Optimized Throttling for OAuth-based Authorization Servers

Peter Schuller[1][0000−0003−2736−2223], Julia Siedl[1,2][0000−0003−3409−5506], Nicolas Getto[1,2], Sebastian Thomas Schork[1][0000−0002−9519−9678], and Christian Zirpins[2][0000−0002−0838−2846]

[1] CAS Software AG, CAS-Weg 1-5, 76131 Karlsruhe, Germany
https://cas-future-labs.de/ {<prename.surname>}@cas.de
[2] Karlsruhe University of Applied Sciences, Moltkestr. 30, 76133 Karlsruhe, Germany
https://h-ka.de/iaf/dss {<prename.surname>}@h-ka.de

Abstract. Responsiveness is a key requirement for web-based enterprise software systems. To this end, throttling (or rate limiting) is often applied to block illegitimate traffic from outside-facing components and protect their computational resources. OAuth-based authorization servers are among the most popular components facing the web. Alas, the OAuth protocol introduces severe challenges to throttling. The OAuth protocol flow introduces indirections to client requests that make it hard to determine their source to apply rate limits. Moreover, fixed limits perform poorly in cases varying between high and low request loads. In this paper we propose solutions for both issues and provide an efficient solution for throttling in the context of OAuth. This includes integrated methods for a) cooperative throttling of authorization and resource servers as well as b) dynamic rate limiting as part of the throttling algorithm. We evaluate our approach based on a real-world use-case of enterprise CRM.

Keywords: Authorization Server Protection, OAuth Protocol, Cooperative Throttling, Dynamic Rate Limiting, CRM

1 Introduction

Modern web applications, especially those offering services to users over the internet, are facing significant variation regarding system usage [23]. In line with the distributed architecture of such systems, the aspects of authentication and authorization are usually provided as services of dedicated *auth servers* that build on standards like OAuth [16]. By design, such services are directly exposed to vast numbers of requests not only from users and clients, but also from the resource servers they are providing authentication and authorization for. These requests are leading to potential performance challenges on the auth server especially in multi-tenant environments [13]. Under peak load, they may cause significant performance drops or even service outages.

To protect auth servers from such issues, measures to manage system load have to be applied on different levels. Modern cloud-based systems offer standard

mechanisms to scale their capacity as one option to do so [4]. Whilst being useful for increases in legitimate traffic, this approach offers an additional surface to attackers. It prevents performance losses and outages but in turn can create direct financial impact. Thus additional measures to block malicious traffic has to be employed. A technique to do so is *throttling* or *rate limiting*. It is based on per-user limits and blocks all traffic from a user exceeding the defined limit.

The most common approach is generic throttling on the network layer. Whilst being widely available [1–3], this approach has the drawback of operating on limited information, e.g., with respect to source and target IP addresses. Such IP addresses are only partially useful to identify users. Techniques like NAT, VPN or proxies obfuscate an unknown number of users behind a single IP address. Thus, it is unclear, whether a single source IP address identifies a specific user or a group, let alone the group size.

For this reason, it is desirable to enrich the context of throttling mechanisms with additional information from higher system layers. Yet, such information should be still generic enough to make resulting throttling mechanisms widely applicable to different systems and domains, so using application-specific features (like end user profiles) is not a good option. A promising approach is to utilize specific information in the context of auth-servers, as these are closely related to the clients and users of the system but still offer a standardized generic view that is commonly available for many web applications.

Consequently, we tackle the question how throttling can be optimized by means and in the context of OAuth-based systems and break it down as follows:

1. How can meaningful user entities be identified for OAuth-based systems?
2. What are efficient and effective throttling limits for such users?

As a running example, we use a cloud-based multi-tenant Customer Relationship Management (CRM) system, more specific SmartWe [17]. It provides business users the ability to manage customer-related data such as appointments, e-mails, tasks or documents. Technically, SmartWe is a web application building on OAuth and consequently features the *user* as resource owner, a *client*, a *resource server* and an *auth-server* (see Figure 1).

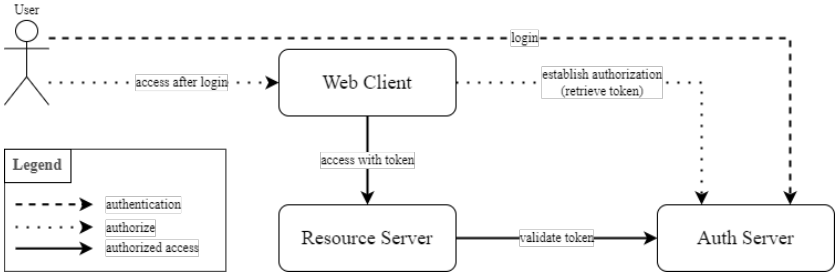


Fig. 1: SmartWe OAuth-based authentication flow

Users interact with a web-based client app. As this client is public, the *Authorization Code Grant Type* is used [5, p. 7], meaning that the client redirects the users' browser to the auth-server. After the user logs in, the client retrieves an initial token that it can exchange with the auth-server into a final *access token*. Different grant types exist that have different ways of obtaining such a token, but thereafter, all grant types follow the same interaction pattern.

