Security Evaluation on Amazon Web Services' REST API Authentication Protocol Signature Version 4

Khanh Hoang Huynh
Security and Network Engineering
University of Amsterdam
Amsterdam, Netherlands
hhuynh@os3.nl

Jason Kerssens
Security and Network Engineering
University of Amsterdam
Amsterdam, Netherlands
jason.kerssens@os3.nl

***Abstract*—The Signature Version 4 protocol is used in Amazon Web Services to sign API requests, providing data integrity, verification of the requesting user, and protection against reuse of the signed portions of requests. In this research we evaluated the security of this protocol by first understanding the protocol workings to choose known attacks that undermine one or more of the three aforementioned features that the Signature Version 4 protocol should provide. We have performed and analyzed a replay attack, modified requests, probed for the possibility of performing an HTTP smuggling attack, and analyzed the possibility for a timing attack. From our research we have found that replay attacks are possible if requests are replayed within a limited time window, modifying signed portions of requests is not possible, HTTP smuggling attacks are prevented, and we have found an indication that timing attacks might be possible. We conclude that, in practice, the Signature Version 4 protocol can be considered secure.**

I. Introduction

Amazon Web Services (AWS) is one of the largest web service and cloud providers globally. It offers many services from Infrastructure as a Service (IaaS) to databases and analytics. Many consumers make use of these services. As use of these services often requires sending private information or performing operations that cost money, it is important that confidentiality, integrity, and availability are preserved. This is also relevant when sending requests to API endpoints, where communication between the requester and the endpoint needs to be carried out in a secure manner. In addition, often one needs to authenticate themselves to become authorized to make requests.

There exist many schemes with which secure communication is possible by providing authentication, authorization, data integrity or confidentiality. Some of the most widely used schemes include Transport Layer Security (TLS) [1], Open Authorization (OAuth 1.0 [2] and OAuth 2.0 [3]), and HTTP authentication [4]. These schemes all offer different functionalities, but none of these schemes provide end-to-end integrity of exchanged data. In order to achieve end-to-end integrity with a scheme, data need to be signed cryptographically.

There is no public standard scheme that achieves this, which is why AWS has implemented its own proprietary protocol named Signature Version 4 (SigV4). The SigV4 protocol is used to sign HTTP requests to AWS' API endpoints in order to provide verification of the identity of the requester (authentication), in-transit data protection (data integrity), and protection against reuse of the signed portions of requests [5]. The previous AWS Signature protocol, Signature Version 1, has been proven insecure [6]. Signature Version 2 has been deprecated and replaced in favor of SigV4, as SigV4 offers a more flexible signing method and better protection against key reuse [7].

In this research we will give an assessment on the security of the SigV4 protocol, which is used when HTTP requests are sent to AWS' API endpoints. This is done by looking at possible attack vectors on the protocol. We will look at attacks that undermine at least one of the aforementioned features that SigV4 provides, i.e., authentication, data integrity, and protection against reuse of signed portions. Normally, SigV4 is used in combination with TLS to further enhance security by also providing confidentiality of requests and protection against replaying requests. In this research we will disregard the use of TLS and only consider SigV4. Investigating other security measures that are in place for AWS' APIs is thus not within the scope of this research.

A. Research questions

As previously mentioned, SigV4 is used to provide verification of the identity of the requester, in-transit data protection, and protection against reuse of the signed portions of a request. Therefore, to evaluate the security of SigV4 on these three points, the following research question is defined:

Does the Signature Version 4 protocol, used when sending a request to AWS REST API endpoints, provide data integrity, verification of the requesting user, and protection against reuse of signed requests?

To answer our research question, we have defined the following sub-questions:

- *How does the Signature Version 4 Protocol ensure data integrity, verification of the requesting user, and protection against reuse of signed requests?*
- *What kind of attacks are able to undermine data integrity, verification of the requesting user, or protection against reuse of signed requests?*

B. Structure

The remainder of this paper is structured as follows. Section II reviews related work of SigV4. In section III we will give an overview of the workings of the SigV4 protocol. In section IV we outline the attacks, which we will perform on the SigV4 protocol. The results of our experiments will be shown in section V. We will discuss these results in section VI and draw a conclusion from them in section VII. Finally, in section VIII, we suggest points for future work.

II. Related Work

This section gives an overview of work that has previously been conducted on the subject of HTTP message signing schemes, attacks on signing methods, and attacks on other security protocols.

A. HTTP Message signing schemes

L. Lo Iacono and H.V. Nguyen [8] have provided a generic authentication scheme for REST. They define three elements of REST messages (control data, resource meta data and resource representation meta data) that form the Uniform Interface. They state that all three elements of the Uniform Interface must be integrity protected and authenticated to prevent malicious changes being made to the message. This can be achieved by creating a cryptographic signature of the message which takes all three elements into account.

The IETF draft *Signing HTTP Messages* [9] shows an approach to signing HTTP messages as well. They define two mechanisms: signing and authorization. The signing scheme is used to verify the integrity of the message while the authorization scheme is used to give a requester access to certain resources.

B. HMAC side-channel attacks

The Signature Version 4 protocol signs messages using Keyed-Hashing for Message Authentication (HMAC) [10]. There exist side-channel attacks on HMAC. C.S. Islam and M.S.H. Mollah [11] have found that information about the HMAC key, such as its length and its Hamming weight, can be revealed by monitoring the execution time of the algorithm.

Fouque, Pierre-Alain, et al. [12] have shown that it is possible to recover the secret key of HMAC-SHA1. They have achieved this by monitoring the electromagnetic radiation emitted by the processor that executes the algorithm. This attack requires physical proximity to the processor to monitor the electromagnetic radiation, however.

C. Attack on Signature Version 1

Colin Percival [6] had found an insecurity in the Signature Version 1 protocol. The exploit relied on the fact that there were no delimiters between keys and values when signing the HTTP query string. This meant that different query strings resulted in the same signature, which allowed an attacker to forge requests with valid signatures more easily.

