WAFFle: Fingerprinting Filter Rules of Web Application Firewalls

Isabell Schmitt Sebastian Schinzel
University of Erlangen-Nuremberg
Chair for IT Security Infrastructures
first.last@cs.fau.de

***Abstract*—Web Application Firewalls (WAFs) are used to detect and block attacks against vulnerable web applications. They distinguish benign requests from rogue requests using a set of filter rules. We present a new timing side channel attack that an attacker can use to remotely distinguish passed requests from requests that the WAF blocked. The attack works also for transparent WAFs that do not leave any trace in responses. The attacker can either conduct our attack directly or indirectly by using Cross Site Request Forgeries (CSRF). The latter allows the attacker to get the results of the attack while hiding his identity and to circumvent any practical brute-force prevention mechanism in the WAF. By learning which requests the WAF blocks and which it passes to the application, the attacker can craft targeted attacks that use any existing loopholes in the WAF's filter rule set. We implemented this attack in the WAFFle tool and ran tests over the Internet against ModSecurity and PHPIDS. The results show that WAFFle correctly distinguished passed requests from blocked requests in more than 95 % of all requests just by measuring a single request.**

I. Introduction

Web application security has become a crucial topic for the success—sometimes even for the survival—of many companies. Examples for critical security vulnerabilities in web applications are Cross Site Scripting (XSS), SQL Injection (SQLi), or Directory Traversal. To attack these vulnerabilities, the attacker sends rogue requests to a vulnerable web application. If the application confuses the payload of the rogue requests with commands, the attack succeeded and the attacker can read, change, or delete sensitive information from the application.

Web Application Firewalls (WAF) are mitigations for these vulnerabilities that do not aim at fixing the actual vulnerable application, but that try to detect and to prevent rogue requests. To distinguish normal requests from rogue requests, WAFs use a set of filter rules in the form of white-lists, black-lists, or a combination of both. Commonly, the WAF will pass only those requests to the application that are classified as normal requests. Requests classified as rogue are usually blocked and thus not passed on to the application. Creating filter rule sets is challenging because on the one hand if the WAF blocks some normal requests (false positive), then the application may not function any more. On the other hand, if the WAF does not block all rogue requests (false negative), then the attacker may circumvent the WAF and exploit a vulnerability in the application. Another obstacle is that rogue requests that aim at exploiting XSS vulnerabilities are different from those aiming at exploiting SQLi vulnerabilities, which indicates the complexity of a filter rule set that detects the most common attacks. Tightening a filter rule set such that all false positives and false negatives are prevented is thus hardly possible with the limited resources of realistic systems. Because there is no reason to believe that any given filter rule set is perfect, it is common to treat the active filter rule set as such as confidential. This is to prevent the attacker to spot and exploit weak spots in the rule set.

This work was supported by Deutsche Forschungsgemeinschaft (DFG) as part of SPP 1496 "Reliably Secure Software Systems".

Side channel vulnerabilities—or side channels—are unintentional and hidden communication channels that appear if the publicly observable behavior of a process correlates with sensitive information [23]. Side channel analysis was traditionally used to break implementations of cryptographic algorithms [12], [3]. On the web, side channel attacks are widely spread and a serious threat to the confidentiality of information on the web. They can be separated in *timing side channels* [8], [2], [18] and *storage side channels* [9]. Timing side channels appear if the response time of a web application correlates with confidential information. Thus, by measuring the response time, the attacker can learn confidential information. Storage side channels appear for example if protocol header data or the indentation of markup language documents correlates with confidential information.

Whereas storage side channels leak information independently of the network connection quality, timing side channels are more difficult to exploit if the network connection adds much noise in the form of random delays (*jitter*). If the variance of the jitter is large compared to the timing difference to be measured, the attacker has to apply filters to approximate the actual timing difference [5].

We present a practical timing side channel attack that allows to remotely distinguish passed and blocked requests. This allows a remote attacker to determine loopholes in the WAF's filter rules and to adjust the attack in a way that it evades the WAF. Furthermore, we extend the attack so that multiple unsuspecting web users perform the attack, thus hiding the identity of the actual attacker. The attack was implemented in the tool "WAF Fingerprinting utilizing timing side channels" (*WAFFle*). We make the following contributions:

- We describe a timing side channel attack against WAFs that directly distinguishes passed requests from blocked requests without relying on ambiguous error messages.

- We combine our timing attack with Cross Site Request Forgeries, which hides the attacker's identity and prevents the WAF from blocking the attack assuming that the attacker distributes the attack to many other users.
- We test the attack over an Internet connection against three common WAF deployment setups and show that the attack is highly practical.

The paper is structured as follows. In the following, we present related work and in Section II we explain the workings of WAFs. We explain the idea behind our attack in Section III. Section IV presents our timing attack and Section V combines the timing attack with Cross Site Request Forgeries. We conclude and discuss possible mitigations in Section VI.