A client can send a request for a resource (e.g., a data object like an appointment) to the resource server. To signal the users' approval for that access, each request includes the access token retrieved previously. Before returning the resource, the resource server sends this token to the auth-server for validation.

As the example shows, distributed systems may generate internal traffic while processing external requests. In the case of OAuth, resource servers query the auth-server to validate tokens. Originating from an internal server (and his IP address), those requests cannot be linked to a user on the network layer. It requires additional information from the application layer to resolve this level of indirection and apply proper throttling. Such information can be found in the request itself on the application layer. Throttling on the application layer includes more information on the client but is specific to the used protocol and cannot be implemented generically. Adaptations to the concrete application and use-case are required to make use of it. As we will show, more generic information on the level of OAuth can be used instead.

Beyond matching requests to users, another challenge for throttling is to adequately quantify their limits. Existing throttling solutions mostly require upfront configuration and in order to be effective for resource protection, limits need to be set to rather low rates. This might impose unwanted limitations to legitimate users, especially in situations where the systems overall load is low and resources are available. However, if limits are set rather high and all legitimate traffic is allowed, this might not be strict enough to protect the systems resources in situations of high load. A promising approach to resolve this problem is to dynamically adapt request rate limits to changing levels of system load. To this end, we show how optimized request rate limits for OAuth-based systems can be dynamically computed from the auth-server load.

Concerning the protection of OAuth-based auth-servers, some research already exist and even standards have been published [22]. However, most existing studies focus on protocol security [15, Chapter 9], [19, 11] or given implementations [18, 21]. To the best of our knowledge, protocol-specific throttling as a protection against overload has not been published yet. As regards dynamic throttling limits, there are also existing approaches. Most of them operate on network layer – either directly [20, 12] or embedded in an SDN [7]. Other approaches have implications on the overall system architecture [24] or focus only on outgoing traffic [14]. To the best of our knowledge, an approach to dynamically determine limits for incoming traffic on the application layer is still absent.

Summarizing the above, our approach provides three main contributions:

1. We outline and discuss several ways to identify single user sessions in an OAuth based system. By doing so, we are able to apply throttling also to endpoints that are subject to indirections via resource servers.
2. We propose a mechanism to set dynamic rate limits based on servers' varying load. It maps the system load measured as average response time to an allowed usage-frame, to obtain a load specific throttling limit.
3. We show how to integrate both solutions in a holistic architecture. That way, we obtain balanced throttling that reliably identifies users and applies appropriate limits for every load situation.

The rest of the paper is structured as follows. We describe details of OAuth-based servers that make it hard to control their request traffic in Section 2. In Section 3, we discuss related work on protecting authorization servers and dynamic request throttling. Our cooperative approach for dynamic throttling of OAuth-based resource access is described in Section 4 followed by its evaluation for the case of a real-world CRM system in Section 5. Section 6 concludes our research and outlines future work.

2 Foundations of OAuth-based authorization servers

In this paper, we focus on auth-servers that implement the protocol standards OAuth 2.0 (OAuth)[5] and OpenID Connect (OIDC)[10].

The OAuth 2 protocol [5] is about authorization and access rights delegation. It does not require any service (representing an OAuth client as defined in [5, sec. 2]) to store end user credentials for the purpose of accessing a certain resource. Instead, *tokens* are requested from a system service and issued to grant access to a resource server. OAuth distinguishes between short-lived *access tokens* used as credentials for the actual API consumption and long running *refresh tokens* used to renew access tokens. Access tokens may have a defined lifetime, are limited to a specific access right (scope) and may be revoked at any time. Access tokens are either self-contained or denote an identifier of variable length and consist of alphanumeric digits and some special characters [6].

In contrast to OAuth, OIDC [10] returns an additional *ID token* in form of a JSON Web Token (JWT) that contains actual user information. Thereby OIDC builds upon and extends OAuth with respect to authentication.

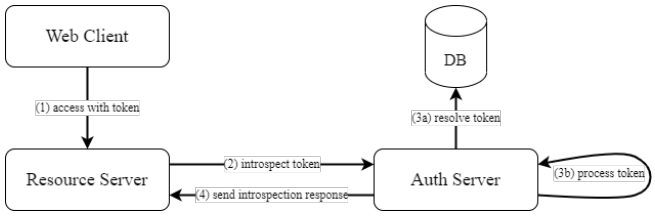


Fig. 2: OAuth token introspection

For validating a token [8], three types of actors are involved: an *authorization server*, a *client* and a *resource server* (see Figure 2). A client requests a resource from the resource server while presenting an access token (1). The resource server makes a request to the authorization server to validate the access token (2). The authorization server checks if the access token is valid (3) and communicates this back to the resource server (4). Depending on the result, the resource server may give the client access to the resource. The validation of the given access token at the authorization server involves database access and additional processing (3a, 3b) making the token introspection an expensive operation.