D. HTTP smuggling

In 2005, Watchfire had documented an HTTP smuggling attack [13]. At a Black Hat USA Briefing on the 7th of August 2019, security researcher James Kettle gave a briefing about HTTP request smuggling using new techniques. Along with the briefing he also published a whitepaper in which he explains these new techniques [14]. These techniques may be used to smuggle requests by placing them inside of valid requests, circumventing the Sigv4 protocol.

III. Design

In this section we will discuss the workings of the SigV4 protocol as well as outline the differences between the SigV4 protocol and the Escher protocol. The Escher protocol is an open source project based on AWS' SigV4 protocol. It allows us to look into the source code to understand SigV4 better, perform attacks on the protocol locally, and modify the protocol if desired.

A. Signing requests with Signature Version 4

SigV4 is a proprietary protocol which is used to create signatures for HTTP requests destined for AWS' API endpoints. HTTP requests can be sent either via the AWS Command Line Interface (AWS CLI) [15], using one of the AWS SDKs [16], or they can be crafted manually. When using AWS CLI or AWS SDKs, the requests are signed automatically. Requests have to be signed manually when the HTTP requests are created manually, however.

When a request is signed it provides verification of its sender, data integrity, and prevents replay attacks of the signed portions of the request. All HTTP requests, with the exception of anonymous API calls and some specific calls to AWS Security Token Service (STS), must be signed.

The SigV4 protocol is explained in AWS' documentation [17]. The protocol consists of four steps, which are shown in Figure 1.

In the first step of the SigV4 protocol, a canonical request is created, as can be seen in Figure 1. This canonical request consists of: the HTTP request method, a URI-encoded version of the absolute component of the URI, the query string, a list of HTTP request headers one wants to sign, a list of HTTP request header names which are signed, and the hashed payload of the request. Some services, such as the Amazon S3 service, allow for unsigned payloads. In this case, the literal string *UNSIGNED-PAYLOAD* is included in the canonical request instead of a hashed payload.

In second step a string is created which will be used in the third step to create a signature. The string begins with the

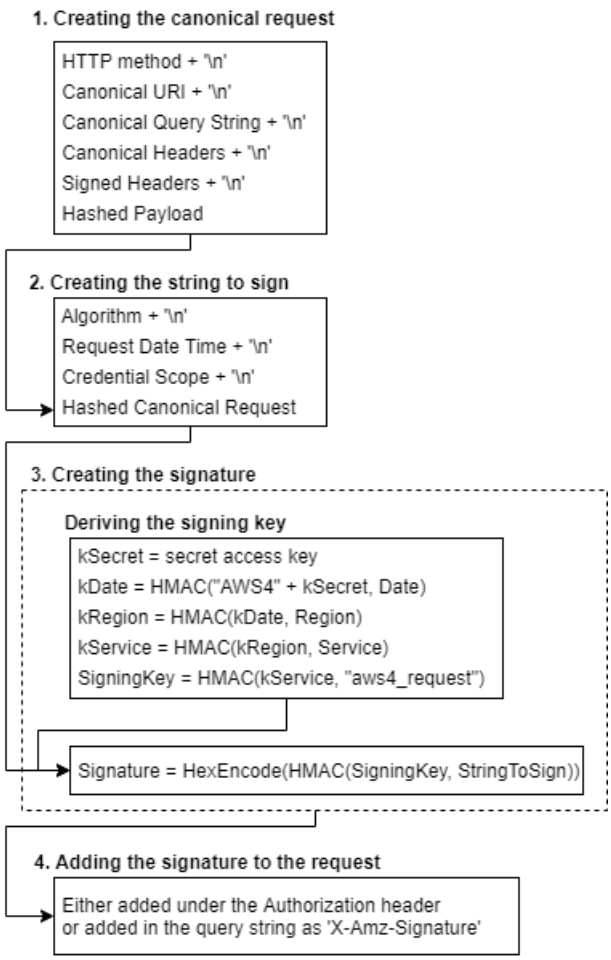

Figure 1. Procedure of the Signature Version 4 protocol for creating and adding a signature to a request.

hashing algorithm used in the first step to hash the payload, usually this is HMAC-SHA256, however AWS supports the use of HMAC-SHA1 as well. Other elements of the string are the timestamp in ISO8601 basic format, the credential scope, and the hashed canonical request. The credential scope specifies the date, the AWS region the request is targeting, and the service for which the request is intended. The hashed canonical request is hashed with the same algorithm specified at the start of the to be created string.

In the third step, the signing key is derived from the secret access key. The secret access key is a security credential for one's user account. Secret access keys are used to authenticate users and make programmatic calls to AWS API operations or to use AWS CLI commands. The procedure for deriving the signing key from the secret access key can be seen in Figure 1. In this procedure, either HMAC-SHA1 or HMAC-SHA256 is used. According to the AWS documentation, the signing key is valid for up to seven days for S3 services [18]. As for IAM this is unknown, as it was not documented.

Finally, in step four, the signature is added to the HTTP request. This can be done in two ways. The first method is by adding a header, which includes the signature and any additional information needed to create the signature, to the HTTP request. The second method is by adding the signing information to the query string of the URL. A limitation of the second method, is that a URL can only have a limited number of characters. This limit is dependent on the server that receives the request. It is recommended to keep the URL length shorter than 2000 characters [19].

A request can be sent to an AWS API endpoint after the steps shown in Figure 1 have been executed. When a signed request is received, AWS will perform the same steps by taking the necessary information from the received request and calculating its corresponding signature. The calculated signature is then compared with the signature that was attached to the received request. If the signatures are the same, the request will be processed, otherwise the request is denied.

B. AWS Server Side Behaviour

This section describes more in-depth how AWS handles requests to their API endpoint and verifies if a request is valid.