Related Work

Bortz, Boneh, and Nandy [2] introduced the concept of cross-site timing attacks with which they could determine whether a user is currently logged on to a site. They measured whether the browser of the victim retrieves an item from the browser cache (which will be very fast) or whether the browser needs to download the item (which will be slow). We extend this approach by combining CSRF attacks [25] with timing attacks, in order to hide the identity of the attacker who could also perform the attack directly.

Fingerprinting on the network level is widely known and the various tools are commonly used in day-to-day penetration testing. The most famous tool is Nmap [7] which is an active network scanner that can scan large IP ranges, fingerprint the producer and version of operating systems, and learn producer and version of network services by analyzing the service banner. p0f [14] is a passive network scanner that analyses network traffic and identifies producer and version of the operating system of the sender. Both tools aim at fingerprinting network stacks but fingerprint firewall filter rules.

Firewalk [15] is a tool that fingerprints rules of packet filtering firewalls. It sends out TCP and UDP packets with a TTL that is one greater as the amount of hops from the sender to the firewall. If the packet passes the firewall, the next hop discards the packet and sends an ICMP_TIME_EXCEEDED message to the sender. Thus, this message indicates that the packet was not filtered by the firewall. Firewalk cannot be used to fingerprint application layer filtering firewalls because they create separate connections to the application, i.e. single packets are never passed from sender to the application. Samak, El-Atawy, and Al-Shaer extend this approach to intelligently choose probing packets for fingerprinting filtering rules [20].

Khakpour et al. [11] were able to distinguish three different network firewall implementations by sending TCP packets with unusual flag combinations and measuring the time it took for a firewall to process the packets. They focused on distinguishing the firewall products but did not fingerprint the active filter rules of the firewalls. The purpose of their work is similar to NMAP and p0f with the difference that they aim at fingerprinting implementations of filter engines.

WAFW00f [21] can detect if a web page is protected by a WAF and can differentiate between 22 different WAF producers. For this, it sends normal and rogue requests to the same URL and compares the responses. It assumes that differences in the responses such as different HTTP status codes denote that a WAF filters the requests. However, the tool does not distinguish between "blocked by WAF" error responses and "caused error in web application" error responses which were possibly rewritten (*cloaked*) by the WAF. Just from analyzing the responses it is therefore not possible to tell with certainty whether a request was blocked by the WAF or passed on to the web application. WAFW00F directly connects to the WAF, i.e. the WAF may learn the IP address of the attacker and block the attack. Furthermore, WAFW00F does not fingerprint the filtering rules but solely determines WAF producers.

WAF Tester [6] is a tool that fingerprints WAF filter rules by analyzing the HTTP status codes and whether the WAF drops or rejects the HTTP request on the TCP layer. It has similar assumptions to WAFW00F regarding the detection of blocked requests from different responses. For example, there is the case where a passed rogue request crashes the web application, which the tool may confuse for a blocked request. WAF Tester therefore tries to distinguish passed requests from blocked requests from certain error conditions in the responses, which is not always possible. Similar to WAFW00F, WAF Tester directly connects to the WAF, i.e. the WAF may learn the IP address of the attacker and block the attack. We show that instead of relying on error messages, measuring the response time of requests gives more reliable information on whether the request was blocked or passed by the WAF. Furthermore, we extend WAF Fingerprinting in a way that it uses cross site request forgeries, which only works with timing attacks. This has the advantage that the WAF does not learn the attacker's IP.

"Mutating exploits" and their effects on the detection of intrusion detection systems (IDS) were analyzed by Mutz and Vigna et al. [17], [27]. Both deal with ways to obfuscate malicious code in a way such that the attack is not detected by IDS but that the attack still works. For this, they generate many variations of an exploit, run them against a victim system and correlate them with the alerts produced by the IDS. Their work is related to ours because an IDS can be modeled as a firewall that only alerts administrators but does not interfere with network traffic. However, their attacker scenario allows the attacker to access the alerts of the IDS. In our scenario, the attacker is weaker because he neither needs to receive alerts, nor does he need access to the firewall's log files.

II. Web Application Firewalls

Besides blocking rogue inbound requests, WAFs are also used to "cloak" those outgoing responses that contain sensitive information such as error messages or stack traces. A securely configured WAF substitutes these error messages with a single generic error page. In this paper, we assume a cloaking WAF where the different error conditions (e.g. an error occurred in application or a rogue request was detected) are indistinguishable for an attacker that analyzes the responses.

A. Filter Rules

WAFs detect rogue requests from a set of filter rules. Although the rule languages differ from product to product, they

basically consist of regular expression and an action. The WAF executes the action if the regular expression matches a request. There are a variety of actions that common WAFs support and the following list provides an excerpt of the possible actions that ModSecurity supports [16]. For our purposes, we are interested in those actions that pass a request on to the web application and in those that block a request, i.e. that do not pass the request to the web application.

