Lasse Kajavalta

REST API SECURITY: TESTING AND ANALYSIS

Master's Thesis
Faculty of Information Technology and Communication Sciences
Marko Helenius
Minna Vallius
April 2022

ABSTRACT

Lasse Kajavalta: REST API Security: Testing and Analysis
Master's Thesis
Tampere University
Advanced Studies in Information Security
April 2022

Application programming interfaces (API) are components that facilitate communication between other applications. APIs are used in various software systems but perhaps most commonly they are found in modern web applications. Web applications and the APIs they utilize are both attractive and easily accessible targets to malicious attackers. Therefore, security of these applications is paramount.

A lot of research is done on trying to map and combat common vulnerabilities in regarding web applications, but API implementations also have their own vulnerabilities. In this master's thesis, one of the primary goals was to find common API vulnerabilities by researching existing literature on the subject and performing in-depth security testing to figure out the level of protection M-Files Cloud Management API provides against these previously recognized vulnerabilities. The research questions of this thesis were to find the most significant vulnerabilities related to the security of API implementations, how these vulnerabilities apply to the M-Files API solution, and how the development process could be improved to ensure security in the future.

The API implementation tested during this thesis is that of M-Files Manage, a customer-facing web application that allows customers to manage their own M-Files Cloud environments and subscriptions. In this thesis, the system was tested against 9 well-known and commonly appearing vulnerabilities of API implementations. For each vulnerability, appropriate security testing was done. Depending on the type of vulnerability and how it can be tested for, testing was done manually, utilizing security testing tools, and by developing new test automation coverage for the code project.

During the testing, 10 security-related issues were found within the API. These issues were reported to the development team of the system and fixed within the API as a result. New improvement ideas for the API and its continuing development were presented, and as a result of this thesis the existing level of security for the API was examined and improved upon. Existing test automation coverage was also greatly improved to take into account many different security aspects.

Keywords: M-Files, REST, API, Security, Cloud, Web applications, Security testing

TIIVISTELMÄ

Lasse Kajavalta: REST-rajapinnan tietoturvan testaus ja analysointi
Diplomityö
Tampereen yliopisto
Advanced Studies in Information Security
Huhtikuu 2022

Sovellusohjelmointirajapinnat (engl. Application Programming Interface, API) ovat komponentteja, jotka mahdollistavat kommunikaation muiden ohjelmien välillä. APIa käytetään erilaisissa ohjelmistoratkaisuissa, mutta kenties yleisimmin niitä esiintyy modernien web-sovelluksien yhteydessä. Web-sovellukset ja niiden käyttämät rajapinnat ovat houkuttelevia ja helposti saavutettavia kohteita mahdollisille hyökkääjille, minkä vuoksi sovellusten turvallisuus on ensiarvoisen tärkeää.

Web-sovelluksiin liittyvien haavoittuvuuksiin liittyen tehdään merkittävästi tutkimustyötä, mutta myös API-ratkaisuilla on omat haavoittuvuutensa. Tässä diplomityössä yksi tärkeimmistä tavoitteista oli etsiä yleisiä API-haavoittuvuuksia tutkimalla aiheeseen liittyvää olemassa olevia lähteitä ja kirjallisuutta. Kartoitettujen haavoittuvuuksien perusteella työssä tehtiin perusteellista tietoturvatestausta M-Files Oy:n toteuttamaa API-rajapintaa vastaan.

Tässä työssä tutkittiin M-Files Manage -verkkosovelluksen taustalla toimivaa rajapintaa. M-Files Manage on web-sovellus, jonka kautta asiakkaat voivat itse hallinnoida M-Files tietovarastoja, pilviympäristöjä ja M-Files tilauksia.

Työssä perehdyttiin 9:ään yleiseen API-rajapintoihin liittyvään haavoittuvuuteen, joita varten suoritettiin tietoturvatestausta M-Filesin API-ratkaisua vasten. Haavoittuvuuden tyypistä riippuen testausta tehtiin manuaalisesti, tietoturvatestaukseen tarkoitetuilla työkaluilla sekä kehittämällä uutta testiautomaatiota API-ratkaisun koodikantaan.

Työtä varten tehdyn tietoturvatestauksen aikana löydettiin 10 tietoturvaan liittyvää ongelmaa M-Filesin rajapinnassa, jotka raportoitiin tuotteen kehitystiimille ja korjattiin työn seurauksena. Työn tuloksena M-Filesin API-rajapinnan tietoturvan tason nykyisestä tilasta saatiin parempi kuva, löydettiin olemassa olleita haavoittuvuuksia korjattavaksi ja täten nostettiin yleistä tietoturvan tasoa. Tuotteen testiautomaatiokattavuuteen tehtiin myös huomattavia parannuksia tietoturvaongelmien varalta.

Avainsanat: M-Files, REST, API, Tietoturva, Cloud, Web-sovellus, Tietoturvatestaus

PREFACE

This thesis was made at M-Files Corporation (M-Files Oy), and I would like to thank Product Security team lead Minna Vallius for providing the challenging topic as well as for all the help and feedback provided during the making of this thesis. Acknowledgements also to all those friends and colleagues who provided encouragement during the writing process.

Tampere, 18 April 2022

Lasse Kajavalta

CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

API	Application Programming Interface
ASVS	Application Security Verification Standard
CORS	Cross-Origin Resource Sharing
CSP	Content Security Policy
CSV	Comma-Separated Values
CVE	Common Vulnerabilities and Exposures
CWE	Common Weakness Enumeration
DAST	Dynamic Application Security Testing
DoS	Denial of Service
HATEOAS	Hypermedia as the Engine of Application State
HSTS	HTTP Strict Transport Security
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IAST	Interactive Application Security Testing
JSON	JavaScript Object Notation
JWT	JSON Web Token
OWASP	Open Web Application Security Project
PaaS	Platform as a Service
REST	Representational State Transfer
SAST	Static Application Security Testing
SCS	Secure Code Scan
SQL	Structured Query Language
SSL	Secure Sockets Layer
TLS	Transport Layer Security
URL	Uniform Resource Locator
XML	Extensible Markup Language
ZAP	Zed Attack Proxy

1. INTRODUCTION

As more and more new services have been developed and older services have adapted to utilize web-based solutions, the importance of modern web applications and application programming interfaces has grown tremendously. This comes with its own challenges from a security point of view. Due to the publicly available nature of web applications and APIs, they can be very attractive targets to malicious attackers.

API-related vulnerabilities may go overlooked as product development often tries to focus on what is visible to the end users rather than what happens in the background of an application. Ensuring the security of an application may also be a challenge as developers face critical deadlines or the risk of a vulnerability is not properly understood.

Just as many other types of applications, APIs can face a great number of different security vulnerabilities that may allow a malicious attacker access to sensitive data or functionality that they should not normally be allowed to access. In order to ensure the security of an API implementation and the sensitive data it processes; security should be a key aspect in testing and overall quality assurance of the application.

The goal of this thesis was to study what types of vulnerabilities exist in REST-based API solutions, how common they are, and how they can be tested for. To achieve these goals, the thesis describes different approaches to security testing, existing tools that can be utilized for API security testing purposes, and common API threats and vulnerabilities.

In this thesis, an existing REST API solution of M-Files Oy was analysed and tested against common API vulnerabilities. The purpose of this thesis was to perform large scale security testing on the M-Files API implementation and either ensure that the API is protected against common API vulnerabilities or discover issues and improvements that can be made to the API in order to ensure its security.

The primary research questions selected for this thesis were:

- What are the most significant vulnerabilities and attacks against REST API implementations?
- How is the M-Files API implementation protected against API vulnerabilities?

- How can the development process ensure the security of a REST API?

This master's thesis consists of several chapters that aim to fulfil the goals of the thesis and answer the research questions. Chapter 2 describes background information related to modern web applications and API solutions. In chapter 3, the focus is on providing background information on the M-Files system and the specific API implementation being studied as a key part of this thesis. Chapter 4 discusses the challenges regarding web application and API security. Chapter 5 goes into security testing and describes different approaches and tools that were studied and utilized during the testing done as a part of this thesis. During chapter 6 API-specific threats and vulnerabilities are discussed in detail, including how they relate to the M-Files API implementation. Chapter 7 will provide an overview of the results found during this thesis. Finally, conclusions and answers to the research questions will be provided in chapter 8.

2. WEB APPLICATIONS

This chapter will provide an overview of some of the common technologies used in web application and API development to serve as background information for a better understanding of the topics covered in this thesis.

2.1 HTTP

Hypertext Transfer Protocol (HTTP) is a widely used application-level request-response protocol used on the World Wide Web. The purpose of HTTP is to function as a means of communication between a client and a web server. Clients and servers exchange many different forms of data using the HTTP protocol: web pages, images, text files, videos, scripts, and many other types of files may be served by utilizing the protocol.

HTTP protocol supports many different request commands referred to as HTTP methods. Every HTTP request is associated with one of these methods that communicate to the server what type of action should be done to the requested resource. HTTP methods have their own defined purposes; however, their use is not limited in any way and application developers may differ from their original purpose. [1]

Table 1: HTTP Methods [2]

HTTP Method	Method description
GET	Request resource from server to client.
POST	Request server to process request payload data.
PUT	Replace target resource with request payload.
DELETE	Remove target resource.
HEAD	Request HTTP headers from target resource.
OPTIONS	Request communication options from server.

CONNECT	Establish a TCP connection tunnel to the server.
TRACE	Perform a message loop-back test to the target resource.

The HTTP protocol supports headers, which allow the client and server to exchange additional information along with a HTTP request or response. HTTP headers can be grouped into request, response, representation, and payload headers. Headers serve many purposes in HTTP communication. Request and response headers contain information about the resource itself and the client or server behind the message. Representation headers describe the information provided in the header, such as the type of data contained within the message or information related to its encoding or compression. Payload headers hold information related to content length and the encoding used in transporting the data. [3]

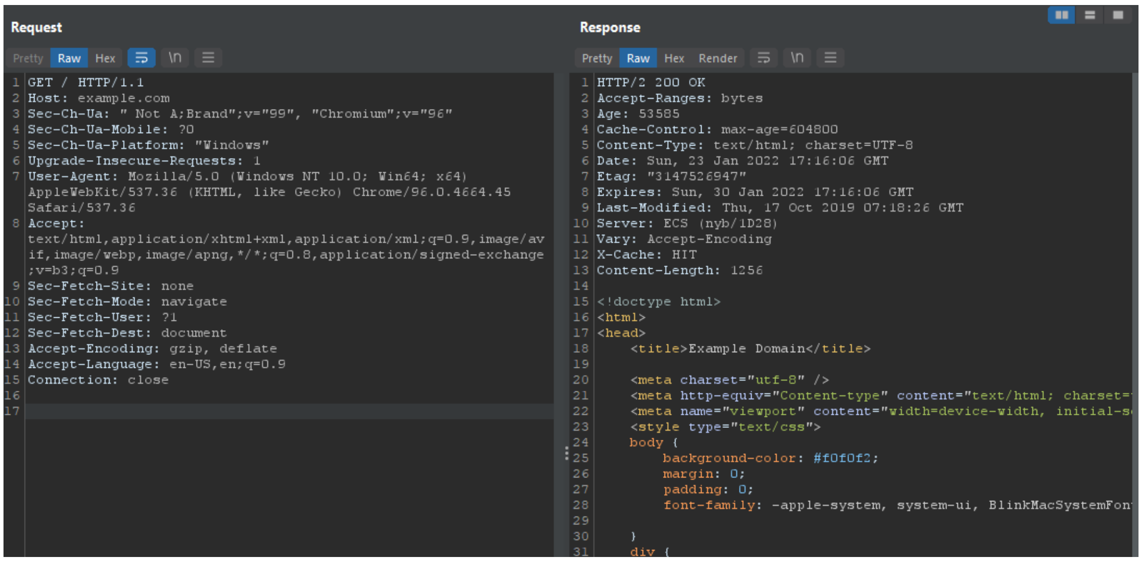
Table 2: Examples of HTTP headers

HTTP header	Description
Accept	Type of data that the client expects to receive from the server.
Content-Length	Byte size of the HTTP message sent to the recipient.
Content-Type	Media type of the resource.
Authorization	Credentials used to authenticate a client with the server.
Allow	List of HTTP request types accepted by a resource.