After the server receives a request, it first checks the HTTP version. Both HTTP/1.1 and HTTP/1.0 are supported. The server continues with checking the action and the version that are specified in the request. Then it verifies if the mandatory parameters of Signature Version 4 are present. If they are all present it proceeds with checking whether the *X-AMZ-Date* is in the correct format (ISO8601 Basic Format). It continues by verifying if the request contains all the headers that are specified in *Signed Headers*. The server proceeds with checking whether a valid signing algorithm is selected. This is done by looking at the value that was provided in *X-AMZ-Algorithm*. It expects either *AWS4-HMAC-SHA1* or *AWS4-HMAC-SHA256* as its value. A valid signature can be created, regardless of the capitalization of this variable. This means that the values *AWS4-hMac-sha1* and *AWS4-HMAC-SHA1* give different signatures for the same requests, but both are valid. Next, it compares the date format of the credential scope and the date that is contained in the *X-AMZ-Date* header, to see if they are the same. The server will proceed in checking whether the rest of the credential scope is correct. That is, the region, service and ending string *aws4_request*. The server then proceeds with checking the user key ID, to see if a secret access key with such an ID does indeed exist. Finally, if all previous checks were successful, it calculates the signature from the values in the request and compares it with the signature provided in the request.

C. Amazon IAM and Amazon S3

In this research, we will be sending API requests to the Amazon Identity and Access Management (IAM) service and to Amazon Simple Storage Service (S3). The IAM service allows users to create and manage AWS users and groups, and use permissions to allow and deny their access to AWS resources [20]. The S3 service is an object storage service. It allows users to store and manage data by creating buckets and uploading data to these buckets [21].

D. Differences SigV4 and Escher

As mentioned earlier, Escher is based on SigV4. However, there are some minor differences between the SigV4 protocol and the Escher protocol. Escher has been implemented in various programming languages. In the case of this research, we make use of the Python wrapper for the Golang implementation. As there are various implementations, this also results in minor differences between each of the Escher implementations. The logic of the protocol, however, is identical across the different implementations. The majority of them are compatible with AWS' SigV4. However, the Escher implementation that we have used is not compatible with SigV4, due to some minor differences in the source code. This was not known a priori when the Escher implementation was chosen.

One of the differences between SigV4 and the Escher implementation that we have used, is that both protocols support different HMAC algorithms. SigV4 supports the use of HMAC-SHA1 and HMAC-SHA256, while Escher supports HMAC-SHA256 and HMAC-SHA512 by default. Header names differ as well. Headers in Escher contain the prefix *x-ems*, while with SigV4 they have *x-amz* as a prefix. The Escher library itself supports changing one to the other. Another difference is the variable name that SigV4 and Escher use to add the credentials needed to create the signature. SigV4 adds a *Credential* variable, while the Escher implementation that was used adds a *Credentials* variable. Looking at the source code of other Escher implementations, we have seen that *Credential* is used. As Escher is an open source project, it would be trivial to resolve these differences.

IV. Methodology

As SigV4 is a proprietary protocol, it was not possible to implement the protocol locally ourselves. In order to analyze the SigV4 protocol we set up a local stack which implements Escher [22]. The local stack is set up by first creating an API endpoint by writing an application in Python using Flask [23]. For that API endpoint we use the Escher Library for validating the request. Then using uWSGI, we made the endpoint reachable via the internet by using a port and a public IP address and bound this to the application. This local stack is created because we wanted to perform attacks on our local servers first, where applicable, before performing them on AWS. This was to minimize interference with AWS normal operations.

We define four different existing attacks in order to evaluate the security of SigV4. The four attacks are replay attack, modifying the request, HTTP smuggling, and a timing attack.

The attacks should undermine at least one of the features that SigV4 provides: verification of the identity of the requester, in-transit data protection, and protection against reuse of the signed portions of a request.

A. Replay Attack

Being able to replay requests would go against the fact that SigV4 should protect against reuse of signed portions. For this reason we performed a replay attack.

As mentioned in Section I we assumed that TLS is not used. This would allow us to perform a Man In The Middle attack (MITM) whereafter we could perform a replay attack. For this attack we used the Burp Suite software [24]. We set up a proxy through which all requests from the requester were sent. The proxy was bound to the localhost IP address 127.0.0.1 and was listening on port 5000.

The API requests were first sent to the proxy. In the Burp Suite software, we moved the API request to the *repeater* which allowed us to send identical request as often as needed. The requests from the proxy were then sent to the destined API endpoint. Once an API request is received, the destined server will respond with a response and corresponding HTTP status code. The destined server described in this paragraph is either our local server or AWS.

The AWS API endpoints that were chosen to send requests to, were from the AWS S3 and IAM services. We have chosen different services as there are minor differences in SigV4 between various services. We used the example code written in Python from the AWS documentation [25] as our base for creating a signed API request manually.

In this attack we have looked at the possibility of performing a replay attack and for how long one is able to perform a replay attack if it were possible. This was done in two steps. First, we looked if it was possible to send the exact same request twice in a small time window. This time window was no longer than 10 seconds. After we had verified that it was possible, we sent the request every minute to the AWS server and examined what the AWS response was. We tried to send many different types of requests. That is, both read and write requests. For example, for the IAM service we use the *GetGroup* and *CreateGroup* request. For the S3 service we use the *GetObject* and *PutObject* requests. In the documentation of AWS it is noted that the *X-AMZ-Expires* parameter can be used to specify for how long a request is valid in seconds [18]. This parameter is exclusive to query string requests. We have also performed a replay attack for requests containing this parameter.

B. Modifying requests

If modification of requests were possible, it would mean that data integrity, which SigV4 should provide, is not realized.

In order to modify requests we used the same setup for performing a MITM attack as described in the replay attack. After sending the request from the proxy to the *repeater*, we modified various headers and parameters to see if it is possible to modify the requests in any way. We did this for both read and write requests. Similar to the replay attack, we first performed the experiment on our local server, before trying to modify the requests destined for AWS. In the case of AWS, the modified request were sent to API endpoints of the IAM and S3 services.