The communication is based on HTTP relying on the underlying network and thus sensible to latency and network failures. The system may be able to partially cope with a temporary, short unavailability of the auth-server. For a short period existing sessions may stay active because tokens are cached by the resource server but new login requests cannot be handled. A persisting unavailability on the other hand represents an imminent risk for the overall system availability.

RFC 6749 [5] on the OAuth 2 standard states for the access token requests that *"the authorization server MUST protect the endpoint against brute force attacks (e.g., using rate-limitation or generating alerts)"*. RFC 7662 [8], specifying the token introspection, discusses attack vectors that arise when it is *"left unprotected and un-throttled"*. Thus, the standards are aware of the importance of protecting the resource server and see throttling as an approach to do so.

Especially the token introspection endpoint has further attack vectors if throttling is applied in a generic manner. This mainly arises from the indirection that related requests have to take. From a logical perspective, the request to this endpoint is a direct result of a request that a client sends to a resource server.

3 Related work

In the following, we summarize existing work that is related to our research problem and solution approach. In particular, this paper deals with *a) protecting authorization servers* and *b) dynamic request throttling*.

Concerning the protection of authorization servers, literature mostly focuses on the security of the protocol itself. The corresponding standard on the "OAuth 2.0 Threat Model and Security Considerations" [22] explicitly considers the communication between authorization server and resource server out of scope. All described attacks are focused on retrieving information or gaining access, but not on causing service outages. Similar arguments can be found in further literature on this topic such as [15, Chapter 9], [19, 11]. [11, p. 185] even discusses security considerations specifically for the token introspection endpoint, highlighting its importance still missing any attacks focusing on its availability. Other research in this field adds the aspect of flaws that are introduced by insecure implementations of the OAuth protocol [18, 21].

As regards dynamic request throttling, authors of [24] present an architecture for internal service design, in which services are split up into multiple event-driven stages which are all connected by using explicit request queues. This is a

problem for already existing services or programmers who don't necessarily want to base their services on the *event-driven architecture* (SEDA). We propose an additional component instead, which produces a dynamic throttling limit.

The importance of such a mechanism can also be seen in [14]. The paper presents an implementation of distributed rate limiters, which enforce a global rate limit across traffic to multiple sites to enable coordinated policing. This can be used to limit global consumption of a priced third-party service to set strict cost limits. While this paper deals with outgoing traffic, it does show the inherent need of such a mechanism in cloud-based services. We use the basis of throttling, but we concern ourselves with incoming traffic.

Another perspective on rate limiting is shown in [20]. The paper introduces a scalable rate limiting method by using a NIC. The host CPU classifies data packets and queues them in a per-class queue, which specifies rate limits for its class. The presented NIC, called SENIC, handles metadata and scheduling. This solution requires additional hardware and its scalability comes from static limits. We propose a software solution instead, that doesn't need specialised hardware.

Authors of [12] propose dynamic rate limiting as a defense mechanism against flooding based (D)DoS attacks. Every edge router in a network that drops a lot of packets gets punished with a strong static limit, while routers that drop few packets may use a weak static limit. This solution works on the network level, while we propose a solution on the application level.

Another dynamic rate limiting mechanism against (D)DoS can be found in [7]. This paper proposes a solution based on a sliding–window algorithm, which works on a weighted network abstraction. The weighting is based on the queue capacity of the controller and the number of rules in the switch. This solution is not using a pure mathematical solution, but is using a deep-learning algorithm for it's dynamics. It focuses security problems that are specific for SDNs. We propose a more general solution instead.

4 A cooperative approach for dynamic throttling of OAuth-based resource access

In the following, we present our approach for optimized throttling of OAuth-based auth-servers. First, we focus on the problem of applying throttling to the introspection endpoint and define related objectives. As a solution, we discuss considerations for user or client identification and describe protocol adaptations. Second, we describe how dynamic limits can be implemented. We propose a metric for estimating the server load and show the related calculation of limits. Finally, we present a common architecture to combine both parts.

4.1 Protecting the introspection endpoint

The first goal is to protect the auth-server from overload via the token introspection endpoint (see Figure 2). This endpoint is particularly important, since all other servers of the system depend on it for authorizing their clients. Furthermore, this endpoint leads to indirections unlike other auth-server endpoints.

General considerations One solution would be to indirectly protect the auth-server by applying throttling at the resource-server only. This has the advantage of evading indirections. However, this approach is not flexible enough as it leaves the auth-server completely dependent on the protection mechanisms of the resource-servers. As a result, this would limit the deployment options of the auth-server. It could only be deployed with trusted resource-servers like those belonging to the same organisation. An additional observation is that a resource-server should be independently protected from the auth-server, because it is publicly available for clients. The approach presented in this paper can be confidently applied to resource-servers outside the control of the organisation if they implement the presented protocol extension.