The HTTP protocol uses a set of defined status codes to indicate whether or not processing a request was successful. When a server returns a response to a client, it includes a HTTP status code in the first line of the response. The response contains a numeric code indicating if the request was successful or what type of error was encountered when processing the request. HTTP status codes are represented as a number to make error processing easier for programs, but the responses usually contain a human-readable error message as well [1]. The different ranges of HTTP status codes are shown in table 3. Each given range contains a multitude or defined error codes that describe the result of the request.

Table 3: HTTP status codes [2]

HTTP status code range	Description
100–199 Informational	The request was received and being processed.
200–299 Successful	The request was successfully received and accepted.
300–399 Redirection	The requested resource should be accessed through different means.
400–499 Client error	The request is invalid or cannot be fulfilled.
500–599 Server error	The server failed to process the request.



```
GET / HTTP/1.1
Host: example.com
Sec-Ch-Ua: " Not A;Brand";v="99", "Chromium";v="96"
Sec-Ch-Ua-Mobile: ?0
Sec-Ch-Ua-Platform: "Windows"
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/96.0.4664.45
Safari/537.36
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/av
if,image/webp,image/apng,*/*;q=0.8,application/signed-exchange
;v=b3;q=0.9
Sec-Fetch-Site: none
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Connection: close


```

```
HTTP/2 200 OK
Accept-Ranges: bytes
Age: 53585
Cache-Control: max-age=604800
Content-Type: text/html; charset=UTF-8
Date: Sun, 23 Jan 2022 17:16:06 GMT
Etag: "3147526947"
Expires: Sun, 30 Jan 2022 17:16:06 GMT
Last-Modified: Thu, 17 Oct 2019 07:18:26 GMT
Server: ECS (nyb/1D28)
Vary: Accept-Encoding
X-Cache: HIT
Content-Length: 1256

<!doctype html>
<html>
<head>
    <title>Example Domain</title>

    <meta charset="utf-8" />
    <meta http-equiv="Content-type" content="text/html; charset=
    <meta name="viewport" content="width=device-width, initial-s
    <style type="text/css">
    body {
        background-color: #f0f0f2;
        margin: 0;
        padding: 0;
        font-family: -apple-system, system-ui, BlinkMacSystemFon

    }
    div {
```

Figure 1: HTTP request and response

HTTP contains methods for providing authentication between the client and the server, such as cookies and the authorization header, however, it does not by itself secure the communication from eavesdropping and tampering by malicious third parties. Due to this, a secure form of HTTP called Hypertext Transfer Protocol Secure (HTTPS) has been developed and is today used in most web applications with its use growing over time [4]. With HTTPS, the request as well as the response are encrypted before being sent across the network. HTTPS utilizes either the Secure Sockets Layer (SSL) or Transport Layer Security (TLS) to encrypt the communication between the different parties. [1]

The purpose of HTTP is to ensure the confidentiality, integrity and to provide server authentication during the communication between the client and the server. Maintaining the confidentiality of the communication requires that any third parties listening to the communication cannot intercept plaintext communication between the client and the server. Integrity of the communication requires that any potential attackers are not able to modify the messages exchanged between the communicating parties. By providing server authentication, the client can be certain that the server they are communicating with is the one they intended to communicate with. [4]

In order to manage and enable the efficient utilization the different resources hosted on the internet Uniform Resource Locators (URL) are used. URLs act as the client's access point to HTTP and other protocols. They enable clients to find, use and share the vast amount of resources found on the internet. [1] All URLs follow the general structure of *<scheme>:<scheme-specific part>*, where the scheme describes how the specified resource is being accessed i.e., which protocol is used during the communication. The scheme-specific part is used to locate the resource and its format depends on the utilized scheme. [5].

URLs used with the HTTP protocol follow the format of

```
<protocol>://<hostname>/<path>
```

where *protocol* is either HTTP or HTTPS, *hostname* is the location of the server where the resource is hosted, and *path* is the resource path to the specific resource being accessed.

2.2 RESTful API

Application programming interface is a software interface that facilitates communication between other applications. APIs expose data and functions to other computer programs to provide some kind of service to them. A RESTful API is an API implementation that follows the architectural style referred to as Representational State Transfer (REST). The REST architecture describes six constraints around which should be applied when building a RESTful service: uniform interface, statelessness, cacheability, client-server architecture, layered system, and code-on-demand. [6]

The uniform interface constraint relates to the generalization of the component interface. This leads to a simplified system architecture and greater visibility of interactions. The REST architecture breaks down uniform interface into four guiding principles: identification of resources, manipulation of resources through representations, self-descriptive messages, and hypermedia as the engine of application state (HATEOAS). [6]

Statelessness is a key component of REST. In REST architecture, the server does not store state information when it communicates with a client. Each request from a client should contain all the necessary information the server needs for processing the request. This information can be sent through the URI, query parameters, request body, or headers. Statelessness improves upon a RESTful service's scalability and facilitates easier load balancing within the system. A drawback of statelessness is that if a client needs to send multiple requests, all state related information needs to be retransmitted with every request. [6]

Cacheability within REST refers to the responsibility of a service to define whether or not a client is allowed to cache its responses and reuse them at a later time instead of resending a request to obtain the cached information. The cacheability of REST architecture improves upon the efficiency and scalability of the service, as well as client performance of the application relying on the RESTful service. Cache constraints are important to be well-managed to prevent clients from sending unnecessary requests but not allow them to utilize outdated data. [6]

The client-server architecture constraint separates the RESTful service and the clients that rely upon it. Its goal is to simplify the architecture of applications and decouple the API service from the client service, allowing them to be developed separately while maintaining the compatibility of the two applications. This separation enables clients to not be concerned with data storage and the service doesn't need to know any specifics of the client implementation, such as the user interface or user state. [6]

A layered system allows the architecture to be composed of hierarchical layers, where each component can only see the other layer that they are interacting with. In a layered system, a client cannot tell if they are directly connected to the server or an intermediary server along the path to the actual service. The layered system constraint works to enable load balancing systems and maintaining security of different components within the system. A drawback to layered systems can be latency and increased overhead in data processing, leading to degraded performance. [6]

Code-on-demand is an additional optional constraint of REST architecture, which allows the extension of client functionality by allowing executable code to be downloaded from the service. This can come, for instance, in the form of Java applets or JavaScript. [6]

REST is not limited to a specific protocol, although it was designed with HTTP in mind, as it is the World Wide Web's primary transfer protocol. The REST architectural style can technically be utilized with any other protocol, as long it allows its constraints to be followed in the implementation. RESTful APIs may utilize various types of data formats in their requests and responses like JSON (JavaScript Object Notation), XML (Extensible Markup Language), CSV (comma-separated values), or plaintext. However, JSON is considered the most popular by far [7]. Modern web services commonly utilize the REST architecture in their API implementations, however, alternatives to it do exist. These alternatives include solutions such as RPC, SOAP and GraphQL.

In a relatively typical REST API implementation, a client application utilizes HTTP requests to retrieve data from the API service, which is then processed by the client application and represented in some way to the end user. This could be, for example, a web service that offers weather forecasts. The user navigates to the client and chooses a city to look up the current weather and forecast. To display the information, the web application sends an HTTP request to the weather API to which the API responds with JSON-formatted plaintext data, which is then further processed to be represented in a user-friendly way within the client application.

2.3 JSON Web Token

JSON Web Token (JWT) is a compact method of authorization and information exchange between parties, such as a server and a client application. JSON Web Tokens consist of a set of unique claims that can be used to exchange various types of information between the parties. Claims can contain any information that the application designers wish to exchange, however, there are a set of registered claims presented in the proposed internet standard. The format of a JWT is a JSON that is typically base64-encoded, however, while not popularly utilized, encryption can also be applied to JWTs. JSON Web Tokens can be signed through asymmetric cryptography, which allows the user to verify the entity that issued the token and the integrity of the contained claims. The issuer signs the content of the JWT with their private key, and it can be verified by decrypting it with their public key available to everyone. [8]

```
Base64-encoded token:
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJTdWJqZWN0IiwibmFtZSI6Ikphb-
mUgRG9lIiwiaWF0IjoxNjQ2NDg4ODc2fQ.i5AOep4Ld5MBrW_bL2a46_oEwSHKCmwGD-TdPdDXZls

Decoded token:
  {
      "alg": "HS256",
      "typ": "JWT"
  }
  .
  {
      "sub": "Subject",
      "name": "Jane Doe",
      "iat": 1646488876
  }
  .i5AOep4Ld5MBrW_bL2a46_oEwSHKCmwGD-TdPdDXZls
```

JSON Web Tokens consist of three parts: header, payload, and signature. The header of a JWT is typically used to identify the type of token, usually simply set to “JWT”, and the algorithm that was used in generating the signature. JWT signatures support three signature algorithms, HMAC SHA256, RSA and *none*, with the last one implying that no signature is expected to be presented with the token.

The payload of a JWT contains the customizable claims included in the token. According to the proposed standard, claims are separated into registered, public, and private claims. Registered claims are non-mandatory but recommended claims which include common information that is often expected to be presented in web tokens, such as information about the issuer, the subject and the expiration time of the token. Public claims are user defined claims that should follow the JSON Web Token registry in order to prevent implementation related collisions. Private claims are unregistered non-public claims which are wholly defined by the user. [8]

The last part of a JSON Web Token is the signature, which is used to verify the validity of the token and to confirm the identity of the issuing entity. The signing process takes the encoded header as well as the payload and using a secret and the header-specified algorithm and generates a signature which is then appended to the token.

JSON Web Tokens are commonly used as an authorization and data exchange mechanisms between a web application and a backend API service. The user authenticates to the service and is given a token which is then sent along with requests to the API. The API will verify the token’s validity by using the signature and authorize the user to content based on the information contained within the token.

3. M-FILES CLOUD MANAGEMENT API

The goal of this chapter is to provide a high-level view of the systems discussed in this thesis. The first subchapter will provide an overview of what the M-Files product is, and the second subchapter will describe the M-Files cloud solution and the Cloud Management API, which is the focus of this thesis.

3.1 M-Files Product

M-Files is an intelligent information management platform developed by M-Files Oy. M-Files enables users to easily store, organize, and access their information by utilizing a metadata driven approach to content management. By helping users organize their documents based on their properties, M-Files enables an efficient way for organizations to manage information and improve their day-to-day business operations. In addition to storing documents, M-Files enables the efficient search, sharing, processing, and collaboration with documents, as well as helping with organizational process automation, data security and compliance related matters, among other things. [9]

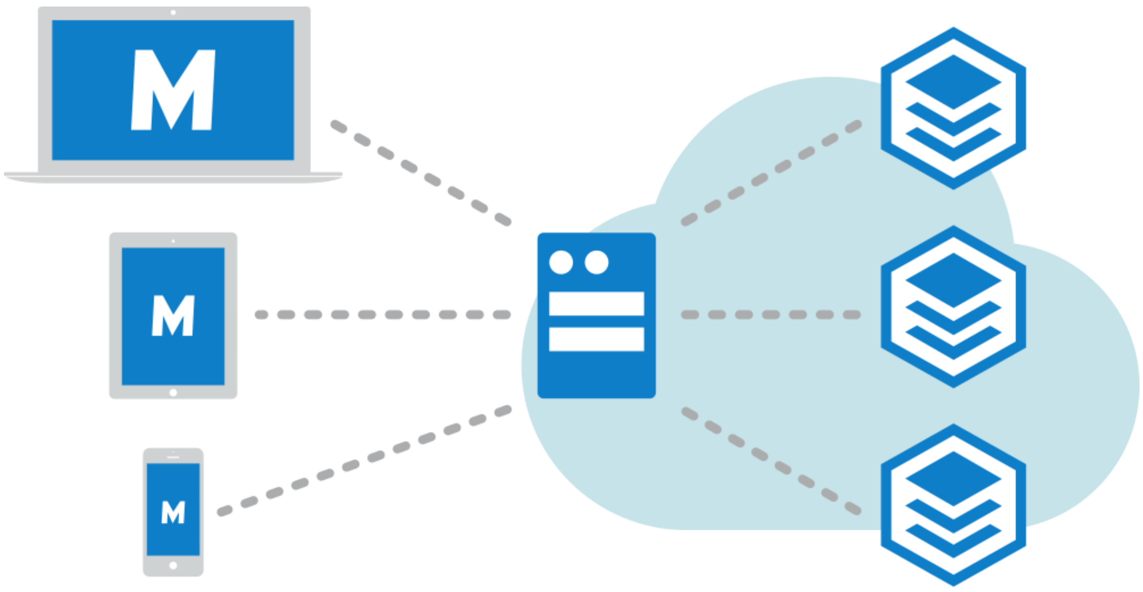


Figure 2: M-Files System Overview: client applications connect to a server, which can host multiple vaults. [10]

The core M-Files product consists of Web, Desktop and mobile client applications, M-Files Server, and M-Files Admin. The content in M-Files is stored in a vault, which resides on one or more M-Files Server instances. The content within vaults can be accessed by the various M-Files client applications. M-Files Admin is an administrative tool that can be used to manage M-Files server instances.