As mentioned in section III, the S3 service supports the use of unsigned payloads. Thus, it should be possible to modify the payload without it affecting the signature. We also tried

modifying the payload for these types of requests and look at the impact that changing the payload could have on the request.

C. HTTP smuggling

For an HTTP request smuggling attack, one encapsulates a single or multiple requests into a single request. This can be achieved by placing requests inside the payload of another request. The request, containing another request in its payload, is then sent to a target server. If the architecture of the API endpoint is poorly designed, one may be able to execute this attack to successfully bypass security, obtain unauthorized access, compromise other users request, and potentially introduce cache poisoning.

Websites nowadays usually consist of a chain of systems. Multiple requests are aggregated at the front-end and are then sent to the back-end on a single connection for further processing. Figure 2, shows this graphically. The back-end distinguishes requests by looking at the headers. This architectural design also means that the back-end and front-end should have the same request boundaries. If this is not the case, one is able to execute an HTTP request smuggling attack. Figure 3, shows a variant of HTTP request smuggling, where the attacker can hijack another user's HTTP request. As, in this case, there is a discrepancy between the front-end and the back-end, the attacker can trick the back-end into believing that the boundary of a request is different by manipulating its headers.

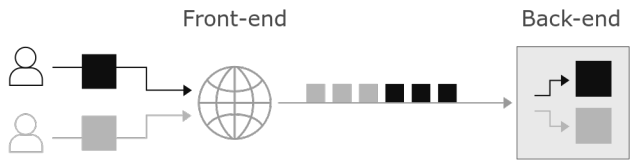

Figure 2. The request flow of modern website architecture from client to front-end server, and from the front-end server to the back-end server. Source: [14]

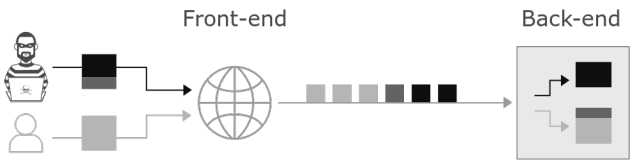

Figure 3. Example of an HTTP request smuggling attack. Source: [14]

As HTTP request smuggling relies on there being a discrepancy between the front-end and back-end, it implies that the attack is design specific. Thus, to perform this attack on our local servers first, before performing them on AWS' API endpoint, is of no use. We will thus only execute this attack on AWS' servers. The nature of this attack, however, is very intrusive and disrupting to other users. To prevent causing any disruption, we have followed the paper by James Kettle that was mentioned in Section II, in which he describes a method on how to execute HTTP smuggling attacks [14]. The method consist of a number of steps, with which HTTP request smuggling vulnerabilities can be found and attacks can be performed. These steps are: detect, confirm, explore, store and attack. Any step, other than the *detect* step, may introduce side effects for other users. Therefore, we have limited ourselves to only perform the *detect* step to limit the disruption we otherwise may cause to AWS' API endpoints and AWS' users.

This attack is similar to the modifying request attack from the previous subsection. The set up for intercepting and modifying requests is identical. We created a legitimate request with a payload and sent it to the Burp suite. The payload is created in a way, which would exploit the discrepancy between the front-end and back-end server, if there is any. We then sent the request to the repeater to finish crafting our HTTP smuggling request.

We have created an HTTP smuggling request for detecting if the AWS servers are susceptible to HTTP smuggling. The request can be seen in Listing 1. This request is used to send to AWS' IAM API endpoint. However, we have also tested a similar request for the S3 service. We do not list this, as the basics shown in Listing 1 remain the same. The basics will be explained in the next paragraph.

In order to exploit a potential discrepancy between the front-end and back-end, the request makes use of both the *Content-Length* and *Transfer-Encoding* headers. Normally the *Content-Length* header is used to determine the length of the payload. While, if the *Transfer-Encoding: chunked* header is used, the payload is divided into chunks. In this case, the length of the chunks is specified in the payload itself. A discrepancy is present if the front-end looks at the *Content-Length* header to determine the length of the request, while the back-end looks at the chunk size of the payload to determine the length, or vice versa. We can manipulate the *Content-Length* value and the chunk size value in such a way that the front-end and back-end do not agree on the boundary of a request.

The request that is constructed, is shown in Listing 1. The exploitation of this request works as follows. If the front-end server takes the *Content-Length* in consideration, then the request until the *G* is sent by the front-end server to the back-end server, while the back-end will wait for the next request, due to the *Transfer-Encoding: chunked*. If this is the other way around, and the front-end takes the *Transfer-Encoding: chunked* in consideration, then the front-end will reject the request, due to the invalid chunk size *Q*. Finally, if both the front-end and back-end are in sync, then they would either both use *content-length* or *Transfer-Encoding: chunked*. The result would be that either the request gets rejected at the front-end or processed by both the front-end and back-end.

We constructed the request in Listing 1, by first creating a legitimate request. This is the constructed request without *Transfer-Encoding: chunked* and the *Q*. We then sent then intercept the request with Burp, and move it to the repeater. In the repeater, we added the *Transfer-Encoding: chunked* and *Q*

to complete the construction of the HTTP smuggling request.

We have also tried to obfuscate *Transfer-Encoding* by placing white spaces or tabs in and around the transfer encoding. By doing this, it may be possible to use the *Transfer-Encoding: chunked* header even though a request with this header may otherwise be declined.

D. Timing attack

A timing attack is the general term for attacks where one analyzes the time it takes to execute an algorithm in order to reveal information about a cryptosystem. In our attack we focus on looking if we could perform a timing attack to gain information about the signature. By examining the execution time of the algorithm which is used to compare the signature, it may be possible to recreate the signature without knowing the key. This would allow an attacker to forge any request. This could be achieved by sending the same request with different signatures, and looking at which signature took the longest to compare. Comparison functions normally perform a byte-wise match between two arrays. If a bit does not match, the comparison would terminate early which would result in a shorter execution time. If a bit was guessed correctly, however, more bits would be compared which results in a longer execution time. Such an attack can be prevented by using a comparison function that is always executed in a constant-time.