Another naive approach would be for the auth-server to apply throttling to resource-servers. If the granularity is a whole resource-server this could be fatal. Imagine that a client makes an excessive amount of requests to the resource-server that subsequently reaches its limit for validating access tokens. Then the resource-server would be left in a non-operative state for all clients. Therefore an auth-server has to apply throttling to requests at a finer granularity.

User identification A partial conclusion is that there need to be protection mechanisms in both auth-server and resource-server. When designing the approach, we observed that resource-servers can minimize the overhead for classification by letting the auth-server do parts of the data analysis. By cooperating, both servers have sufficient information to classify requests, while keeping the exchanged information at a minimum.

When referring to access tokens, they can be invalid. In the context of this paper, "invalid" only refers to correctly formed access tokens. They may be invalid because they have expired, are guessed randomly by a malicious actor or are damaged due to client implementation errors. Access tokens that have the wrong length or contain illegal characters are not examined here, because they can be filtered out easily by an API-gateway.

When the resource-server receives a request, it can categorize it by access token, client ID and the source IP address + port of the client. To know if the access token is valid, the resource-server has to query its token cache or the auth-server. The resource-server has to query the auth-server for each unknown access token, as the number of possibly valid access token is virtually unlimited.

On the side of the auth-server, this request needs to be categorized in order to apply throttling mechanisms before it is possibly processed. To be efficient, this categorization needs to be faster than the processing itself and thus must be light on resource usage. Therefore, user information associated with an access token should not be used for throttling, as this would allocate the same resources it is meant to protect. Consequently, the categorization can only be done via the access token. Since an access token represents the access rights for a client, this categorization identifies a combination of resource owner and client instance. However, if invalid access tokens are used to classify requests for throttling, the auth-server has to store those potentially unlimited numbers of invalid access

tokens. For the auth-server, those invalid access token do not entail any meaning other than they were received by a resource-server. The resource-server however, can map these invalid access token to an IP address.

Approach to protect the introspection endpoint We use access tokens for classification and enhance them with the client IP address in case of invalid access tokens. Depending on how access tokens and IP addresses are combined, different scenarios are possible at the client. Table 1 classifies these scenarios and depicts their semantics. In particular, there are client errors and malicious attacks. Some combinations are marked "unusual", as for this kind of error to occur, an implementation has to have errors at numerous places.

		Same access token		Different access token	
		Token valid	Token invalid	Token valid	Token invalid
Same IP address	Normal rate	**Single client**	Error/attack	**Many clients at same access point**	Error/attack
	High rate	Error/attack	DOS/ unusual error	**Web application**	DOS/ unusual error
Different IP addresses	Normal rate per IP	**Single client switches access point**	Error/attack		
	High rate per IP	Attack/ unusual error	DDOS/ unusual error		

Table 1: Classification of resource server requests

The idea is that an auth-server counts requests for access tokens and the percentage of invalid requests from IP addresses of clients. If it detects unwanted behaviour, it generates specific errors on these requests. The resource-server has to remember these errors and block those requests instead of forwarding them to the auth-server. Thus, the auth-server needs to manage counters for access tokens and IP addresses. The resource-server needs to manage a collection of currently blocked access token and currently blocked IP addresses of clients. The modified process looks the following (modifications marked as *italics*):

1. A resource-server receives a client-request for a resource with an access token.
2. *If the access token or IP address is marked as blocked, the resource-server may return an error.*
3. Otherwise the resource-server makes a request to the auth-server to validate this access token *while also passing the client IP address with it.*
4. *The auth-server returns a specific error if the limit is reached for the access token or IP address.*
5. The auth-server validates the access token *while registering the use of access tokens and whether an invalid access token originated from that IP address.* Then it returns the result.

6. *If the resource-server received an error it marks the IP address or access token as blocked and may return an error to the client.* Otherwise it answers the client request normally.

The existing protocol of the introspection endpoint changes in two ways. First, the resource-server has to relay the client IP address to the auth server. Second, the resource-server has to handle the two new types of errors the auth-server produces. Those are throttling errors and errors for an IP address, where invalid access tokens originated.

In the auth-server, overhead results from classifying the requests. There are two parameters to configure the classification. First, a limit for the number of requests per access token within a time window needs to be defined. Second, there needs to be a maximum percentage of allowed invalid requests from an IP address. This percentage should also include a threshold for the number of requests from that IP address. Otherwise a new IP address would be blocked immediately if the first request contains an invalid access token. The configuration of these parameters determines the effectiveness of the whole approach. Section 4.2 discusses, how an adaptive approach can be useful here.

At the level of the resource-server, some additional computational resources are needed as well. The collections of blocked access token and IP addresses need to be managed. This data is less complex than the data managed by the auth-server and changes less frequently.

Advantages and limitations The advantage of relaying the client IP address to the auth server is, that the case of invalid access token can get handled without needing to store those tokens. The auth-server should not store a collection of invalid tokens indiscriminately, because this list may grow indefinitely. Thereby, the auth-server can protect itself from a broken client that sends invalid token.