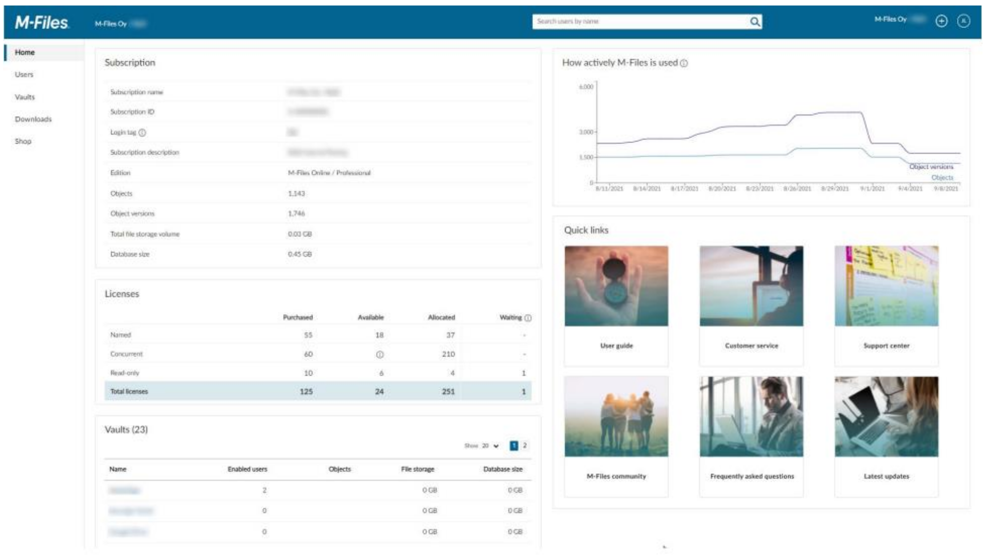
3.2 M-Files Cloud

M-Files is offered to customers as an on-premises solution as well as a cloud-based solution managed by M-Files Oy. M-Files Cloud offers the same experience as an on-premises server solution but with easy scalability and reliable performance provided by its Microsoft Azure-based cloud platform.

The M-Files Cloud solution is built upon Azure Service Fabric. Service Fabric is a distributed systems platform offered by Microsoft, where customers can package, deploy, and manage microservices and container-based applications. It offers an easy way for developers to build, publish and manage applications and scale them as needed. Service Fabric offers the possibility of running containerized applications within the customer's own on-premises data centres, Microsoft's or with any other cloud provider. [11]

Within the M-Files Cloud solution, M-Files Server instances are run as containerized applications within Microsoft's data centres. M-Files server applications also utilize cloud SQL (Structured Query Language) databases for data storage. M-Files maintains and manages these cloud resources for their customers and provides support for them. Utilizing cloud-based solutions, M-Files may easily scale up or down their cloud systems as new customers join or the load on an individual server goes up.

Manage is a single-page web application running in Microsoft Azure that M-Files customers can use to manage their subscriptions. Through M-Files Manage customers are able to edit vault users, allocate licenses and manage cloud vaults on their subscription. Additionally, Manage provides downloads for M-Files software and the ability for customers to purchase additional licenses for their subscription.



Subscription

Subscription name	[redacted]
Subscription ID	[redacted]
Login tag ⓘ	[redacted]
Subscription description	[redacted]
Edition	M-Files Online / Professional
Objects	1,143
Object versions	1,746
Total file storage volume	0.03 GB
Database size	0.45 GB

Licenses

	Purchased	Available	Allocated	Waiting ⓘ
Named	55	18	37	-
Concurrent	60	ⓘ	210	-
Read-only	10	6	4	1
Total licenses	125	24	251	1

Vaults (23)

Show 20 1 2

Name	Enabled users	Objects	File storage	Database size
[redacted]	2		0 GB	0 GB
[redacted]	0		0 GB	0 GB
[redacted]	0		0 GB	0 GB

Figure 3: M-Files Manage home page [12]

Manage utilizes a central database called OneDB for the secure storage of licensing, user and vault information of M-Files subscriptions. When customers edit their subscriptions through Manage, the changes are made and stored in OneDB.

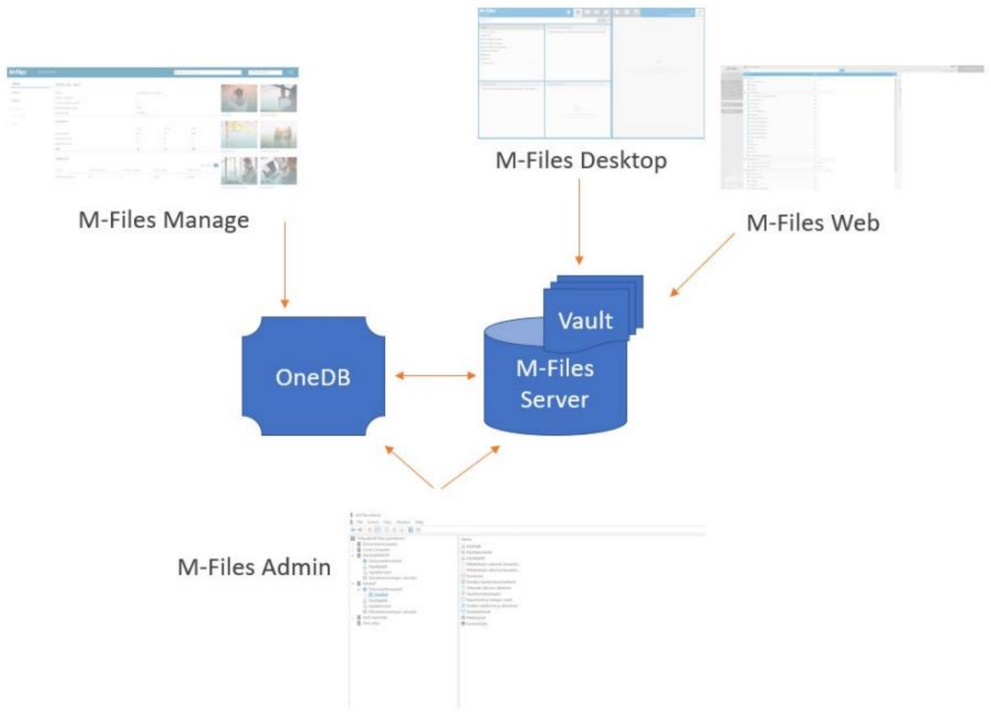


Figure 4: M-Files Clients [12]

Behind the single-page application of M-Files Manage operates the M-Files Cloud Management API, which is used for the communication between the Manage web client application and the OneDB database. The API is also utilized to communicate with the M-Files Server instances that host the document vaults.

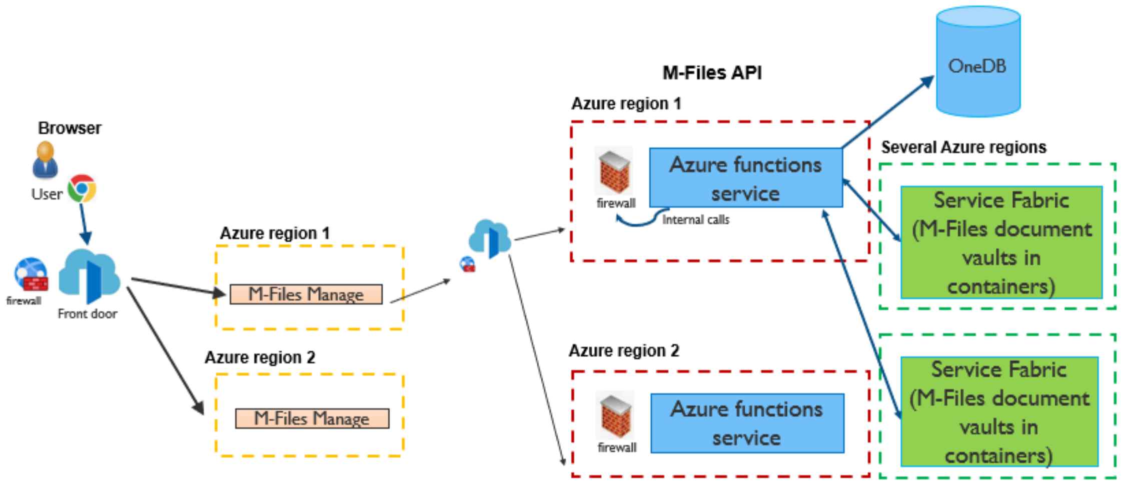


Figure 5: M-Files Cloud Management API

When users browse the M-Files Manage applications and do operations there, the web application communicates with OneDB through the M-Files Cloud Management API, which is implemented as an Azure Functions service.

Azure Functions is a serverless computing service provided by Microsoft Azure. It enables developers to focus on the application requirements and the logic by abstracting some of the hosting and maintenance related tasks involved. Among other things, Azure Functions can be used for building web APIs, processing file uploads, running scheduled tasks, and processing real time data. Azure Functions services are hosted by Microsoft and are a reliable high-performance solution for many types of applications. [11]

M-Files Manage and the Cloud Management API handle sensitive customer data and are utilized for crucial functionality of M-Files Cloud services. Because of this, the security of these systems is crucial, and it needs to be tested against common vulnerabilities.

4. WEB APPLICATION & API SECURITY

API vulnerabilities are a very real issue and may often go overlooked as the primary focus in development tends to be what is visible and accessible to the end users. For developers the difficulty of designing a large-scale API can be a challenge, which can lead to drifting towards simplicity in its design to make it more usable [13].

As much as developers could strive for perfection, vulnerabilities in applications are inevitable as implementation technologies age and attacker techniques develop and knowledge increases. Sometimes, however, vulnerabilities can be known beforehand and still make their way into a final product, as found in a survey done by Enterprise Strategy Group (ESG). For their report, they surveyed 378 IT, cybersecurity, and application development professionals and according to their findings, 48% of organizations are regularly pushing vulnerable code to their applications with an additional 31% admitting to doing it occasionally. [14] When the organizations pushing vulnerable code were questioned about why this is done, the findings present a clear image of how business goals sometimes interfere with security operations. 54% of respondents were pressured by critical deadlines, 49% considered the vulnerabilities low risk and not worth the effort to remediate, and 45% discovered the vulnerabilities too late to have time to do anything about them [14].

Web applications and APIs are attractive targets to malicious attackers as they're easier to reach than other potential targets. These types of web services are publicly available and do not require much prior target knowledge from the hacker. Cybersecurity and cloud services company Akamai has published a report of their observations on web attacks. In the report, they found the most commonly attempted web application attack vectors to be SQL injection, local file inclusion and cross site scripting, totaling over 10 billion detected attacks over a period of 18 months [15].