To find out if such an attack is possible, we have conducted an analysis on the number of correct consecutive bits versus the response time of the server. We look at the response time, as an attacker is not able to measure the execution time of the comparison. This analysis, however, requires to send many requests, which could potentially flood the destined server. AWS prohibits flooding of their AWS services under their customer service policy [26]. Therefore, we were only able to perform this analysis on our local server, which uses Escher.

In order to perform this analysis, we first created a request with a legitimate signature, which consists of 256 bits. We then performed a series of steps. In the first step, the last bit of the signature is set to an incorrect value. In the second step, the last and the penultimate bits of the signature are set to an incorrect value, and so on. We then measured the response time for a request with a correct signature and requests with signatures containing one or more consecutive incorrect bits. The assumption was that if the signature is correct, it would also mean that the comparison of the signature would take longer amount of time. So we expect to see a decline in execution time when the signatures are compared with each other as more bits of the signature are set to an incorrect value.

Figure 4 shows how an incorrect signature is created. Figure 5 shows the first step of the analysis, in which the last bit of the signature is set to an incorrect value. As can be seen, this is achieved by performing an XOR operation with a bit string containing a 1 at the end. This bit string has the same length as the signature.

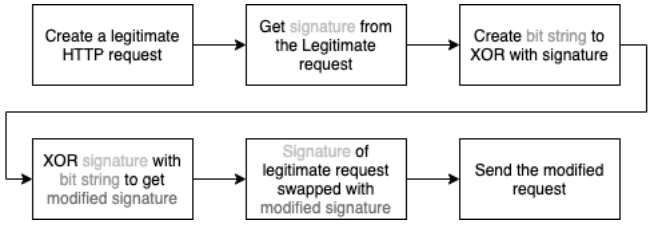

Figure 4. The flowchart for creating a request with an incorrect signature from a correct one.

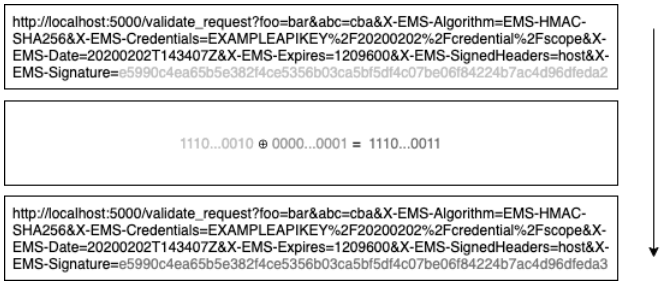

Figure 5. Example of how a request with the correct signature is converted to a request where one bit of the signature is incorrect.

After creating an incorrect signature, we would use the same legitimate request, but change the correct signature with the incorrect one. The request was then sent to our local servers. This was done 100 000 times, because a lot of repetitions would stabilize the mean and standard deviation and therefore the statistical inference would be more reliable. We did not do this more often, as it would otherwise take too long to execute. After this step we would create the next bit string to create the next wrong signature.

In order to send the requests, we made use of the *Requests* library in Python. Using the *elapsed* object from the response class, we were able to measure the amount of time elapsed between sending a request and the arrival of the response. The measured elapsed time is then used to see if there is a correlation between the number of incorrect bits in the signature and the response time.

To see if there is a correlation between the response time and the number of incorrect bits in the signature, we define a significance level α of 0.1. H_0 is defined as there being no correlation between the response time and number of incorrect bits in the signature. We then calculate the p-value. If it lies under the significance level, we reject H_0. This indicates that there is indeed a correlation between the two parameters. If not, we do not have enough evidence to reject H_0.

V. Results

This section consists of the results from the experiments that we have defined in section IV.

A. Replay Attack

We have verified that replay attacks are possible with requests of both the IAM and S3 services that are sent to AWS. The same request can be replayed multiple times and

```
POST /?Action=ListUsers&Version=2010-05-08 HTTP/1.1
Host: iam.amazonaws.com
User-Agent: python-requests/2.22.0
Accept-Encoding: gzip, deflate
Accept: */*
Connection: close
X-Amz-Date: 20200203T111158Z
Authorization: AWS4-HMAC-SHA256 Credential=<AMAZON_API_KEY_ID>/20200203/us-east-1/iam/aws4_request,
SignedHeaders=host;x-amz-date,
Signature=25495f300ab93c6f86d78c9dd4a76e071574d153b2f08273ff0016c1f8d91009
Content-Length: 4
Transfer-Encoding: chunked

0

G
Q
```

Listing 1. The detecting HTTP smuggling request that we have used to test on AWS'API endpoint.

will be processed by the server. We have verified this for both read and write requests. We have found that AWS has a time limit on how long a request is valid, depending on the service. With the IAM service, a request is valid for 15 minutes.

As described in Section IV, the *X-AMZ-Expires* can be used to specify for how long a request is valid. This parameter is limited to query strings. We found that IAM ignores this header altogether. Therefore, with IAM, a request can be valid for at most 15 minutes. Other services may accept it and use it, however. The AWS S3 service, for example, does support the use of the *X-AMZ-Expires* parameter. For S3 service requests, the time window in which it is possible to replay requests can thus be larger, if the *X-AMZ-Expires* is used.

B. Modifying requests

We have found that requests can partly be modified. Only the parameters and headers that are not relevant for the SigV4 and the API action itself can be changed. For example, one is able to add or modify the *Content-Length* header, however this does not implicate the requester. One is able to implicate the requester by adding the *Transfer-Encoding: chunked* header.

Any parts of the request that are signed cannot be modified, as it would result in a mismatch of signatures. Some API request allow additional parameters, such as the *X-AMZ-Expires* parameter mentioned in the previous subsection, however these parameters need to be added to the request URL, which is also signed. Changing these parameters would thus result in a mismatch of signatures as well.

As S3 supports the use of unsigned payloads, it is possible to modify the payload without it resulting in a different signature. While S3 supports this function, IAM does not.