If throttling was only implemented in the resource-server, a client could reset its limit by refreshing its access token. The resource-server does not have a mechanism to monitor relations between access token. The presented approach has the advantage that the auth-server can track an access token even if it is refreshed. Since the auth-server also handles token refreshments it can update the corresponding throttling counter when this endpoint is called.

As regards limitations, the presented approach does not define strategies to handle DoS, let alone DDoS attacks. First of all, it presents a simple mechanism to identify its indirect clients at a sensible granularity. To be more explicit, the granularity can be described as the client session. Still, the contribution is that the presented mechanism may be integrated in frameworks that handle attacks, so they become more effective for OAuth scenarios.

Also, the solution could be further improved for the case of invalid access tokens from the same client IP address. If multiple clients are using the same IP and one tries using an invalid access token, all clients might be blocked. This edge-case can get mitigated by dynamically adapting the percentage of allowed invalid requests from the same IP address based on server load. Partly, this can

also be mitigated by using a less strict limit if the client is a web application. For accuracy, further research is needed on how to temporarily store invalid access tokens in those edge-cases while limiting the maximum number of stored tokens.

4.2 Dynamic rate limiting

The goal of a dynamic throttling limit is to combine the positive effects of a weak limit, such as an increased user experience, with the positive effects of a strong limit, such as the protection of server resources. A dynamic limit, at any given point in time, is therefore derived from the server load at that time. To that end, the average response time is used as an indicator to see, how big the current server load is. From there, the limit, that is required to either bring the server load down or to keep it steady, is calculated.

Server load The average response time is calculated on the last responses in a given time frame. While a shorter time frame, such as a few seconds, can depict sudden traffic spikes in the resulting value, a longer time frame, such as a minute, can represent the overall server load situation better. If sudden traffic spikes can be expected, a middle ground value between these two should be considered.

To calculate the server load, an upper and lower limit for the average response time has to be defined beforehand. If the server has an average response time close to or above the upper limit, it is presumed to be busy at that given point in time. Otherwise, if the average response time is under the lower limit, the server is presumed to be underutilized. This relation between a) the average response time of the server and b) it's presumed server load is used to calculate a limit. This limit can be used by a throttling mechanism.

These average-response-time-limits must be defined upfront. In our implementation, the lower limit was set to one second, since that is roughly the point, when a user will start noticing a delay [9]. As a maximum, five seconds was chosen. Since a user will lose concentration after a delay of ten seconds [9], some buffer is present to ensure that the server won't reach such a high delay.

Throttling limit Since the goal of a dynamic limit is to counteract a rising server load, the limit has to get stricter accordingly. Nonetheless, the throttling limit can't shrink without any limitations. If it would get too low, a normal user could not use the service anymore. Therefore a minimum and a maximum limit have to be defined based on the expected server behaviour in rising traffic.

The expected server behaviour can be seen in Figure 3. Expected is a stable load curve, which can be seen as the bottom line, up until a specific point, at which the curve will grow exponentially. At this point, the throttling limit needs to get stricter. The curve above the load curve shows the defined maximum average response time. The throttling limit needs to be at it's strictest, when the load curve cuts the maximum average response time.

This behaviour is represented in the first two rows of Equation 1. Here, $f(x)$ is defined as the calculated limit that will later be used by the throttling

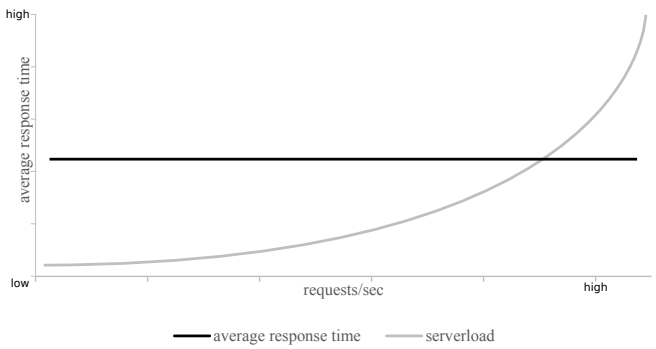


Fig. 3: Expected server behaviour for rising request load

mechanism. Furthermore, $Min\_Avg\_Resp\_Time$ is defined as the value, where the load curve starts to grow. $Max\_Avg\_Resp\_Time$ is defined as the value, where the graphs intercept. Both points can be seen in Figure 3.

$$f(x) = \begin{cases} Max\_Limit & x \leq Min\_Avg\_Resp\_Time \\ Min\_Limit & x \geq Max\_Avg\_Resp\_Time \\ t(x) & Min\_Avg\_Resp\_Time < x < Max\_Avg\_Resp\_Time \end{cases} \tag{1}$$

The third row in Equation 1 is a function that calculates the limit for a load value in between these set average response time limits. To this end, a linear function $t(x)$ with negative slope is used (see Figure 4).