Just like other applications, APIs can face many different types of security vulnerabilities. API vulnerabilities may affect the confidentiality and integrity aspects of the system by, for example, enabling access to a database, allowing hackers to view, inject or modify protected data. Vulnerabilities could also be related to the availability of the API and other systems that rely upon it. For instance, a denial-of-service attack on the backend API of a web application could effectively make the web application unusable as well.

Research of real-life API implementations that investigated 60 randomly chosen publicly available APIs found that 53, or 88%, of those APIs contained security vulnerabilities of varying severity. The research was done on various types of web application APIs developed for different purposes, such as business, weather, and music APIs. Researchers found various issues including vulnerabilities such as insecure direct object reference, rate limiting, clickjacking, request forgery, scripting attacks and broken authentication. The most common issue found in the research paper was insecure direct object reference, found on 24% of API implementations that could provide access to confidential data. [13]

Commercial solutions for web application and API security are available, but their usefulness for a specific API implementation is something that needs to be studied. In a research article published in the Journal of International Technology and Information Management, cyber security customers and suppliers were surveyed about the current state and the future of API security. They found that 67% of respondents felt that current API security standards did not provide sufficient protection against potential threats. They also found that 65% of the surveyed customers were planning to use microservices and serverless computing in the future. [16] These results would suggest that security is a major concern to API developers, and they are not satisfied with the currently available security solutions. Also, as the use of microservices and serverless computing increases, security of third-party cloud services becomes increasingly important.

Web applications and API services moving more and more towards cloud-based solutions comes with its own challenges that arise from the virtualized, distributed, shared and public nature of cloud services. Cloud specific security issues may be related to data storage and computing, virtualization, service availability, network security, access control, software security or compliance related matters. Cloud services offer quick deployment, cost efficiency and availability but these come at a cost of unique security issues that previously may have gone unaccounted for. [17]

API Services have been a major target for malicious attackers in the past and continue to be a concern today. Even large successful organizations with popular services such as Facebook, YouTube, and GitLab have suffered from API security issues [18] [19] [20]. In 2018, a Facebook photo API issue granted third-parties unauthorized access to photos their users had uploaded into their service and affected up to 6.8 million users and 1500 applications that had utilized the API. Similarly, a YouTube API issue allowed a malicious user to upload videos to any user's channel, and a GitLab vulnerability allowed API users unauthorized access private directories.

Large organizations are often targeted by hackers and often have bug bounty programs in place to encourage the disclosure of found vulnerabilities to the developers. These types of programs have been proven to be cost-effective and have resulted in thousands of reported and fixed vulnerabilities across different organizations [21]. Results of such programs and the resulting fixed vulnerabilities are often shared to the users of those services, however, the number of existing vulnerabilities and those which are never reported on or disclosed to users can only be speculated upon.

Web application and API related security vulnerabilities can be protected against through following good practices during its development, proper security testing and an overall investment in security. The developers should know when they are doing something wrong and learn to recognize vulnerabilities before they are introduced into the application. Likewise, the testing process should involve inspecting the security aspects of the application.

5. SECURITY TESTING

The process of security testing is a validation of requirements directed towards a software system related to its security properties. These properties include aspects such as confidentiality, integrity, availability, authentication, authorization, and nonrepudiation. [22]

Confidentiality assures that sensitive information is not revealed to those who should not be able to access it. Integrity of a system is related to its data being protected against accidental or malicious modification or destruction. Availability aims to guarantee that the system is reliable and accessible to its users and will not be taken down by bad actors. Authentication validates the user of a system and provides a measure for controlling who has access to said system. Authorization manages the extent to which information or functionality an authenticated user has access to within the system. Nonrepudiation aims to ensure that a user of a system cannot deny having done something, and that a reliable audit trail exists for investigation purposes.

The security requirements for a system can be described either as a positive or negative requirements, i.e., how an application should behave or what it should not do [22]. Using the security requirement of availability as an example, a positive requirement could be described as: “The system should limit users to 100 requests per minute”, whereas a similar negative requirement could be: “The system should not be able to be taken down by a malicious attacker attempting a denial-of-service attack.”

The end goal of security testing is to recognize and prevent vulnerabilities in the software. A vulnerability is a software fault related to its security properties. An attack against a software system is an exploitation of one or more of these vulnerabilities, resulting in the compromise of one or more security properties of the system. This malicious abuse of open vulnerabilities in a system is referred to as an exploit. [22]

Security testing aims to verify that a software system meets its security requirements by simulating attacks that attempt to compromise its security. A security tester plays the role of a malicious hacker trying to attack a system in order to exploit its vulnerabilities. [22] The vast majority of software security incidents are caused by attackers exploiting known vulnerabilities, which may originate from either software faults or design flaws of the system [23]. The process of security testing can differ depending on the whether or not the goal is to verify a positive or negative requirement of the system. Positive requirements can be tested against with more classical testing techniques, such as unit tests, while negative security requirements may often require more advanced techniques and research, including risk analysis, penetration testing, and the utilization of vulnerability databases [22].

Security testing of software is often started relatively late in the software development life cycle, which has been considered to be an ineffective and inefficient practice. To prevent security issues from making their way into production environments, security aspects need to be considered throughout the development process of an application. [22] This can be done, for instance, through initiating a security threat modeling and risk analysis process from the very beginning of the design phase and specifying the security-related requirements of the system early, so that they can be taken into account during its implementation. One major security development process model is Microsoft SDL, which aims to pursue the principles of secure by design, security by default, secure in deployment, and open communication of security threats [24].

5.1 Approaches to Security Testing

5.1.1 White and Black Box Testing

The actual process of security testing can proceed in different ways depending on the system under test and the person executing the testing. When it comes to security testing, common concepts often referred to include white and black box testing. The primary difference between these two is whether or not the tester has access to the underlying source code of the application.

In white box testing, the tester has full access to the source code and no limitations to how much they know about its implementation. This enables the person doing the testing to develop more advanced test cases and also better target the nature of those test cases to apply to specific weaknesses that the system might be vulnerable to. White box testing can often find more complex issues within the system, such as code optimization issues, insecure coding practices, and also apply unit testing practices.

Black box testing is executed by a tester that has no access to the source code and often knows relatively little about the actual implementation of the application being tested. As the tester lacks knowledge of the system’s internal workings, the testing process can be simpler and relies on sending inputs to the system and analyzing its outputs. When it comes to security testing, black box testing may not always be as extensive as white box testing, where the test cases are more targeted and results can be analyzed in depth, however, when considering malicious attackers, black box testing suits well as a simulation of what information an outside actor would have access to.

5.1.2 Manual testing

There are many solutions available that offer automated security testing of applications; however, sometimes manual testing is required as an automated solution might not be viable for one reason or another. Manual security testing relies on the person's knowledge of different software vulnerabilities and possible exploits that they then apply to try and break the targeted system while acting as an attacker. This type of security testing can vary between white and black box testing and may be done by an outside security tester that has limited knowledge of the actual implementation, or internally by someone that has full access to analyze the system and its source code.

Both manual and automated security testing can check for the same types of vulnerabilities, but a security tester may be more effective, especially if the targeted system is not a standardized solution, where an automated security testing tool can achieve good coverage and fully understand the requests and responses of the targeted system.

A person doing manual security testing can take advantage of various tools in the testing process to launch simulated attacks, collect data, and assess vulnerabilities, and as such, the process may be partially automated, but the actual results are reviewed by someone with expertise and the ability to better assess whether or not the results present a genuine vulnerability within the system.

In comparison to a fully automated process, manual security testing can avoid large numbers of false positives. Negatives of manual security testing are that it can be a time-consuming process requiring someone with a lot of expertise. It can also be more difficult to apply active security testing to a system, where regressions would be detected in a timely manner, in comparison to a fully automated solution integrated into the CI/CD build pipelines of an application.

5.1.3 Automated testing

Automated security testing tools can test a target system in various different ways depending on the tool's purpose and its goal. Dynamic application security testing (DAST) tools test an application by utilizing a black box approach where requests are generated, and the responses are then analysed to detect places where the application could be vulnerable. In case of web applications, DAST tools typically crawl the web application to locate resource URLs and API endpoints, which are then targeted with various inputs attempting to identify known vulnerabilities within the system.

Static application security testing (SAST) tools are an opposite approach to automated security testing. With SAST tools, the source code of the application is reviewed to identify places where the application could be vulnerable. SAST tools can be applied during an application's development to prevent bad code from being written, or it can be done later to check for vulnerable code in a complete application. In contrast, DAST tools can typically only be applied when an application has a working version and is nearing the end of its development.

When comparing DAST and SAST tools, SAST tools can typically achieve a better level of code coverage, as the scanner has access to the codebase and can detect even hidden inputs to the application. Meanwhile, DAST tools can be more resistant to false positives. Both approaches have their advantages, which is why combinations of these two have been developed to a grey box DAST tools, referred to as interactive application security testing (IAST) tools. IAST tools combine the benefits of both a black box DAST tool and a white box SAST tool. Implementations of IAST tools from different vendors vary. Some of them work by utilizing sensors, installed within the application, that relay information back to the scanner during its execution. [25]

Automated security testing tools can be a cost-effective solution when regular security testing is required, however, finding the right tool that fits the application can be a time-consuming and difficult process. Many of the automated security testing tools apply very well to generic applications, but when the scale and complexity of a system grows, automated security scanning tools can quickly become ineffective.

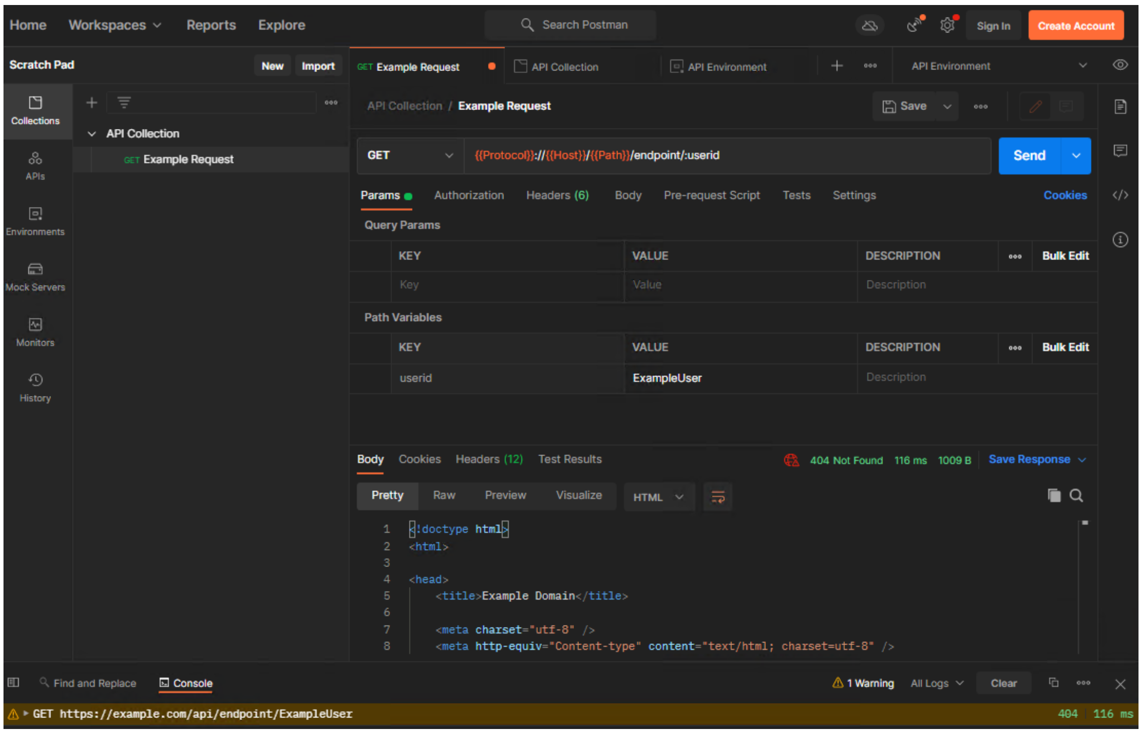
In addition to using dedicated tools for automated security testing, unit testing frameworks can be utilized to create relatively simple test cases that can often catch erroneous code from being written in the first place.