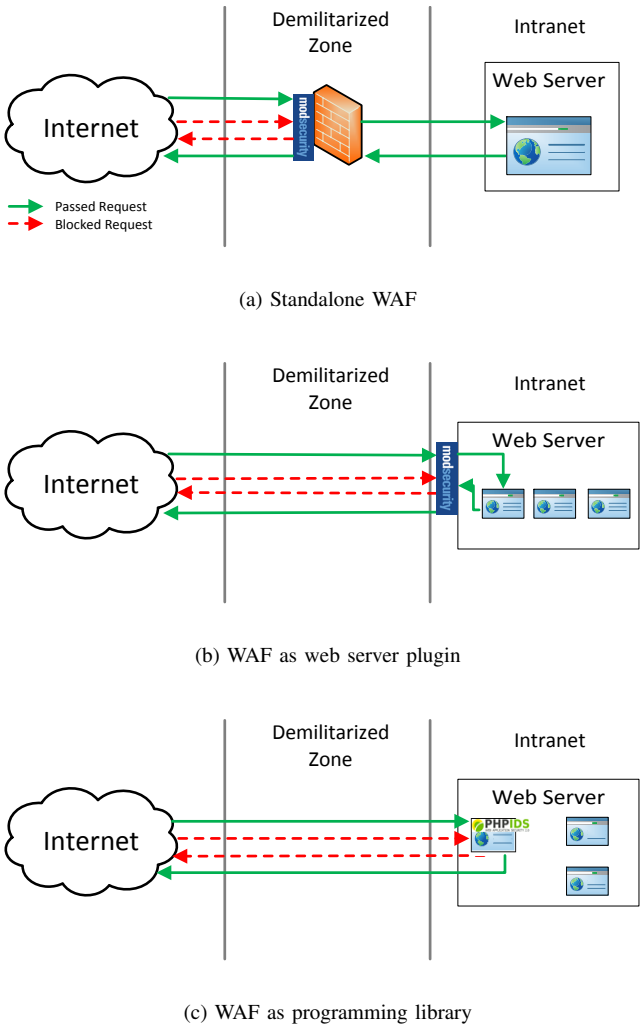
Examples for Passing Actions:

- **log** - This action causes ModSecurity to log a match in the apache error log.
- **pass** - This action is mostly used together with the *log* action if someone only wants to log a match but does not want to take further actions.
- **allow** - In contrast to the *pass* action the *allow* action will not only let a request pass a particular match but will allow it though the whole filter set. This action could for example be used to provide whitelisting for a particular IP address.

Examples for Blocking Actions:

- **deny** - This action stops further processing immediately and returns a HTTP 500 error to the client.
- **block** - This action stops further processing immediately and terminates the TCP connection of the client by sending a TCP FIN packet.

It is important to note that the default rule set of WAFs often consists of several dozen or hundred filter rules and that the regular expression of each rule can be quite elaborate. This makes common rule sets complex and difficult to audit, i.e. for the administrator, it is difficult to spot loopholes in a rule set even when he has full access to the rules.



(a) Standalone WAF

(b) WAF as web server plugin

(c) WAF as programming library

Fig. 1. Different topological deployment options for WAFs.

B. WAF Network Topologies

We consider three common ways to deploy a WAF. The first topology is to install the WAF standalone (reverse-proxy) as shown in Figure 1(a). Here, clients directly connect to the IP of the WAF. The WAF connects to the IP of the web application, passes the request, retrieves the response and passes the response to the client. WAF and web application are different hosts in this scenario. If a request is blocked, the rogue request never reaches the host that runs the web application.

The second scenario is to load the WAF as a plugin into the same web server that also serves the web application as shown in Figure 1(b). The clients connect to the web server and the web server ensures that the request is first passed to the WAF plugin and then to the actual web application. If a rogue request is blocked, the web server will never pass the request to the web server module that processes the web application.

Thirdly, there is the scenario where the WAF is directly included into the web application as a programming library as shown in Figure 1(c). Here, the client connects to the web application and the web application passes the request to the WAF library. If a rogue request is blocked, the web application will not pass the request to the actual processing logic.

C. A Timing Side Channel in WAFs

As the tools WAFW00F [21] and WAF Tester [6] exploit storage side channels, all they can possibly observe are the following three different responses.