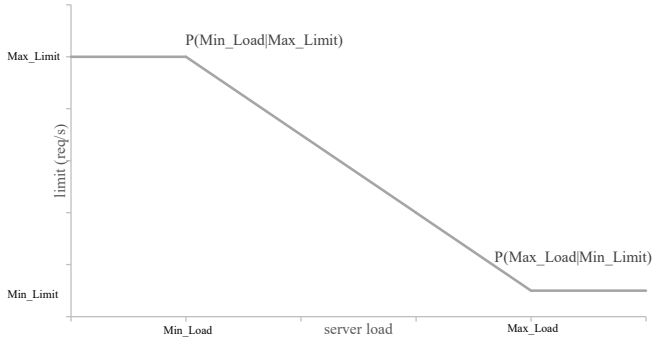


Fig. 4: Graphical representation of throttling function $f(x)$

Since a limit has to be defined for every request, this function has to be called for every request and before the throttling. For clients, this process does obfuscate the throttling limit and create additional complexity for the developer. however, this obfuscation can also be a benefit in attacking scenarios as well, since a potential attacker can't just stay under the throttling limit.

4.3 Common architecture

Summarising the above, we presented concepts for solving two problems that arise for throttling of OAuth-based systems. First we use tokens in combination with IP addresses to identify users.Once we properly identify the users, the next step is to find good throttling limits. We presented an approach to dynamically link this limit to the current server load.

In the context of the overall system architecture, both partial solutions are implemented directly in the auth-server. This shared context enables us to integrate the parts. When a request arrives two major steps have to be taken:

1. The user for whom this request is to be accounted and the usage is to be calculated, needs to be identified. This is usually done based on IP address. In our approach, this is done as described in section 4.1.
2. Based on this usage and a limit, the decision whether to process the request or not needs to be taken. Usually this limit is statically configured. In section 4.2 we describe our approach to dynamically calculate this limit based on the current system load.

By separating both parts in self-contained components, we enable a modular usage of the overall approach. Not only can the parts be reused, they can also be recombined with enhanced variants for each sub-task.

In a broader systems view, only the first part of our solution has an effect on other components. It requires all resource-servers that query the user introspection endpoint to include the IP address, from which the request originated.

5 Evaluation

In Section 4, we outlined a solution for throttling in OAuth-based systems. As our solution addresses two separate issues, our evaluation is split in two parts as well. By using one common example, we underpin the integration of both solutions, providing a comprehensible and complete approach. For both parts, we show that they fulfill the requirements whilst being efficient and scalable.

Subsequently, the version of the auth-server without modification will be called *original server* and the server that implements the proposed solutions is called *modified server*. Concerning the experimental set-up, we have used *JMeter* to model user interactions with the auth-server. Both the server and *JMeter* were run locally on the same machine to prevent fluctuation in the response time caused by the network connection. Since the auth-server is designed scalable, the actual number of requests is not important since it only reflects the capabilities of the local machine. The most meaningful observations concern comparisons of the original and modified servers.

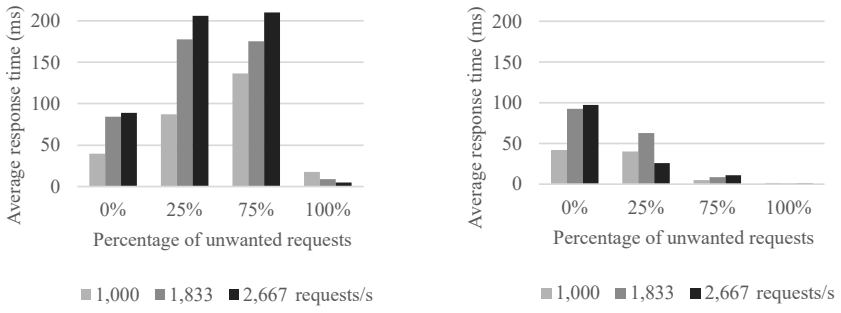
5.1 Testing the protection of the introspection endpoint

The following test is a comparison of the response times of the auth-server under load with and without throttling. The goal was to validate that the proposed

solution for the introspection endpoint produces an acceptable overhead in the auth-server. Along the way, the findings also confirm the precondition that throttling can reduce server load. There were no concrete attack scenarios included in the test, only scenarios where the volume of request exceeds the defined limits. The reason is, that it does not serve the goal of measuring the overhead and throttling is imprecise against attacks by design.

The idea of the testing approach is to approximate the server load by measuring the response time of the auth-server. The testing framework simulates the behaviour of the resource-server. It sends requests for different clients to the introspection endpoint. As an indicator for the event that the auth-server reached its processing limit, it is observed when the response time stops to increase.