5.2 Tools in API Security Testing

There are many tools that can be utilized for the purpose of security testing API services. Some of these applications are freely available open-source projects and some of them are commercial solutions offered by security-oriented organizations. This chapter will describe some of the tools used during the research and testing done during the master's thesis.

5.2.1 Postman

Postman is an API platform designed to simplify the development and testing process of API implementations. Postman's key features include the creation of collections, environments, variables and automatable test scripts. Collections within Postman allow the consolidation of API requests, parameters, descriptions and tests to a single exportable format, which can then be stored in version control and simplify the development process within a team. Variables allow the user to store and reuse values within their scripts and requests. Environments are collections of these variables that are used to group and save an environment-specific set of values for testing purposes.



GET {{Protocol}}://{{Host}}/{{Path}}/endpoint/:userid

Query Params

KEY	VALUE	DESCRIPTION
Key	Value	Description

Path Variables

KEY	VALUE	DESCRIPTION
userid	ExampleUser	Description

404 Not Found 116 ms 1009 B

```
<!doctype html>
<html>

<head>
    <title>Example Domain</title>

    <meta charset="utf-8" />
    <meta http-equiv="Content-type" content="text/html; charset=utf-8" />
```

```
▸ GET https://example.com/api/endpoint/ExampleUser                404   116 ms
```

Figure 6: Postman Application

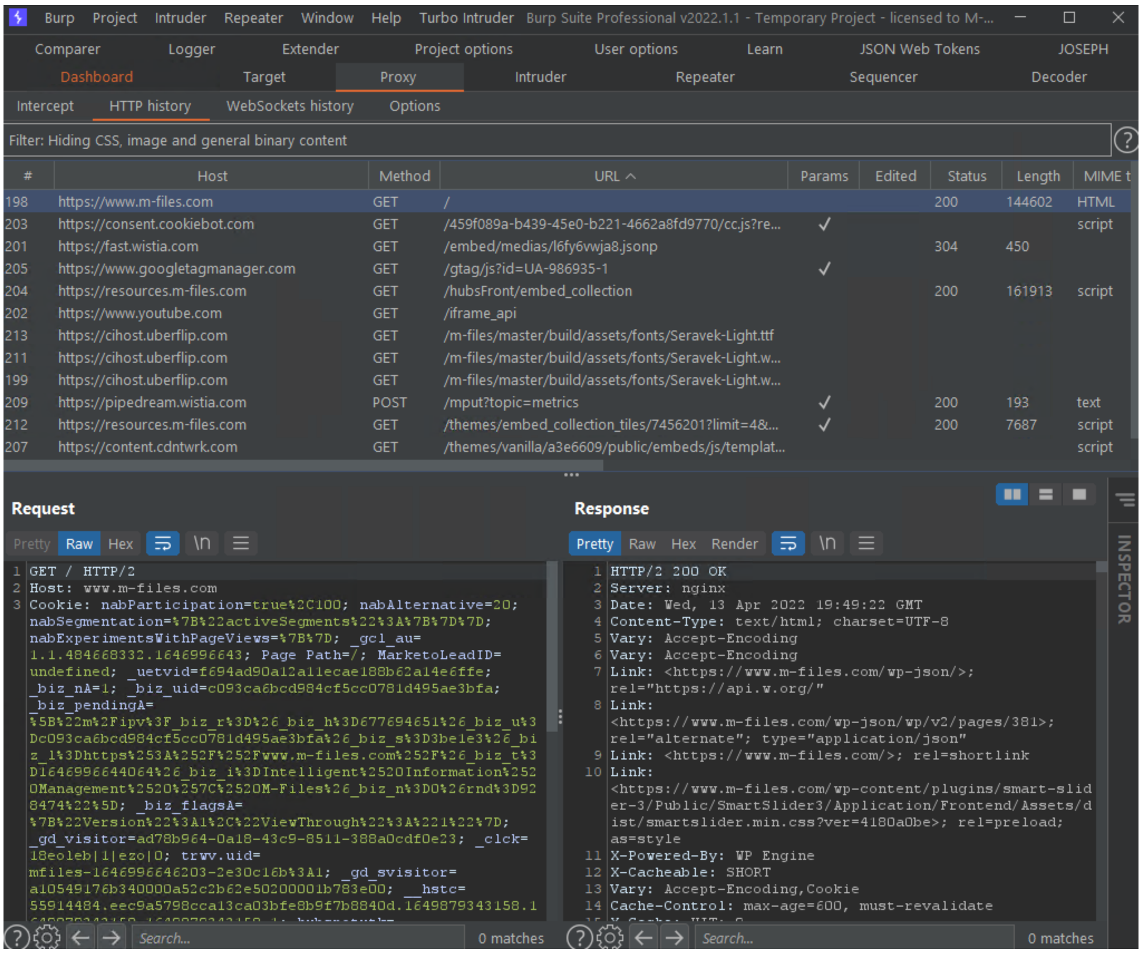
Postman can be used to save collections of preconfigured API requests with customized parameters, and because of this it is a convenient tool for manual testing of API endpoints. With Postman, requests can easily be modified to cover various test cases and the responses can be examined to find potential issues with an API implementation. Documentation can also be integrated into API collections and thus the behaviour of different endpoints can be reviewed to function as intended.

In this thesis, Postman was used extensively in combination with other tools. It proved to be a very useful tool for security testing purposes, as it can be configured to use a HTTPS proxy, allowing tools like Burp Suite to be utilized, which can then be used to automatically review the API responses to discover points where the API implementation could potentially be vulnerable.

5.2.2 Burp Suite

Burp Suite is a security and penetration tool developed by PortSwigger that enables in-depth security testing of web applications [26]. It is one of the most widely used web application security testing tools as it contains many features and good extensibility for different use cases.

Features of Burp Suite include interception and analysis of HTTP(S) requests, exposing attack surfaces, assessing target applications and security features, facilitating manual security testing, automated security scans and application fuzzing. Burp Suite can be utilized for automated scanning of web applications and automatic assessment of results, linking to relevant recognized web application vulnerabilities, and generating reports.



Filter: Hiding CSS, image and general binary content

#	Host	Method	URL	Params	Edited	Status	Length	MIME t
198	https://www.m-files.com	GET	/			200	144602	HTML
203	https://consent.cookiebot.com	GET	/459f089a-b439-45e0-b221-4662a8fd9770/cc.js?re...	✓				script
201	https://fast.wistia.com	GET	/embed/medias/l6fy6vwja8.jsonp			304	450	
205	https://www.googletagmanager.com	GET	/gtag/js?id=UA-986935-1	✓				
204	https://resources.m-files.com	GET	/hubsFront/embed_collection			200	161913	script
202	https://www.youtube.com	GET	/iframe_api					
213	https://cihost.uberflip.com	GET	/m-files/master/build/assets/fonts/Seravek-Light.ttf					
211	https://cihost.uberflip.com	GET	/m-files/master/build/assets/fonts/Seravek-Light.w...					
199	https://cihost.uberflip.com	GET	/m-files/master/build/assets/fonts/Seravek-Light.w...					
209	https://pipedream.wistia.com	POST	/mput?topic=metrics	✓		200	193	text
212	https://resources.m-files.com	GET	/themes/embed_collection_tiles/7456201?limit=4&...	✓		200	7687	script
207	https://content.cdntwrk.com	GET	/themes/vanilla/a3e6609/public/embeds/js/templat...					script

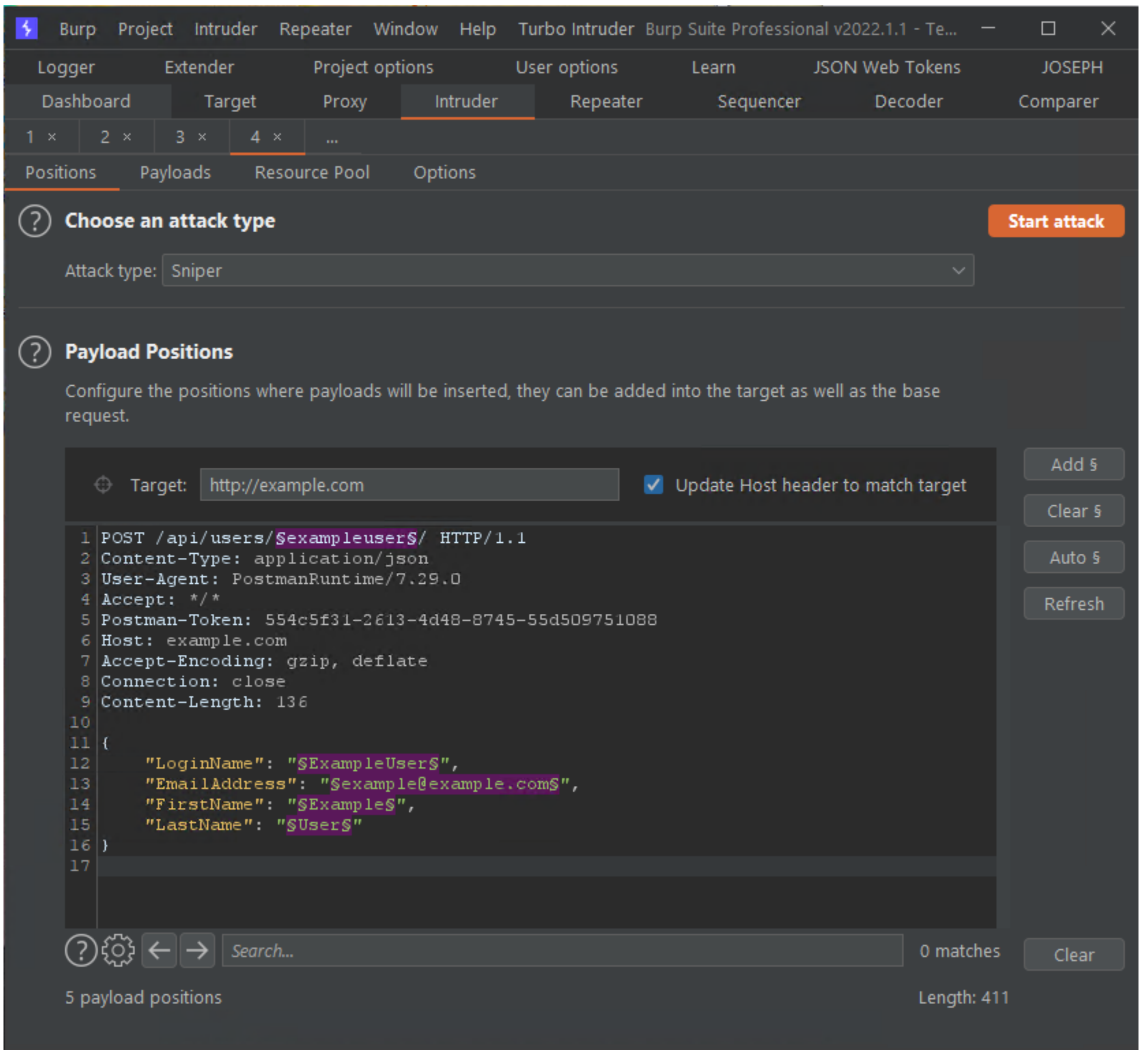
```
GET / HTTP/2
Host: www.m-files.com
Cookie: nabParticipation=true%2C100; nabAlternative=20;
nabSegmentation=%7B%22activeSegments%22%3A%7B%7D%7D;
nabExperimentsWithPageViews=%7B%7D; _gcl_au=
1.1.484668332.1646996643; Page Path=/; MarketoLeadID=
undefined; _uetvid=f694ad90a12a11ecae188b62a14e6ffe;
_biz_nA=1; _biz_uid=c093ca6bcd984cf5cc0781d495ae3bfa;
_biz_pendingA=
%5B%22m%2Fipv%3F_biz_r%3D%26_biz_h%3D677694651%26_biz_u%3
Dc093ca6bcd984cf5cc0781d495ae3bfa%26_biz_s%3D3be1e3%26_bi
z_l%3Dhttps%253A%252F%252Fwww.m-files.com%252F%26_biz_t%3
D1646996644064%26_biz_i%3DIntelligent%2520Information%252
0Management%2520%257C%2520M-Files%26_biz_n%3D0%26rnd%3D92
8474%22%5D; _biz_flagsA=
%7B%22Version%22%3A1%2C%22ViewThrough%22%3A%221%22%7D;
_gd_visitor=ad78b964-0a18-43c9-8511-388a0cdf0e23; _clck=
18eoleb|1|ezo|0; trwv.uid=
mfiles-1646996646203-2e30c16b%3A1; _gd_svisitor=
a10549176b340000a52c2b62e50200001b783e00; __hstc=
55914484.eec9a5798cca13ca03bfe8b9f7b8840d.1649879343158.1
```

```
HTTP/2 200 OK
Server: nginx
Date: Wed, 13 Apr 2022 19:49:22 GMT
Content-Type: text/html; charset=UTF-8
Vary: Accept-Encoding
Vary: Accept-Encoding
Link: <https://www.m-files.com/wp-json/>;
rel="https://api.w.org/"
Link:
<https://www.m-files.com/wp-json/wp/v2/pages/381>;
rel="alternate"; type="application/json"
Link: <https://www.m-files.com/>; rel=shortlink
Link:
<https://www.m-files.com/wp-content/plugins/smart-slid
er-3/Public/SmartSlider3/Application/Frontend/Assets/d
ist/smartslider.min.css?ver=4180a0be>; rel=preload;
as=style
X-Powered-By: WP Engine
X-Cacheable: SHORT
Vary: Accept-Encoding,Cookie
Cache-Control: max-age=600, must-revalidate
```