1) *WAF error message.* The WAF responds with a unique error message (or drops or rejects the request). This either means that (a) the rogue request was blocked by the WAF or (b) that the WAF passed the request to the web application that responded with an error message and which was then cloaked by the WAF.
2) *Webapp error message.* The web application responds with an error message that is different from the WAF error message. Here it is clear that the WAF neither blocked the request, nor cloaked the web application's error message.
3) *Normal response.* A normal response with no error is observed. There are three possibilities that may cause this behavior. *(a)* The WAF removed the malicious part of the rogue request, thus passing the equivalent to a normal request to the web application. *(b)* Another option is that the WAF passed the rogue request but

the web application ignored the malicious part of the request. *(c)* Lastly, the WAF could have passed the rogue request and the malicious part was executed, but it produced no visible result. An example for this are "blind SQL Injection" attacks where an attacker can execute malicious SQL commands but cannot access the result of the command [10].

Thus, just from observing responses, one cannot distinguish passed requests from blocked requests because error messages can occur for both cases. In this paper, we introduce a timing side channel attack against WAFs that allows us to directly distinguish blocked requests from passed requests without relying on ambiguous error messages in responses. We exploit the fact that a blocked request finishes earlier than a request that is passed on to the web application as described in Section II-B. Thus, the response time should allow to distinguish passed and blocked requests.

III. General Methodology of the Timing Attack

We expect that blocked requests finish earlier than passed requests because the actual application logic that processes the request is never reached. Thus, the timing difference between passed and blocked requests equals the processing time of the application logic. The longer this processing time is, the smaller the negative effect of jitter on the measurement, the easier it is for the attacker to distinguish passed and blocked requests. Note that the attacker is free to choose those URLs with long running processes to ease the fingerprinting process. Furthermore, the attacker may combine the fingerprinting process with denial of service attacks such as "HashDos" [4], [1] to artificially increase the processing time.

A. Attack Idea

The attacker in our scenario has selected a target to attack and is now in the reconnaissance phase where he wants to find out whether a WAF protects the application and what filter rules are active in the WAF.

We assume that the WAF returns an error message immediately if a request is classified as rogue request, without passing the request to the application. In contrast, a normal request is passed on to the application. Our hypothesis is that rogue requests have a measurably shorter response time than normal requests. The attacker should thus be able to distinguish those requests that were blocked by the WAF from those that were passed on to the application.

To perform the attack, the attacker needs to guess two different requests. The first should result in a passed response and is easy to get. The second should contain maliciously looking payload that any WAF certainly blocks, e.g. the string `' OR '1'='1` which is a trivial SQLi exploit. The attacker sends these requests to the WAF and measures the response time. In the following section, we explain an efficient method to distinguish passed requests from blocked requests.

B. Analyzing the Timing Measurements

In this section, we present our notion of *possibilistic timing attacks* [24]. We split our attack into the learning phase and the attack phase as shown in Figure 2. In the learning phase, we measure the response times $T = \langle t_1, t_2, \ldots t_n \rangle$ of n passed requests and define a "blocking boundary" such that

$$t_{boundary} = min(T) - \epsilon$$

where ϵ accounts for the fact that the true minimum boundary of T may be slightly lower given more measurements.

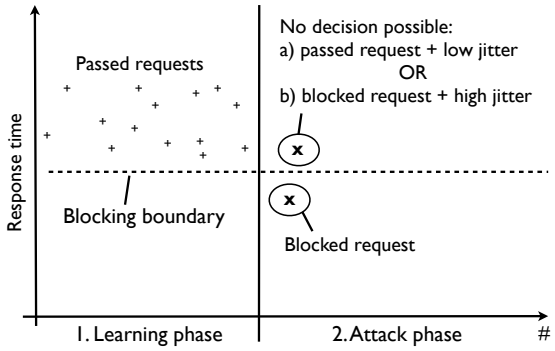


Fig. 2. Possibilistic timing analysis: Response times below the "Blocking boundary" denote blocked requests, response times above are candidates for passed requests.

In the attack phase, the attacker sends rogue requests and wants to know whether the WAF passed the request or blocked it. Any timing measurement $t < t_{boundary}$ denotes a blocked request. Any timing measurement $t \geq t_{boundary}$ is a *candidate* for a passed request. It is only a candidate because t either denotes a passed request and low jitter or it denotes a blocked request and high jitter. In order to confirm the candidate, the attacker repeats the measurement until a satisfying confidence is reached that the candidate is a passed request. This method is called "possibilistic timing analysis" because some measurements are definite and others require repetitions to confirm the result [23].

Note that $t_{boundary}$ can vary between different URL paths of the same site, and should therefore be calculated for each unique URL path. In the attacks scenarios described in the following sections, however, we used a single $t_{boundary}$ for all URL paths and got very good results with only few exceptions.

IV. Black-Box Fingerprinting of WAF Filter Rules

Now that we described the general methodology of our attack in the previous section, we constructed all three WAF network topologies described in section II-B. To test our approach, we chose the free WAF product ModSecurity [26] in version 2.5.12-1 for scenarios depicted in Figure 1(a) and 1(b). To implement the scenario of Figure 1(c), we used PHPIDS [13] in version 0.5.6, which is an intrusion detection system that scores incoming requests. High scores indicate an attack, in which case we blocked the request, emulating a WAF.