Figures 5a and 5b show the resulting response time for sending requests to the introspection endpoint. The different shades represent the requests per second that should be produced by *JMeter*. The percentage of unwanted requests are those requests that produce a throttling error in the modified version of the server. For example, for 25 percent of unwanted requests, when sending 1,000 request per second, 250 of those requests produced a throttling error in the modified server version. The original server responded to all requests. Wanted requests were modeled in a way that 100 different access tokens where combined randomly for each request with 101 different IP addresses. The unwanted requests were modeled by a combination of one access token and one IP address.



(a) Response times of original server (b) Response times of modified server

Fig. 5: Response times for different percentages of unwanted requests and different numbers of requests per second to the introspection endpoint

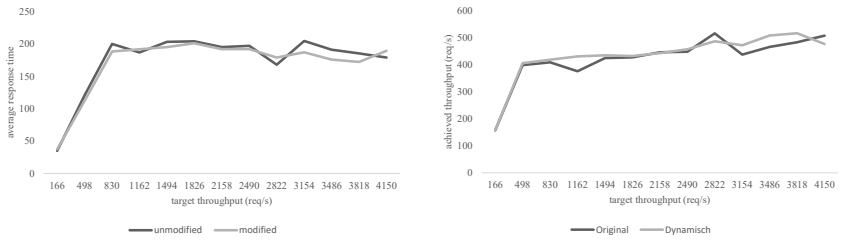
Both diagrams show that the response time increases as more requests are sent. Due to a peculiarity of the server implementation, the response time for the original auth server increases when the percentage of unwanted requests is increased. This does not influence the test results. For 100 Percent of unwanted requests the response time is low for the original server because then the introspection for only one access token is requested and it gets cached.

When comparing the response time of the original server and the modified server for 0 percent unwanted requests, the overhead of the proposed solution becomes visible. For the same number of requests, the original server has a response time that is about 5 Percent lower. This means that the solution produces measurable linear overhead. The results from the modified server also show that the response time is dramatically decreased when a percentage of the requests get throttled. When all requests get throttled the server has a response time of under 10 ms meaning the classification is efficient.

5.2 Testing the dynamic calculation of rate limits

A user interacts with the auth server according the OAuth 2 *Authentication Code Grant Flow* and every test models 100 users.

Overhead To show the overhead of the dynamic component, *JMeter*-Tests were carried out against the original auth-server as well as the modified one with respect to the average response time and the achieved throughput. After every test, the target throughput got increased to represent increasing traffic rates. The results for the average response time are shown in Figure 6a.



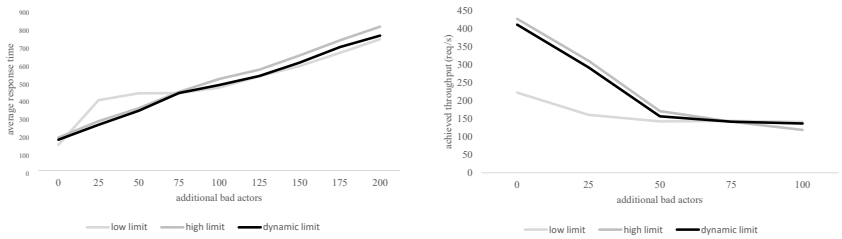
(a) Comparison: average response time (b) Comparison: achieved throughput

It can be observed that the difference in terms of the average response time is minimal. At a target throughput of 830 request per second, the server is starting to get to it's limit and the difference starts to get lost in noise. The results for the achieved throughput are shown in Figure 6b. Again, no significant difference can be observed. The two servers are almost identical up until a target throughput of 830, where the limit is reached once again. Random noise can be observed again, but there isn't any noteworthy difference in the reached throughput.

Concluding the results, the overhead of the component doesn't impact the server in terms of a significant slowdown. This applies wit respect to the average response time as well as the number of requests the server can handle.

Throughput The component was tested against basic throttling algorithms with a high as well as a low static limit. Instead of a rising target throughput,

additional bad actors got added. These do not follow the OAuth 2 standard, but instead repeatedly request a resource from the resource server with a forged token to increased server load. The test results with respect to the average response time is show in Figure 7a.



(a) Average response time

(b) Throughput

Fig. 7: Comparison of static vs. dynamic limits

It is expected, that a low limit grows slower than a high limit, which can be observed from 75 bad actors and above. A spike on the server with the low limit from zero to 75 bad actors can be observed, which happens due to the high average response times for the first few requests. These are the requests for the login page, which take longer to render than a request for a single resource. It also can be observed, that the server with a dynamic limit follows the static limit, which results in the lower average response time. The results regarding the achieved throughput are shown in Figure 7b.

A higher limit is expected to result in a higher achieved throughput. With a rising server load, both high and low limits result in a lower achieved throughput, but the higher limit decreases stronger. Therefore these two curves cut at 75 additional bad actors. It can be observed that up until this point, the dynamic component results in a similar throughput than the high limit. After this point the results will be similar to the results of the low limit.

With respect to the average response time and the achieved throughput, the dynamic component results in a good middle ground between a high and a low limit to a certain extent. With a rising server load, the component was observed to result in similar values than the two static limits.