C. HTTP smuggling

We have found that if the *Transfer-Encoding: chunked* header is added to a request, it would result in a response with an HTTP status code 500 "Internal Server Error" for the IAM service. As for S3, the server does not respond anything at all but the connection was terminated whenever *Transfer-Encoding: chunked* was used. As mentioned in the methodology, we have also tried to obfuscate the *Transfer-Encoding: chunked*. Below, the attempts to obfuscate this header, are shown:

1) Transfer-Encoding: chunked
2) Transfer-Encoding:[Tab]chunked
3) Transfer-Encoding[Tab]:[Tab]chunked
4) Transfer-Encoding: Chunked
5) Transfer-Encoding : chunked
6) Transfer-Encoding: chunked x
7) Transfer-Encoding: chunkedx
8) Transfer-Encoding: xchunked

The results from trying to obfuscate the *Transfer-Encoding: chunked* with IAM API endpoints was that either, the API endpoint would respond with an HTTP status code 500 "Internal Server Error" or with an HTTP status code 501 "Not Implemented". The results for the S3 service was either a response with an HTTP status code 501 or there would be no response at all.

D. Timing attack

The result of this attack is a graph which shows the number of incorrect bits of the signature on the x-axis of the graph and the response time in microseconds on the y-axis. When the graph is read from left to right, the first data point is the average response time for a request with a correct signature. The right-most data point of the graph is the average response time for a request with a signature which is completely wrong compared to the correct one. Moreover, from the data we have calculated both the correlation coefficient and the p-value to determine if a timing is attack would be feasible.

As described in Section IV, we calculate the p-value to see if it is justified to state that there is a correlation between the number of incorrect bits in the signature and the response time. The p-value has a value 0.0577, which is lower than the significance level of 0.1.

VI. Discussion

This section discusses the results from section V.

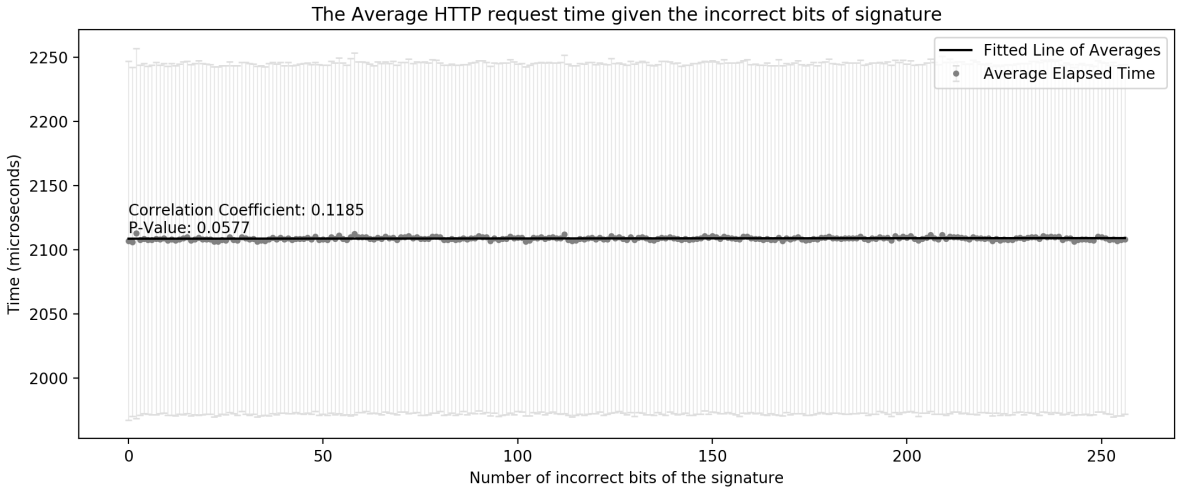

Figure 6. The graph shows the average time it took for a client sending a request and receiving an response given the number of wrong bits of the signature of the request. For every average 100 000 requests were send.

A. Replay Attack

SigV4 on itself is susceptible to replay attacks, as described in section V. Worth noting was that this was already warned in the documentation. In practice, it is only possible to replay requests under a condition. That is, the request that is to be replayed should be sent with HTTP. If HTTPS is used, the attacker needs to break this first, before it can start replaying the requests. This is because nonces are used in TLS which would prevent replaying of requests.

AWS has disabled HTTP connections for the IAM service. It only accepts HTTPS connections. The S3 service does support the use of HTTP, however. A replay attack on S3 services is thus possible in practice.

It is unknown if other services also support the use of HTTP, as we have not tried to send requests to other services. We have also not looked into the possibility of attacking TLS, or looking into the cipher suite that is used by AWS, as this was outside of our scope. However, there are known attacks against TLS, such as down-grade attacks, where the use of a weak cipher suite can be exploited [27].

As mentioned in Section IV we have looked at the possibility of replaying both read and write requests. Replaying read requests could result in an attacker gaining private information. For example, when a *GetObject* request, which is part of the S3 service, is replayed, an attacker would be able to read files without having the permission to do so. It may also be possible for an attacker to perform operations that another user has to pay for. Writing objects to a bucket, for example, costs money. If a write request, such as *PutObject*, which is part of the S3 service, is replayed many times, an attacker could increase the expenses of the original owner of that bucket.

B. Modifying requests

As seen from section V, no significant modifications can be made to the request because the significant headers and parameters are signed. As for completely changing the request, one needs to know the secret access key or signing key in order to forge a new request.

While, normally, no significant modifications can be made, it is possible to change the payload of requests when the original sender specified in the request that the payload should not be taken into account when creating the signature. This is done by providing the *UNSIGNED-PAYLOAD* parameter. This feature is supported by S3 but not by IAM. As S3 supports HTTP connections, it is possible that an attacker could intercept such an unsigned payload request, change the payload and send it to an S3 API endpoint. Depending on the request that was intercepted, it could allow the attacker to overwrite objects, delete objects, or change policies of a bucket, among other things.