We chose phpBB as web application that the WAF protects. This web application and the WAFs were hosted at the French cloud computing provider OVH. We used a host in the network of the University of Mannheim in Germany to perform the timing attack against the WAF. This intracontinental measurement setup reflects that our attack is highly practical. Our client-side

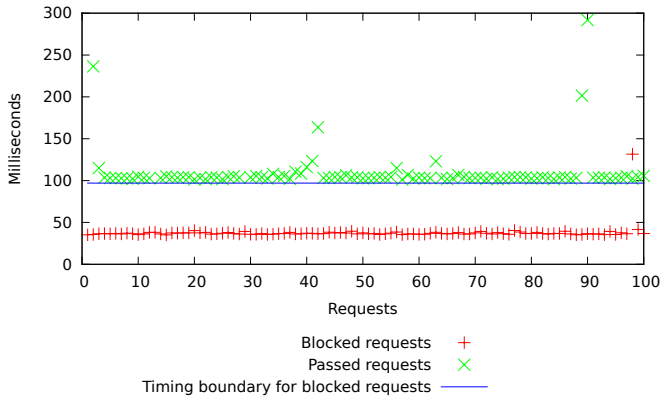


Fig. 3. Timing differences of a standalone WAF for passed responses and blocked responses

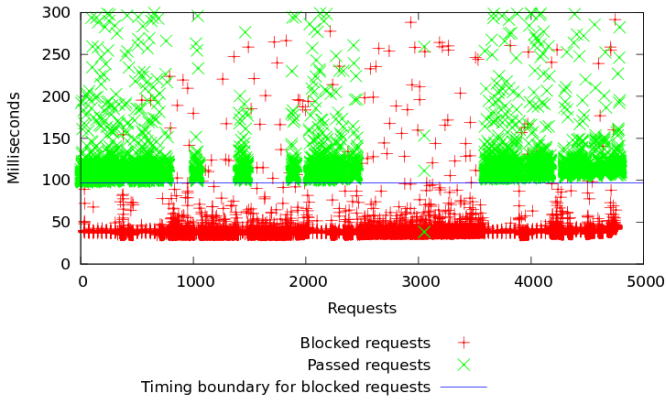


Fig. 4. Measuring the response time for each request in the standalone WAF scenario.

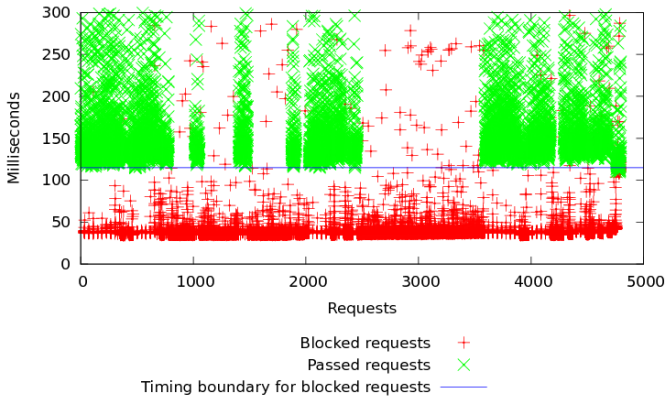


Fig. 5. Measuring the response time for each request in the WAF as Web Server Plugin scenario.

measuring computer ran an Intel Pentium 4 CPU with 3.20 GHz and the WAFs were installed with the default settings.

Our prototype implementation of the attack (WAFFle) starts by initiating the learning phase as described in section III-B, in which it determines whether a WAF exists or not. If a WAF exists it calculates the blocking boundary for blocked requests. In this simple test, we repeatedly measure the response times of passed requests and blocked requests and plot the result in Figure 3. It shows a clearly visible timing difference between passed responses and blocked responses, which confirms that a WAF filters the requests. Below are two basic examples that ModSecurity and PHPIDS will either pass or block in their particular standard configuration.

Passed request:	`GET /?p=1234567890 HTTP/1.1`
Blocked request:	`GET /?p='%20or%201=1-- HTTP/1.1`

In the next step, WAFFle crawls the web application to find all combination of URLs and parameters. It then sends the rogue payloads within the found parameters and measures the response time. If the response time is below the blocking boundary, it classifies the requests as blocked. A response time above the blocking boundary is marked as a candidate for a passed request. WAFFle then repeats the measurement to confirm the result.

A. *Direct Fingerprinting of WAF Filter Rules*

We now compile a list of malicious payloads that are commonly used to exploit vulnerabilities (e.g. from [19]) and send them to the WAF-protected web application. Our attacker ultimately aims to find a *polymorphic representation* of malicious payload that evades the WAF's active filter rules. Polymorphic representations are semantically identical but syntactically different to a malicious payload. Thus, we extend the list with polymorphic representations of the malicious payloads as shown in the following example.