6 Conclusion

In this paper we showed how throttling by means and in the context of OAuth-based systems can be optimized.

To overcome the issue of indirection on the auth-servers' token introspection endpoint, we presented a cooperative throttling approach. It combines the clients' source IP address that is forwarded by the resource server with the provided access token to accurately identify user sessions. We showed that with growing amounts of unwanted traffic, response times are dramatically improved.

Regarding the applied throttling limits, we contribute a dynamic throttling solution. It links the throttling limit to the server load, which is measured in terms of the average response time. We showed that our approach is able to combine the advantages of high limits in low load scenarios with the advantages of low limits in high load scenarios.

Putting both parts together, we discussed how the two partial solutions can be integrated to achieve an overall optimized throttling mechanism for OAuth-based systems. Whilst the integrated architecture has been designed, yet no integrated implementation exists. Thus as future work, an implementation of the overall approach is to be done.

References

1. Apache mod_evasive module, https://github.com/jzdziarski/mod_evasive, [Online; last accessed 15-June-2022]
2. limitipconn2 readme, http://dominia.org/djao/limitipconn2-README, [Online; last accessed 15-June-2022]
3. Rate limiting with nginx and nginx plus, https://www.nginx.com/blog/rate-limiting-nginx/, [Online; last accessed 15-June-2022]
4. Abbott, M.L., Fisher, M.T.: The art of scalability: Scalable web architecture, processes, and organizations for the modern enterprise. Addison-Wesley (2015)
5. D. Hardt, E.: RFC6749 - The OAuth 2.0 Authorization Framework (2012)
6. D. Hardt, M.J.: RFC6750 - The OAuth 2.0 Authorization Framework: Bearer Token Usage (2012)
7. El Kamel, A., Eltaief, H., Youssef, H.: On-the-fly (d)dos attack mitigation in sdn using deep neural network-based rate limiting. Computer Communications **182**, 153–169 (2022)
8. J. Richer, E.: RFC7662 - OAuth 2.0 Token Introspection (2015)
9. Miller, R.B.: Response time in man-computer conversational transactions. In: Unknown (ed.) Proceedings of the December 9-11, 1968, fall joint computer conference, part I on - AFIPS '68 (Fall, part I). p. 267. ACM Press, New York, New York, USA (1968)
10. N. Sakimura, J. Bradley, e.a.: OpenID Connect Core 1.0 (2014)
11. Parecki, A.: Oauth 2. 0 Simplified. Lulu.com (2017)
12. Patil, R.Y., Ragha, L.: A dynamic rate limiting mechanism for flooding based distributed denial of service attack. In: Fourth International Conference on Advances in Recent Technologies in Communication and Computing (ARTCom2012). pp. 135–138. Institution of Engineering and Technology (2012)
13. Pieter-Jan Maenhaut, Hendrik Moens, M.D.e.a.: Characterizing the performance of tenant data management in multi-tenant cloud authorization systems. In: 2014 IEEE Network Operations and Management Symposium (NOMS). IEEE (2014)
14. Raghavan, B., Vishwanath, K.e.a.: Cloud control with distributed rate limiting. ACM SIGCOMM Computer Communication Review **37**(4), 337–348 (2007)
15. Richer, J., Sanso, A.: OAuth 2 in action. Simon and Schuster (2017)
16. Roberto, S.d.O., da Silva, R.C., Santos, M.S., Albuquerque, D.W., Almeida, H.O., Santos, D.F.: An extensible and secure architecture based on microservices. In: 2022 IEEE International Conference on Consumer Electronics (ICCE). IEEE (2022)
17. SE, S.W.: SmartWe World SE | Unabhängige CRM Cloud-Plattform. https://www.smartwe.de/ (2022), [Online; last accessed 15-June-2022]

18. Shernan, E., Carter, H., Tian, D., Traynor, P., Butler, K.: More guidelines than rules: Csrf vulnerabilities from noncompliant oauth 2.0 implementations. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. pp. 239–260. Springer (2015)
19. Siriwardena, P.: Advanced API security: OAuth 2.0 and beyond. Apress (2019)
20. Sivasankar Radhakrishnan, Yilong Geng, V.J.e.a.: SENIC: Scalable NIC for End-Host rate limiting. In: 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14). USENIX Association, Seattle, WA (Apr 2014)
21. Sun, S.T., Beznosov, K.: The devil is in the (implementation) details: an empirical analysis of oauth sso systems. In: Proceedings of the 2012 ACM conference on Computer and communications security. pp. 378–390 (2012)
22. T. Lodderstedt, e.a.: RFC6819 - OAuth 2.0 Threat Model and Security Considerations (2013)
23. Urdaneta, G., Pierre, G., Van Steen, M.: Wikipedia workload analysis for decentralized hosting. Computer Networks **53**(11), 1830–1845 (2009)
24. Welsh, M., Culler, D.: Adaptive overload control for busy internet servers. In: Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems - Volume 4. p. 4. USITS'03, USENIX Association, USA (2003)