C. HTTP smuggling

Although we have not successfully been able to send HTTP smuggling request to AWS, it is still crucial that we have performed this attack to see if AWS' API endpoints are protected against these kinds of attacks. The attack was still in the scope of this research as the attack could be used to circumvent the authentication of users. The front-end server of the AWS API endpoints that we have tested are not vulnerable against the HTTP smuggling attack described in Section IV. The IAM API server would respond with an HTTP status code of 500 whenever it would the *Transfer-Encoding: chunked* header was used. The S3 API server, would not send back a response but instead terminated the connection. This did not reveal if the S3 was susceptible to HTTP smuggling attacks.

The HTTP status code 501 occurred whenever we tried to obfuscate the chunked directive by placing an arbitrary x before the directive. Placing white space around the chunked directive resulted often in a bit slower response from the IAM API server, however the response HTTP status code was still

500. As for S3, there was no response at all but a termination in the connection. Therefore, from the observations that we have made, we assume that the AWS' API front-end server is secured against HTTP smuggling attack.

D. Timing attack

As stated in Section V, the p-value is lower than the significance level. We thus reject our initial hypothesis, which stated that there is no correlation between the number of incorrect bits in the signature and the response time. From our results in Figure 6, we saw that there is a weak positive correlation of 0.1185. We expected there to be a negative correlation, however. As explained in Section IV, we expected to measure a faster response time when the number of incorrect bits was higher, as the comparison function would then terminate earlier. While the correlation that was found was indeed found to be significant, we cannot confirm that a timing attack would actually be possible, as the results are in the order of microseconds.

One may argue, that our assumptions on how the signatures are being compared and read may have been incorrect. We have taken this into account when we had set up the experiment for testing. We noticed that the Escher implementation that we have used, used an unequal comparison operator for comparing its own calculated signature from the request and the signature of the request. This made us look deeper into how Golang comparison would work on a low level. As far as we can tell, Golang uses the system native endianness for reading bytes on a low level. The system that we have tested on, used little endian as its native byte order. From our example signature modification in figure 5, the bit that we had changed would be the last bit to be read due to the system's endianness.

Instead of looking at the response time, a different metric could have been measured, such as CPU cycles. However, an attacker would not have any other measurement unit available to them besides the response time. A timing attack normally would be executed by an attacker by sending multiple requests to see if there is a discrepancy in time depending on the variables in the request being changed.

VII. Conclusion

In this research, we have looked at various attacks in order to evaluate security of the SigV4 protocol. This protocol is used by AWS to provide verification of the identity of the requester, in-transit data protection, and protection against reuse of the signed portions of a request sent to a service's API endpoint. Each AWS' service has subtle differences between them, but the basic workings are the same. In this research, we have looked only at the IAM service and S3 service. We saw that the protection of the SigV4 protocol is dependent on HMAC-SHA1 or HMAC-SHA256. There exist no feasible attacks on these two algorithms, however. Therefore, we have looked at other ways to attack or circumvent the SigV4 protocol. We have specified attacks which would undermine at least one of the aforementioned three points that the SigV4 protocol should provide. We have looked at replay attacks, modifying of the request, HTTP smuggling, and timing attacks.

We have found that replay attacks are possible for both the IAM and S3 service within a specific time window. By default, replaying requests is possible within the first 15 minutes of the request being signed. Normally, replaying of requests is prevented by using SSL/TLS. The S3 service, however, allows one to send request without SSL/TLS.

For modifying requests, we were not able to make any significant changes to the request. All important data for the request is signed, and cannot be tampered with. However, with S3 service, there is an option to not sign a payload. When this option is used, the payload is susceptible to modification.

We have looked into the possibility of performing an HTTP smuggling attack to circumvent the SigV4 protocol. We had crafted requests with which we could detect if HTTP smuggling is possible. These requests elicited a 500 or 501 HTTP status code response or no response at all. The requests were thus not accepted, indicating that HTTP smuggling is not possible.

For the timing attack we were only able to make an analysis for our local server, which makes use of Escher, as we were not allowed to flood the AWS servers. We found a weak positive correlation between the number of incorrect bits in the signature and the response time. This could indicate that, by looking at the response time, one can gain information about the signature.

Revisiting our research question, we will answer them in a concise manner. *How does Signatures Version 4 Protocol ensure data integrity, verification of the requesting user, and protection against reuse of signed requests?*

The SigV4 protocol ensures data integrity by creating a signature. The integrity is only limited to the things that one has specified to sign. The protocol authenticates a user, by recreating the signature from data in the request and compare it with the signature of the same request. It verifies a user by taking the API Key ID and the corresponding secret access key to recreate the signature. This secret access key is bound to a specific user. Finally, the protocol protects against reuse of signed request due to expiration time for a request.

What kind of attacks are able to undermine data integrity,verification of the requesting user, or protection against reuse of signed requests?

The attacks that we have conducted that were able to undermine one of the three points in the research question are, replay attacks, modifying the request and possibly timing attack. Replay attacks are able to undermine the protection against reuse signed request. The modifying request attacks is able to undermine the data integrity. This is only true for the specifically specified *UNSIGNED-PAYLOAD* option from S3 services. Timing attacks may be able to undermine the verification of the requesting user. Our timing experiment only gave an indication of a possibility that a timing attack may be able on Escher. We could not confirm if this is also true for the SigV4 protocol.

We can conclude that the SigV4 protocol is secure as long the user creating the request acts carefully. The practical attacks that we have found (replay attack and modifying the request), are mitigated if HTTPS is used instead of HTTP, to send the requests.

In conclusion, the SigV4 protocol does provide data-integrity for the portions of the requests that have been signed. It also verifies the requesting user. It only partly protects against reuse of signed requests, however as replaying requests is possible within a specific time window.

VIII. Future Work

As we have seen, there are subtle differences between AWS services. Further research could be done into what differences there are between the various services that AWS provides. Looking for vulnerabilities in services other than IAM and S3, which we have looked at in this research, could lead to finding service-specific vulnerabilities in the SigV4 protocol.