Malicious payload	`' OR '1'='1`
Polymorphic representation	`' OR '2'='2`
Polymorphic representation	`' OR '1' = '1`

We sent these payloads to all URLs and all parameters of phpBB, which resulted in overall 4797 requests, and recorded the response times. To validate these response times, we configured ModSecurity and PHPIDS in our test environment to return error messages in the case of a blocked message. We recorded the status codes along with the response times and could therefore validate the results of the timing attack. For example, if WAFFle classified a particular request as blocked, we also expected an error message. Furthermore, if WAFFle classified a request as passed, we expected no error message. Otherwise, WAFFle classified a request wrongly.

Figure 4 shows the results of measuring the response times of the malicious payloads in the standalone WAF scenario. We found that already 95.2 % of all measurements correctly indicated passed or blocked requests without any measurement repetitions. Thus, we can reach perfect measurement conditions with only few measurement repetitions. The scenario where the WAF is loaded as a web server plugin yields very similar results as shown in Figure 5. We expected that the attack would perform worse in the third scenario, where the WAF is deployed as a programming library, but we were surprised to find that the attacks works similarly well as shown in Figure 6. The insight here is, that the overhead of the network connection in the standalone WAF scenario is negligible compared to the delay induced by the WAF filtering engine. In summary, our timing attack correctly detected passed and blocked requests in more than 95 % of all cases

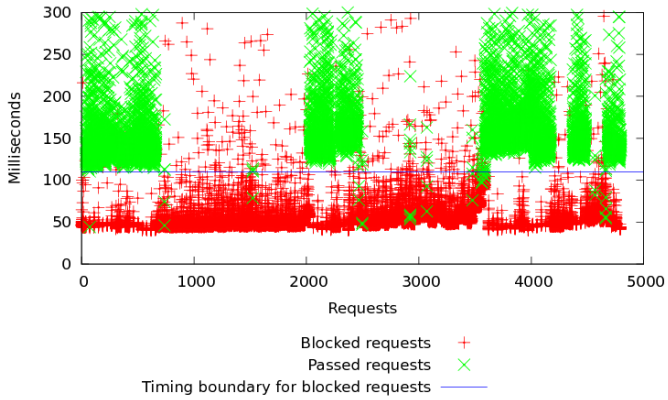


Fig. 6. Measuring the response time for each request in the WAF as Programming Library scenario.

WAF topology	Figure	Timing difference	Correct
Standalone	1(a)	62.63 ms	95.2 %
Web server plugin	1(b)	81.86 ms	95.4 %
Programming library	1(c)	48.22 ms	96.3 %

TABLE I
TIMING DIFFERENCE BETWEEN BLOCKED REQUESTS AND PASSED REQUESTS PER WAF TOPOLOGY.

as summarized in Table I.

Although this attack is very efficient because in most cases the attacker only needs a single timing measurement to distinguish passed from blocked requests, badly configured WAFs may leak this information through different error messages, because they do not cloak responses. In this case, the attacker can analyze the error messages instead of the response time. The downside of both approaches is that the attacker possibly needs to send large amounts of requests to find loopholes in a filter rule set. WAFs can detect this attack and block the attacker from finishing it. We therefore extend our tool such that it tricks unsuspecting web users to perform the actual requests, thus combining Cross Site Request Forgeries (CSRF) and timing attacks. This hides the identity of the attacker and prevents the WAF from blocking the fingerprinting attack if many users simultaneously conduct the attack.

V. CROSS-SITE FINGERPRINTING OF WAF FILTER RULES

The direct timing attack can be improved to disguise the identity of the attacker and to prevent the WAF from blocking the fingerprinting attack. For this, we combine our timing attack with a CSRF attack. Note that this is different from the Cross-Site timing attacks of Bortz, Boneh, and Nandy [2] because they gain confidential information about the users, e.g. whether the user is logged on to a site. As opposed to this, we abuse other users to learn confidential information about WAFs and thus from the server side.

As a precondition for our attack, the attacker must be able to lure web users to a web site where he can place malicious HTML and JavaScript coding (step 1 in Figure 7). This code tricks the web users' browsers to send the malicious request to the victim web application (step 2 and 3). Simultaneously, the browser measures the response time of the malicious request and sends the result back the attacker (step 4).