Figure 7: Burp Suite as a HTTPS proxy

One of the most useful features of Burp Suite is the ability to proxy and capture HTTP traffic from other applications to inspect and edit those requests. Most programs that utilize HTTP can be configured to use a proxy and the traffic can easily be proxied through Burp Suite. This can also be done on the operating system level if all traffic on the computer should be proxied.

During this thesis, Burp Suite was used together in this way with Postman and JSON Web Token Toolkit. Proxying requests through Burp allows for in depth investigation of requests and responses. Burp can automatically warn of suspicious transactions and is able to, for instance, suggest places where an injection vulnerability might be possible.


Burp Project Intruder Repeater Window Help Turbo Intruder Burp Suite Professional v2022.1.1 - Te...
Logger Extender Project options User options Learn JSON Web Tokens JOSEPH
Dashboard Target Proxy Intruder Repeater Sequencer Decoder Comparer
1 × 2 × 3 × 4 × ...
Positions Payloads Resource Pool Options
Start attack
Add §
Clear §
Auto §
Refresh
Search...
0 matches
Clear

Choose an attack type

Attack type: Sniper

Payload Positions

Configure the positions where payloads will be inserted, they can be added into the target as well as the base request.

Target: http://example.com ☑ Update Host header to match target

```
POST /api/users/§exampleuser§/ HTTP/1.1
Content-Type: application/json
User-Agent: PostmanRuntime/7.29.0
Accept: */*
Postman-Token: 554c5f31-2613-4d48-8745-55d509751088
Host: example.com
Accept-Encoding: gzip, deflate
Connection: close
Content-Length: 136

{
    "LoginName": "§ExampleUser§",
    "EmailAddress": "§example@example.com§",
    "FirstName": "§Example§",
    "LastName": "§User§"
}
```

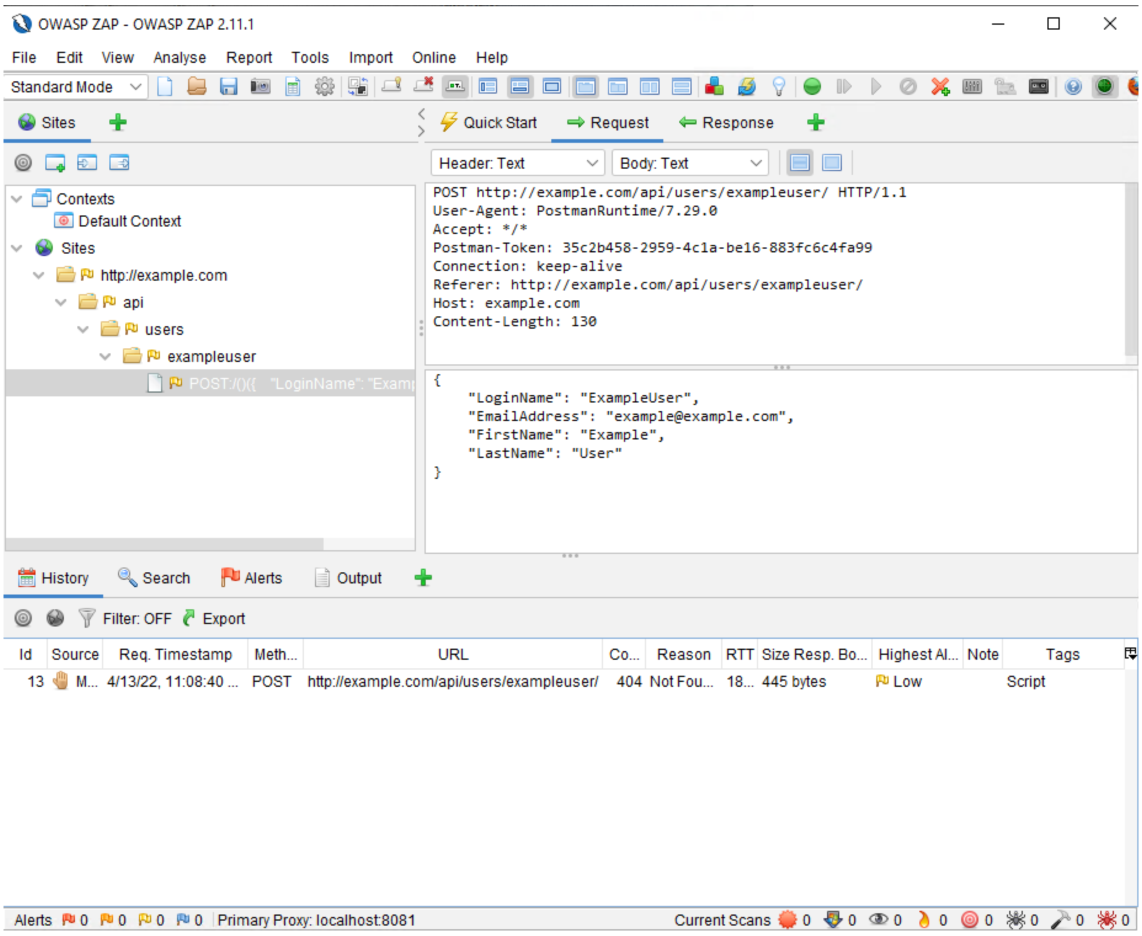
5 payload positions Length: 411

Figure 8: Burp Intruder

Captured requests can be easily modified and repeated with Burp Suite. During this thesis, the Intruder tool proved particularly useful, as it can be utilized to configure positions within a captured HTTP request that can then be injected with either common fuzzing payloads provided by Burp or with user configured values. Fuzzing API endpoints and looking for e.g., injection vulnerabilities is relatively trivial by utilizing Burp Suite.

5.2.3 OWASP ZAP

OWASP Zed Attack Proxy (ZAP) [27] is a free open-source web application security scanner that offers much of the same functionality as Burp Suite. It can be utilized for proxying and capturing HTTP requests, as well as scanning for known web application vulnerabilities.


OWASP ZAP - OWASP ZAP 2.11.1
File Edit View Analyse Report Tools Import Online Help
Standard Mode
Sites
Quick Start
Request
Response
Header: Text
Body: Text
Contexts
Default Context
Sites
http://example.com
api
users
exampleuser
POST:/()({ "LoginName": "Examp
History Search Alerts Output
Filter: OFF Export
Alerts 0 0 0 0 Primary Proxy: localhost:8081
Current Scans 0 0 0 0 0 0 0 0

```
POST http://example.com/api/users/exampleuser/ HTTP/1.1
User-Agent: PostmanRuntime/7.29.0
Accept: */*
Postman-Token: 35c2b458-2959-4c1a-be16-883fc6c4fa99
Connection: keep-alive
Referer: http://example.com/api/users/exampleuser/
Host: example.com
Content-Length: 130
```

```
{
    "LoginName": "ExampleUser",
    "EmailAddress": "example@example.com",
    "FirstName": "Example",
    "LastName": "User"
}
```

Id	Source	Req. Timestamp	Meth...	URL	Co...	Reason	RTT	Size Resp. Bo...	Highest Al...	Note	Tags
13	M...	4/13/22, 11:08:40 ...	POST	http://example.com/api/users/exampleuser/	404	Not Fou...	18...	445 bytes	Low		Script

Figure 9: OWASP ZAP

ZAP can be used as an HTTPS proxy just like Burp Suite and offers similar functionality in terms of fuzzing API endpoints. The main difference between OWASP ZAP and Burp Suite is the fact that Burp is a commercial solution while ZAP is completely open source and available to anyone. Their functionality is very similar and for most scenarios either tool will do the job.

During this thesis, both OWASP ZAP and Burp Suite were utilized in investigating the API, however, Burp Suite was used to a much bigger extent. The main reason for this came down to personal preference, however, Burp Suite seemed to be more widely used and solutions to commonly encountered issues were easier to find.

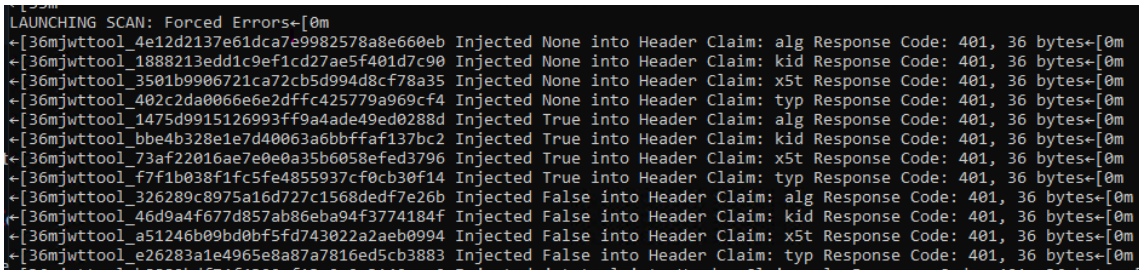
In terms of automatic alerts of potential vulnerabilities, ZAP and Burp can both detect similar issues, but they do have their own alerts and automatic scans of web services often have differing results. When doing automatic scanning of an API or a web application, utilizing both tools can be beneficial.

5.2.4 The JSON Web Token Toolkit

The JSON Web Token Toolkit is an open-source penetration testing tool that can be used to test and verify protection for multiple known JWT-based security vulnerabilities [28].

