the draft versions of this paper.

Changes

Version 1.1 (18 July 2012)

- First official release.

Version 1.2 (30 October 2012)

- Added new SEC Consult `multipart/form-data` evasion against ModSecurity.