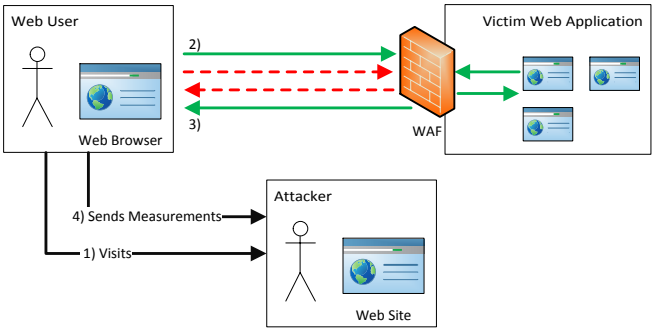


Fig. 7. Overview of the cross site timing attack.

```
1  <script>
2    var time;
3    var img = document.createElement('img');
4    img.onerror = function() {
5        var end = new Date();
6        time = end - start;
7        sendResult(time); // send result to attacker
8    }
9    img.style.display = 'none';
10   document.body.appendChild(img);
11   var start = new Date();
12   img.src = "http://domain.tld/path?" + parameter
         + "=" + exploit;
13 </script>
```

Fig. 8. Pseudo JavaScript code showing the cross site timing attack.

There are various ways in a browser to time a web request and in our tests we chose that same technique proposed by Bortz, Boneh, and Nandy [2]. In this coding shown in Figure 8, the attacker creates an image tag. Just before he copies the malicious payload to the URL of the image (line 12), he records the starting time. As the request most certainly will not return a valid image, the browser fires the *onerror* function that the attacker defined in lines 4-8. In this function, the attacker records the ending time, and sends the timing difference between starting and ending time to the attacker.

It is important to note, that this cross site attack only works reliably with the timing attack, because the Same Origin Policy [28] of web browsers does not allow reading or writing response bodies from other origins. Thus, in this cross-site scenario, it is not possible to read the error messages in responses of badly configured WAFs, which means that analyzing error messages is not an option in this cross site scenario. However, we show that it is still possible to read the response time of the request.

We implemented the cross site timing attack and ran it against the proxy WAF scenario. Figure 9 shows that also the cross site extension to WAFFle reliably distinguishes blocked and passed requests. Note that this attack can be distributed to many different web users and if each only fingerprints a few requests, the WAF cannot prevent the attack by simply blocking the IPs of the various senders.

VI. CONCLUSION

We present a new fingerprinting attack that allows to remotely distinguish requests that were blocked by the WAF or passed by the WAF. The attack extends existing tools in a way

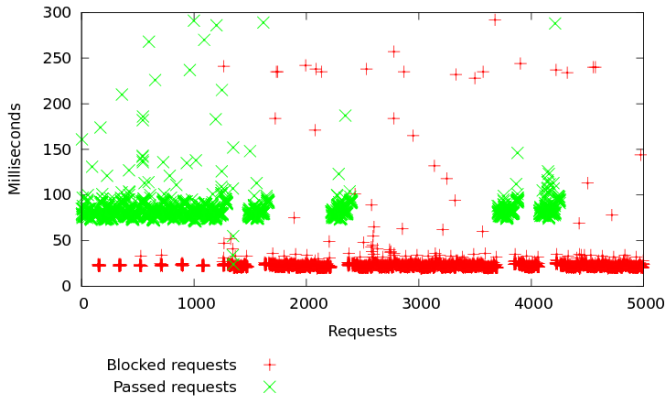


Fig. 9. Results of the cross site timing attack.

that it does not rely on error messages in the responses of the WAF or the web application, which are easy to hide if the WAF is configured securely. Instead, it distinguishes blocked from passed requests solely by analyzing the response time of the requests. This makes our attack difficult to prevent.

Furthermore, we extend the timing attack by combining it with Cross Site Request Forgeries, which hides the identity of the attacker. If this attack is spread to many users, the WAF cannot block the fingerprinting attack simply by blocking IP addresses. This allows an attacker to find loopholes in filter rules with little effort. We tested the attack over the Internet against three common WAF deployment scenarios and we argue that the attack works against all WAFs.

Preventing timing attacks in networked applications by artificially delaying responses is difficult in practice, because the security depends on how the delay is chosen. Random delays are known to be ineffective and padding to the worst case execution time is not practical. Adding a deterministic and unpredictable delay may be a solution to this [22].

Our attack highlights the importance that filter rule sets need to be carefully written and audited to prevent loopholes. Thus, the best mitigation for our fingerprinting attack is to have no loopholes in the WAF's rule set. As a consequence, the attacker may still be able to fingerprint the rules but he does not find loopholes.

References

[1] Alexander Klink and Julian Wälde. Efficient denial of service attacks on web application platforms, 2011. 28th Chaos Communication Congress http://events.ccc.de/congress/2011/Fahrplan/events/4680.en.html.

[2] Andrew Bortz, Dan Boneh, and Palash Nandy. Exposing private information by timing web applications. In Carey L. Williamson, Mary Ellen Zurko, Peter F. Patel-Schneider, and Prashant J. Shenoy, editors, *WWW*, pages 621–628. ACM, 2007.