These vulnerabilities include JWT signature algorithm-based attacks, such as the none-algorithm vulnerability [29] and the RSA/HMAC algorithm confusion vulnerability [30]. These vulnerabilities allow a malicious user to edit claims within their token to gain access to resources they shouldn't be able to access. By being able to bypass the signature checks or by generating their own, seemingly valid, signatures, the attackers can gain access to any resources protected by JWT-based authentication. [31]

The toolkit is a console application written in python. It can be utilized to test REST APIs that use JSON Web Tokens for user authentication for common security misconfiguration issues. The toolkit contains multiple features that help with testing JSON Web Tokens, such as automated scans, token fuzzing, dictionary attacks on signing keys, creating your own tokens, editing token content, and more.



```
LAUNCHING SCAN: Forced Errors←[0m
←[36mjwttool_4e12d2137e61dca7e9982578a8e660eb Injected None into Header Claim: alg Response Code: 401, 36 bytes←[0m
←[36mjwttool_1888213edd1c9ef1cd27ae5f401d7c90 Injected None into Header Claim: kid Response Code: 401, 36 bytes←[0m
←[36mjwttool_3501b9906721ca72cb5d994d8cf78a35 Injected None into Header Claim: x5t Response Code: 401, 36 bytes←[0m
←[36mjwttool_402c2da0066e6e2dffc425779a969cf4 Injected None into Header Claim: typ Response Code: 401, 36 bytes←[0m
←[36mjwttool_1475d9915126993ff9a4ade49ed0288d Injected True into Header Claim: alg Response Code: 401, 36 bytes←[0m
←[36mjwttool_bbe4b328e1e7d40063a6bbffaf137bc2 Injected True into Header Claim: kid Response Code: 401, 36 bytes←[0m
←[36mjwttool_73af22016ae7e0e0a35b6058efed3796 Injected True into Header Claim: x5t Response Code: 401, 36 bytes←[0m
←[36mjwttool_f7f1b038f1fc5fe4855937cf0cb30f14 Injected True into Header Claim: typ Response Code: 401, 36 bytes←[0m
←[36mjwttool_326289c8975a16d727c1568dedf7e26b Injected False into Header Claim: alg Response Code: 401, 36 bytes←[0m
←[36mjwttool_46d9a4f677d857ab86eba94f3774184f Injected False into Header Claim: kid Response Code: 401, 36 bytes←[0m
←[36mjwttool_a51246b09bd0bf5fd743022a2aeb0994 Injected False into Header Claim: x5t Response Code: 401, 36 bytes←[0m
←[36mjwttool_e26283a1e4965e8a87a7816ed5cb3883 Injected False into Header Claim: typ Response Code: 401, 36 bytes←[0m
```

Figure 10: Automated JWT Toolkit Scan

The JSON Web Token Toolkit works well when combined with Burp Suite, as all requests from the tool may be routed through its proxy service so both requests and responses can be inspected in greater detail. Overall, the tool is relatively easy to configure and use, and it provides an efficient method for checking against well-known JWT authentication vulnerabilities.

5.2.5 Security Code Scan

Security Code Scan (SCS) is an open-source static code analyzer for .NET based software solutions that provides detection for common security vulnerability patterns. It's developed by GitHub Security Lab and offered as a Microsoft Visual Studio extension, NuGet package as well as a stand-alone runner that can be used as a console application. The tool can also be integrated into the CI pipelines within version control solutions such as GitHub and GitLab. [32]

When used as a stand-alone application, Security Code Scan loads the software solution and parses through the source code looking for potential vulnerabilities, such as SQL injection, cross-site scripting, path traversal, weak cryptographic protocols, and use of hardcoded secret values. If the tool finds potential issues within the code, it will point these out to the developer and let them know which vulnerability was found and where it resides. If integrated to a development environment such as Visual Studio, the tool may be configured to run in the background and point out potential security issues as new code is being written.

As with any code analysis tool, there may be false positives and minor issues reported within the results, which do not require fixing. The tool offers a way for developers to suppress and ignore issues as well as set custom severity for each rule it's configured to scan against.

During this thesis, Security Code Scan was used to automatically analyze the source code of the API implementation and the results of these scans were reviewed manually. Despite the lack of actual findings and having to deal with some false positives, SCS proved to be a useful tool as it is very lightweight and was able to scan large projects quickly.

6. API THREATS AND VULNERABILITIES

This chapter takes a look at API related security vulnerabilities, describing the vulnerabilities, providing examples as well as methods for protecting and testing for these vulnerabilities. To improve upon readability of the thesis, as well as for the sake of conciseness, this chapter also describes the process of how M-Files API was reviewed and tested for these vulnerabilities.

6.1 Broken Authentication

API implementations with broken authentication can cause a major impact to data security. A broken authentication vulnerability can involve, among other things, unprotected API endpoints, weak authentication methods, lacking access token validation, the use of weak non-expiring tokens, or vulnerability to brute force attacks.

Implementing proper authentication can be a difficult part of API design but it is one of the most important factors when it comes to securing the system. The authentication mechanisms are exposed to all users of the application and as such they should follow the best security practices and be heavily tested to ensure their security. When it comes to API implementations, there are two major authentication-related issues that can arise: the lack of protection mechanisms and the misimplementation of those mechanisms [33].

Two commonly used authentication types within API implementations are API keys and authentication tokens such as JWTs. Depending on the type of API these can be used in different ways. An API implementation may utilize one or the other or a combination of these authentication methods.

In token-based authentication, a user authenticates to the API and is issued a token that is stored in the user's browser and sent along with the HTTP requests to the API. Different types of implementations exist for token-based authentication and the exact location where the tokens are stored by the user's browser also varies. Common types of tokens used with APIs include OAuth access tokens and JSON Web Tokens. One major difference between these two is the fact that OAuth tokens are randomly generated strings that provide no other value to the application than authentication, while JWTs contain claims with various types of information about the user, that can be accessed by applications. Access tokens are typically stored in one of two places by the browser: the local storage, or sessions cookies. There is not a clear answer to which is better as they both have their advantages and disadvantages, for instance, cookies are typically more vulnerable to CSRF attacks, while the local storage is more accessible to XSS attacks.

API keys are typically used to identify a specific service or a business utilizing an API rather than an end user and they usually have a much longer expiration time [34]. With API keys, a developer of the application utilizing the API registers an API key that is sent with every request to authenticate and authorize the application as a client of that API service.

A weak authentication method can pose a security risk to any application. Issues such as exposing authentication details in URLs as well as utilizing weak encryption and hashing algorithms are typical issues when it comes to authentication methods. Choosing the right type of authentication to use and following the best practices described for that particular authentication type are key in maintaining a secure system.

A secure API implementation should protect against brute force attacks such as credential stuffing in order to limit the possibility of attackers gaining access to the system. This can be done, for instance, by implementing lockout mechanisms within the authentication process. With lockouts in place that block a user following unsuccessful authentication attempts, an attacker would not be able to efficiently run automated tools to perform brute force attacks or guess commonly used login credentials.

Proper validation of access tokens within an API is a key factor when it comes to authentication. A token-based authentication mechanism should always make sure that the token is valid before giving access to the requested resource. With JWTs this includes all claims written to the token as well as the signature. It should also be verified that the token has not expired, as a potential attacker could possibly get access to an expired token and use that to authenticate themselves. Different forms of authentication can have their own vulnerabilities that can cause harm if not protected against. For instance, several JWT libraries have known exploits that can allow access to an unauthorized attacker [31].

Good common practices to follow in implementing API authentication include proper understanding of the authentication mechanisms, utilizing common authentication standards, protecting all authentication-related endpoints, implementing multi-factor authentication where possible, and not utilizing API keys for user authentication [33].

6.1.1 M-Files API

The M-Files API utilizes a combination of API key and token-based authentication in its implementation. The services that utilize the API are registered to the Azure Functions service and keys are generated that allow them access to various API methods. For all API methods, checks are in place to make sure that only the specified services are capable of calling the execution of those functions. In addition to API key checks, all users of M-Files Manage are authenticated with a separate authentication service application that generates JSON Web Tokens that are used for authentication and authorization purposes. Users of services that rely upon the API have access to different methods and data.

The authentication provider of the API services has been extensively tested previously and was not a major part of this thesis, however, testing was done on the API's validation of API keys and access tokens. All API endpoints were also verified to require authentication when called.

The authentication of the M-Files API was tested with the JSON Web Token Toolkit, described in more detail earlier in chapter 5.4. The toolkit provides an easy way to check for common vulnerabilities in JWT libraries that could allow an attacker to bypass authentication and access sensitive data as a result. The tool was used in combination with Burp Suite to proxy and analyze the traffic.

In addition to testing for JWT-related issues, automated tests were developed for the M-Files API build pipeline to catch any issues that could arise in the future. The goal was to catch a situation in which these issues are introduced as a result of changes done to the API during its continuing development. For instance, if a new library is taken into use that contains one of the previously known vulnerabilities, the tests will fail resulting in the vulnerability never making its way into the actual deployment versions of the API service. Tests were also created to verify the proper use of API keys generated for different services. As with other tests, these tests will ensure changes to API key usage do not change the intended functionality without being caught.

No authentication-related issues were found within the API during testing. Largely thanks to previous testing work done on the authentication service and best practices followed in its development.

6.2 Broken Authorization

Effective and well-implemented user authorization is a key component in web applications and API services. API implementations should have consistent well-defined requirements for authorization to ensure proper access control to different resources. A malicious user should not be able to access any resources for which they have not been explicitly granted access to. This includes the resources of other API users as well as any other functionality, like admin panels, that may be built into the API implementation.

Broken authorization is a major concern in API implementations and is well recognized within the OWASP API Security Top 10 to be one of the most common security issues. OWASP has separated broken authorization into two separate issues within their list: broken object level authorization and broken function level authorization. [33]

Object level authorization is the mechanism through which users should only have access to objects they have been granted access to. If a user tries to access resources belonging to someone else that they should not be able to access by, for instance, changing a URL parameter, they should receive an error and not be able to access these resources, or obtain any information about them.

If an API is vulnerable because of broken object level authorization, an authenticated user may be able to access other user's resources through changing a parameter within the request URL. For instance, if an API provides access to the user's personal information with an API call such as "GET /user/**userid1**/info", changing the id parameter to another user's id and making a new request such as "GET /user/**userid2**/info" should result in an error, and not allow access to the user.

Function level authorization refers specifically to protecting sensitive functions from unauthorized users of the API. While object level authorization dealt with protecting API resource objects, the goal here is to protect any other functionality the API may have. API implementations can have much more complex functionality than just fetching resources from a database, and the access to these functions should be well protected with access only being granted to those who are properly authorized. API implementations may, for instance, have endpoints with functions meant for administrative purposes that only a certain userbase should have access to.

One example of broken function level authorization is a case where users can access unintended functionality by simply changing the type of HTTP request. Users may have authorized access to fetching API resources with an HTTP GET method, but if the authorization mechanisms of the API are not well implemented, they could simply change the type of request to, for instance, PATCH or DELETE to edit or delete those resources.

To prevent these vulnerabilities, proper authorization should be implemented for all API endpoints and unintended usage of those endpoints, such as unsupported HTTP methods should be kept in mind. For sensitive resources, access should be denied by default and explicitly granted to users. Authorization should be carefully designed and tested to work as intended before an application is taken into wide use. API implementations should also not contain any “hidden functionality” that does not implement proper authorization. Security through obscurity is bad design, as fuzzing and API crawling are common and easy to perform tactics used by attackers.

6.2.1 M-Files API

Goal of testing the M-Files API for broken authorization was to ensure proper authorization checks were implemented for all API endpoints and that the authorization is implemented correctly.

To accomplish these goals, the source code of the API service was reviewed, manual testing was done on the API, and test automation was ensured to provide adequate coverage to detect any regression-related issues that could threaten the API’s security. The test automation coverage for authentication and authorization related issues was found to be quite extensive, and as such, few new test cases were created during this thesis.

The source code of each API endpoint was manually reviewed to contain implementation for authorization. This accomplished the goal of ensuring that authorization is present. After this, the authorization had to be tested to function as intended.

The authentication provider of M-Files API was tested to grant the correct access rights to different types of users. Automated unit test coverage was ensured to detect a regression situation where users would be granted unintended privileges.

API endpoints were verified to properly check user authorization and allow access only to authenticated users that should be allowed to utilize those resources and API endpoints. Manual checking of different API endpoints was done by accessing those resources with different types of authenticated users. Existing test automation coverage was reviewed and some new test cases for unauthorized access were added.

Some examples of authorization related test cases that test automation can cover include situations such as: no access, lacking access, invalid access tokens, accessing other customer’s resources, admin functionality, and unintended HTTP methods.

No authorization related issues were found within the API during this thesis and access control was found to be working as intended. Some value came out of testing for this issue however, as test automation coverage for this issue was reviewed and improved upon.

6.3 Excessive Data Exposure

Excessive data exposure is an API vulnerability in which the backend API behind a client application returns unnecessary and revealing information that is not required or requested by the client application.

API implementations should never rely on the client implementation to filter the data returned by the backend API as any user can read the raw HTTP responses returned to the web application. The API should only return as much information as is required for the application to function correctly.