Performing a timing attack on AWS services could be tried as well. In this research, we have analyzed the possibility of performing a timing attack on our local server, which ran Escher. This was done because flooding of AWS servers is prohibited. Permission to perform flooding attacks for penetration testing can be requested, however.

Finally, attacks on HMAC-SHA1 are considered infeasible but have proven to be possible. These attacks could become feasible in the future, and may become a valid attack vector on the SigV4 protocol.

Acknowledgment

We would like to thank our supervisors Alex Stavroulakis and Aristide Bouix from KPMG N.V., for their time, guidance and feedback throughout this work. Not only have they helped us with the research and and writing of this paper, they have also helped us with giving feedback for our presentation. We are grateful that we have been able to do our research at KPMG as we have enjoyed our time working on this paper.

References

[1] Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. August 2018. DOI: 10.17487/RFC8446. URL: https://rfc-editor.org/rfc/rfc8446.txt.

[2] Eran Hammer-Lahav. *The OAuth 1.0 Protocol*. RFC 5849. April 2010. DOI: 10.17487/RFC5849. URL: https://rfc-editor.org/rfc/rfc5849.txt.

[3] Dick Hardt. *The OAuth 2.0 Authorization Framework*. RFC 6749. October 2012. DOI: 10.17487/RFC6749. URL: https://rfc-editor.org/rfc/rfc6749.txt.

[4] Professor John Franks et al. *HTTP Authentication: Basic and Digest Access Authentication*. RFC 2617. June 1999. DOI: 10.17487/RFC2617. URL: https://rfc-editor.org/rfc/rfc2617.txt.

[5] *Authenticating Requests (AWS Signature Version 4)*. Visited on: 08-01-2020. URL: https://docs.aws.amazon.com/AmazonS3/latest/API/sig-v4-authenticating-requests.html.

[6] Colin Percival. *AWS signature version 1 is insecure*. Visited on: 09-01-2020. 2008. URL: http://www.daemonology.net/blog/2008-12-18-AWS-signature-version-1-is-insecure.html.

[7] Jeff Bar. *Amazon S3 Update - SigV2 Deprecation Period Extended Modified*. Visited on: 09-01-2020. 2019. URL: https://aws.amazon.com/blogs/aws/amazon-s3-update-sigv2-deprecation-period-extended-modified/.

[8] Luigi Lo Iacono and Hoai Viet Nguyen. "Authentication scheme for REST". In: *International Conference on Future Network Systems and Security*. Springer. 2015, pp. 113–128.

[9] Mark Cavage and Manu Sporny. *Signing HTTP Messages*. DRAFT. IETF, October 2019, pp. 1–22. URL: https://www.ietf.org/id/draft-cavage-http-signatures-12.txt.

[10] Dr. Hugo Krawczyk, Mihir Bellare, and Ran Canetti. *HMAC: Keyed-Hashing for Message Authentication*. RFC 2104. February 1997. DOI: 10.17487/RFC2104. URL: https://rfc-editor.org/rfc/rfc2104.txt.

[11] Chowdhury Sajadul Islam and Mohammad Sarwar Hossain Mollah. "Timing SCA against HMAC to investigate from the execution time of algorithm viewpoint". In: *2015 International Conference on Informatics, Electronics & Vision (ICIEV)*. IEEE. 2015, pp. 1–6.

[12] Pierre-Alain Fouque et al. "Practical electromagnetic template attack on HMAC". In: *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer. 2009, pp. 66–80.

[13] Ronen Heled. "HTTP REQUEST SMUGGLING". In: (2005).

[14] James Kettle. "HTTP Desync Attacks: Request Smuggling Reborn". In: (2019). Visited on: 30-01-2020. URL: https://portswigger.net/kb/papers/z7ow0oy8/http-desync-attacks.pdf.

[15] *AWS Command Line Interface*. Visited on: 10-01-2020. URL: https://aws.amazon.com/cli/.

[16] *AWS SDK*. Visited on: 10-01-2020. URL: https://aws.amazon.com/tools/.

[17] *Signing AWS Requests with Signature Version 4*. Visited on: 10-01-2020. URL: https://docs.aws.amazon.com/general/latest/gr/sigv4_signing.html.

[18] *Authenticating Requests: Using Query Parameters (AWS Signature Version 4)*. Visited on: 29-01-2020. URL: https://docs.aws.amazon.com/AmazonS3/latest/API/sigv4-query-string-auth.html.

[19] *WWW FAQs: What is the maximum length of a URL?* Visited on: 30-01-2020. URL: https://web.archive.org/web/20190902193246/https://boutell.com/newfaq/misc/urllength.html.

[20] *AWS Identity and Access Management (IAM)*. Visited on: 31-01-2020. URL: https://aws.amazon.com/iam/.

[21] *Amazon S3*. Visited on: 31-01-2020. URL: https://aws.amazon.com/s3/.

[22] *Escher - stateless HTTP request signing library*. Visited on: 09-01-2020. 2019. URL: https://github.com/emartech/escher.

[23] *Flask — The Pallets Projects*. Visited on: 29-01-2020. URL: https://www.palletsprojects.com/p/flask/.

[24] *Burp Suite - Cybersecurity Software from PortSwigger*. Visited on: 22-01-2020. URL: https://portswigger.net/burp.

[25] *Examples of the Complete Version 4 Signing Process (Python)*. Visited on: 22-01-2020. URL: https://docs.aws.amazon.com/general/latest/gr/sigv4-signed-request-examples.html.

[26] *AWS Customer Support Policy for Penetration Testing*. Visited on: 08-01-2020. URL: https://aws.amazon.com/security/penetration-testing/.

[27] Eman Salem Alashwali and Kasper Rasmussen. "What's in a downgrade? A taxonomy of downgrade attacks in the TLS protocol and application protocols using TLS". In: *International Conference on Security and Privacy in Communication Systems*. Springer. 2018, pp. 468–487.