[3] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks (Amsterdam, Netherlands: 1999)*, 48(5):701–716, August 2005.

[4] Scott A. Crosby and Dan S. Wallach. Denial of service via algorithmic complexity attacks. In *Proceedings of the 12th USENIX Security Symposium*, pages 29–44. USENIX, August 2003.

[5] Scott A. Crosby, Dan S. Wallach, and Rudolf H. Riedi. Opportunities and limits of remote timing attacks. *ACM Transactions on Information and System Security*, 12(3), 2009.

[6] Deniz Cevik. Waf tester v1.0, 2012. http://ttlexpired.com/blog/?p=234.

[7] Gordon *Fyodor* Lyon. Nmap network scanning - the official nmap project guide to network discovery and security scanning, 2009. http://nmap.org/book/osdetect.html.

[8] Edward W. Felten and Michael A. Schneider. Timing attacks on web privacy. In *SIGSAC: 7th ACM Conference on Computer and Communications Security*. ACM SIGSAC, 2000.

[9] Felix C. Freiling and Sebastian Schinzel. Detecting hidden storage side channel vulnerabilities in networked applications. In *Proceedings of the 26th IFIP TC-11 International Information Security Conference (IFIP/SEC)*, 2011.

[10] Kevin Spett. Blind sql injection, 2003. http://www.net-security.org/dl/articles/Blind_SQLInjection.pdf.

[11] Amir R. Khakpour, Joshua W. Hulst, Zihui Ge, Alex X. Liu, Dan Pei, and Jia Wang. Firewall fingerprinting. In *31th Annual IEEE Conference on Computer Communications (INFOCOM)*, Orlando, Florida, 2012.

[12] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, RSA, DSS, and other systems. In *CRYPTO: Proceedings of Crypto*, 1996.

[13] Mario Heiderich, Christian Matthies, and Lars H. Strojny. Php-intrusion detection system, 2012. https://phpids.org/.

[14] Michal Zalewski. p0f v3, 2012. http://lcamtuf.coredump.cx/p0f3/.

[15] Mike Schiffman and David Goldsmith. firewalk v0.99.1, 1999. http://packetstormsecurity.org/UNIX/audit/firewalk/.

[16] Modsecurity Wiki. Reference manual: Actions, 2012. http://sourceforge.net/apps/mediawiki/mod-security/index.php?title=Reference_Manual#Actions.

[17] Darren Mutz, Christopher Kruegel, William Robertson, Giovanni Vigna, and Richard A. Kemmerer. Reverse engineering of network signatures. In *IN PROCEEDINGS OF THE AUSCERT ASIA PACIFIC INFORMATION TECHNOLOGY SECURITY CONFERENCE, GOLD*, pages 1–86499, 2005.

[18] Yoshitaka Nagami, Daisuke Miyamoto, Hiroaki Hazeyama, and Youki Kadobayashi. An independent evaluation of web timing attack and its countermeasure. In *Third International Conference an Availability, Reliability and Security (ARES)*, pages 1319–1324. IEEE Computer Society, 2008.

[19] Robert "RSnake" Hansen. Xss (cross site scripting) cheat sheet, 2012. http://ha.ckers.org/xss.html.

[20] Taghrid Samak, Adel El-Atawy, and Ehab Al-Shaer. Firecracker: A framework for inferring firewall policies using smart probing. In *ICNP*, pages 294–303. IEEE, 2007.

[21] Sandro Gauci and Wendel G. Henrique. Wafw00f - web application firewall detection tool (svn r33), 2012. http://code.google.com/p/waffit/.

[22] Sebastian Schinzel. An efficient mitigation method for timing side channels on the web. In *2nd International Workshop on Constructive Side-Channel Analysis and Secure Design (COSADE)*, 2011.

[23] Sebastian Schinzel. *Unintentional and Hidden Information Leaks in Networked Software Applications*. PhD thesis, Friedrich-Alexander Universität Erlangen-Nürnberg, 2012.

[24] Sebastian Schinzel. Time is on my side - exploiting timing side channel vulnerabilities on the web, 2011. 28th Chaos Communication Congress http://events.ccc.de/congress/2011/Fahrplan/events/4640.en.html.

[25] Chris Shiflett. Foiling cross-site attacks, 2003. http://shiflett.org/articles/foiling-cross-site-attacks.

[26] Trustwave's SpiderLabs Team. Modsecurity - open source web application firewall, 2012. http://www.modsecurity.org/.

[27] Vigna, Robertson, and Balzarotti. Testing network-based intrusion detection signatures using mutant exploits. In *SIGSAC: 11th ACM Conference on Computer and Communications Security*. ACM SIGSAC, 2004.

[28] w3c Wiki. Same origin policy, 2012. http://www.w3.org/Security/wiki/Same_Origin_Policy.