Excessive data exposure is a common and well-recognized API vulnerability and is listed in the OWASP API Security Top 10. There are also several Common Weakness Enumeration (CWE) entries related to this vulnerability, such as *CWE-200: Exposure of Sensitive Information to an Unauthorized Actor* [35], *CWE-213: Exposure of Sensitive Information Due to Incompatible Policies* [36], and *CWE-209: Generation of Error Message Containing Sensitive Information* [37].

Excessive data exposure vulnerabilities may be introduced through things such as inefficient code review practices, the simple implementation of methods that return all data without filtering it, or the reuse of previously safe API methods in places where more strict data access policies should be in place.

Excessive data exposure can be prevented by reviewing the API responses and taking into account the consumer of the data and what is the level of access they should have to the requested resources. Ideally this review should be done during the development of the application in order to not introduce technical debt into the application. When new types of clients begin utilizing an API method, a new review into the level of returned data should take place and it should be placed under question as to whether the client actually requires the level of data returned by the API.

Excessive data exposure also applies to exceptions and error messages and should be considered there as well. The API implementation should not return any unnecessary and revealing data to the client application. The error responses may contain information about the requested resource or reveal implementation related details of the API application, such as the technologies used in the backend, database structure, or other sensitive information. When it comes to API related error messages returned to an application, the messages should be reviewed just as any other response and the type of client, and their level of access should be considered.

As an example of excessive data exposure, let’s assume we have an API endpoint which is called by a web application when a user looks at a list of the vaults, they have access to. The API is called as follows:

```
GET /api/{userid}/vaults/
```

with the *userid* being replaced with the username of the authenticated user. Now, the user with the username “jane” navigates to the page where all their accessible vaults are listed, and the API is called:

```
GET /api/jane/vaults/
```

While the user browses to the page, the web application calls the API, and it responds with the following JSON data in its HTTP response:

```
{
    "vaults": [
        {
            "vaultName": "Customer Support",
            "vaultAddress": "cs.example.com",
            "description": "Customer Support Database",
            "users": [
                {
                    "userName": "jane",
                    "fullName": "Jane Doe",
                    "email": "jane.doe@example.com"
                },
                {
                    "userName": "john",
                    "fullName": "John Smith",
                    "email": "jsmith@example.com"
                }
            ]
         },
         {
            "vaultName": "Human Resources",
            "vaultAddress": "hr.example.com",
            "description": "HR Database",
            "users": [
                {
                    "userName": "frank",
                    "fullName": "Frank Davis",
                    "email": "frankdavis@example.com"
                },
                {
                    "userName": "john",
                    "fullName": "John Smith",
                    "email": "jsmith@example.com"
                },
                {
                    "userName": "jane",
                    "fullName": "Jane Doe",
                    "email": "jane.doe@example.com"
                }
            ]
         },
         {
            "vaultName": "Management",
            "vaultAddress": "management.example.com",
            "description": "Management Database",
            "users": [
                {
                    "userName": "john",
                    "fullName": "John Smith",
                    "email": "john@example.com"
                }
            ]
        }
    ]
}
```

After the API call is successful, the web application client displays to the user all the vaults where they have a user account, however, looking at the raw result of the API call, it is clear that it contains much more information than is required for the purpose of listing the vaults to which the user has access to. Not only does the API response contain vaults to which the user does not have access, it also contains all the users of each vault. None of this information is displayed within the web application, as it is filtered out by the client application calling the API.

The web application and API implementation described in the above example relied upon client-side filtering of information and as a result much more information was available to the user than originally intended. Excessive data exposure should always be considered during web application and API implementation, as client-side filtering of information is not secure. The results of the web application's underlying API calls are not secret, and a knowledgeable user is always able to see the results of the requests made while browsing the web application.

6.3.1 M-Files API

The M-Files API implementation was tested and reviewed for the excessive data exposure vulnerability as a part of this thesis. Testing for this vulnerability consisted of enumerating through all the defined API endpoints and reviewing the level of information they returned in both successful and unsuccessful responses. For each API method the end user and purpose of the API call was considered so as to ensure good practices were being utilized in sanitizing the amount of data returned to the client.

Burp was used as an HTTP proxy to capture requests and their responses. The requests to the API were generated both by using the Manage web application as well as directly called with Postman. Both successful and unsuccessful requests were used. In addition to manual exploratory penetration testing, automated audit scans were also run with Burp to modify the requests further and trigger errors that might have been missed with manual testing.

During the testing for this vulnerability, one issue was found within the API implementation where a certain method was retrieving unnecessarily excessive amounts of information from the backend database on successful API requests. This could have revealed information to the user that was otherwise not available to them anywhere within the client application. The issue was brought up with the developers and fixed within the API as a result.

Another issue was found regarding unsuccessful API requests and the resulting error messages. The issue was related to a raw SQL database error message being returned to the client rather than being replaced with a more generic error message. This particular issue could have potentially revealed details about the underlying database structure and backend configuration. The issue was reported to the developers of the API and fixed as well as improvements made so that this type of error message is never returned to the client and is always replaced with a defined client-side error message or a more generic error response.

6.4 Injection

Injection vulnerabilities are some of the most common web application flaws, where malicious attackers are able to provide user input that is relayed through the application to the backend system causing unexpected behavior [38]. Injection vulnerabilities have been around for a long time, but they are still regularly found within modern applications. These vulnerabilities occur when an attacker is able to manipulate the user-provided parameters of a query in a way that alters the syntax and functionality of the query executed between the application and its backend systems, such as SQL databases the application relies on [39].

Injection flaws can cause many types of harm and they are considered a very serious issue in all web applications, including API implementations. Injection flaws are listed as number 8 within The OWASP Foundation’s API Security Top 10 [33]. The effects of injection flaws can vary depending on the type of attack and the targeted systems. Examples of the effects of these vulnerabilities include attackers executing operating system commands on the system, compromising backend data stores, hijacking user sessions, and impersonating other users or services [38].

There are many types on injection flaws found in web applications. The type of injection flaws that should be considered when reviewing the security of an application depend on the technologies used in implementing the application. Injection vulnerabilities can include attacks such as SQL injection, cross-site scripting, buffer overflow, XML injection, command injection, LDAP injection, HTTP response splitting and file inclusion attacks [39].

Injection vulnerabilities are commonly caused by improper input validation, filtering, or sanitizing; improper authentication and authorization mechanisms, and implementation bugs within the system [39]. One of the most common types of injection flaws is SQL injection, which can occur when the application utilizes user supplied data in constructing its backend database queries.

Let’s assume we have a web application with a backend API that is used to fetch the user’s details while they browse the application. Looking at an example of an API-based SQL injection, lets assume user details are being fetched from the API as follows:

```
GET /api/user/ExampleUser/

API Response:
{
    "LoginName": "ExampleUser",
    "EmailAddress": "example@example.com",
    "FirstName": "Example",
    "LastName": "User"
}
```

The API will receive the GET request from a user and in the backend, the API implementation does an SQL query with the request without proper controls on the client-provided data:

```
userInput = request.uri.segments[ 3 ];
String Query = “SELECT LoginName, EmailAddress, FirstName, LastName FROM
dbo.Users WHERE LoginName = ” + userInput;
accountDetails = Execute( Query );
response.Write( accountDetails );
```

This lack of control on user-supplied data and dangerous concatenation of SQL commands makes the API vulnerable to SQL injection faults. Now, if the user sends a request such as

```
GET /api/user/ExampleUser OR 1=1/
```

The API implementation would now execute SQL query that returns all login accounts in the database:

```
SELECT LoginName, EmailAddress, FirstName, LastName FROM dbo.Users WHERE
LoginName = ExampleUser OR 1=1
```

Protection against injection vulnerabilities can be done in multiple ways. Common solutions for injection prevention include validating, filtering, and sanitizing all user provided data, escaping special characters, and limiting the number of returned records. Validation can often be performed automatically by a trustworthy library, however even those can have bugs that lead to security vulnerabilities. [33] Injection faults may also be defended against by other means, like minimizing system privileges, consistent coding standards, database and web application firewalls, intrusion detection and prevention systems, and anomaly detection systems [40].

6.4.1 M-Files API

Testing the M-Files API implementation against injection vulnerabilities consisted of exploratory manual penetration testing of the API, automated fuzzing of the different endpoints and the development of automated tests to detect any potential injection faults arising from further development of the product.

Manual testing of the API was done utilizing the tools described in chapter 5.4, namely Postman, Burp Suite and OWASP ZAP. For testing the different API endpoints, Postman was configured to utilize both Burp Suite and OWASP ZAP as an HTTP proxy. By sending requests through Postman, they were captured and recorded in the other applications, where they could be repeated with different parameters and fuzzed for potential vulnerabilities. Both Burp Suite and OWASP ZAP offer the ability to inject data in any part of the HTTP request. This enabled the API to be tested against injection vulnerabilities within various points in the request, including the URI, HTTP headers and the request body.

There are a multitude of different injection fault types as covered previously in this chapter. To narrow down the amount of testing required, and to make sure the testing provided actual value, these faults were narrowed down to the ones that could theoretically apply to the system under test. After reviewing the implementation, the chosen injection types to test for included SQL injections, command injections as well as file inclusion and path traversal related injections.

During manual testing and fuzzing of the API, first the API requests were sent successfully through Postman while proxying the traffic through ZAP and Burp applications, where they would be recorded. After this, the recorded requests of different endpoints were taken one at a time and fuzzed with various injection strings utilizing different locations within the request. After the requests were sent the responses from different endpoints were reviewed to be as expected. The system should never interpret the input strings coming from the user as a command that should be executed. Typically, the API responded with a differing error response depending on the type of input string. These error messages had error codes such as 401, 403, 404 and 500. These errors and the error messages that the system returned were reviewed to be appropriate and not contain any suspicious details, or details that would reveal anything about the system to a potential attacker, like a verbose SQL query error.

After manually testing and fuzzing the API endpoints, the test automation coverage of the system was improved to contain tests that would detect any security related regressions related to this issue during the API's future development. The tests operate much the same as manual testing and fuzzing of the API, however, the scale of the testing is not quite the same as exhaustive fuzzing of API endpoints. Careful consideration was done as to which API endpoints should be tested and against which type of injection flaws. The created tests include, for instance, various SQL injection commands in different parts of the HTTP request with differing encoding types. These tests were introduced to the version control CI pipeline and are ran regularly as new functionality is developed to the API.

6.5 Mass Assignment

The mass assignment vulnerability is a web application API related vulnerability, which results from unsafe API operations that accept unintended property values through HTTP requests. Depending on the used language or framework the mass assignment vulnerability may have alternative names, like “autobinding” or “object injection”. [41]

Attackers may utilize mass assignment to edit resource property values which they normally should not have access to by providing additional properties in the HTTP request while making API calls. If the application does not have proper filtering and control for which properties can actually be edited by the API call, the user can do unintended modifications within the backend database and cause major harm as a result.

As an example, suppose an API provides access to a user database where the user details are stored as follows:

```
/api/user/ExampleUser/
{
    "LoginName": "ExampleUser",
    "EmailAddress": "example@example.com",
    "FirstName": "Example",
    "LastName": "User",
    "Role":"user"
}
```

As a part of the normal operation of the web service, the user is allowed to edit some of their details within the web application, which calls the backend API as follows:

```
POST /api/user/ExampleUser/edit HTTP/1.1
{
    "EmailAddress": "example@example.com",
    "FirstName": "Example",
    "LastName": "User"
}
```

Instead of the typical request, the malicious user calls the API directly, or captures and edits the request, and instead sends in the request:

```
POST /api/user/ExampleUser/edit HTTP/1.1
{
    "EmailAddress": "example@example.com",
    "FirstName": "Example",
    "LastName": "User”,
    “Role”:”admin”
}
```

Now, because of insufficient filtering of the properties provided in the request, the user can edit the *role* property within the backend database and change their role from a regular user to an admin user. In this example, the user did privilege escalation through mass assignment, but it may also be possible to do other forms of harm if this vulnerability exists within an application.

OWASP has listed mass assignment as one of the top 10 API vulnerabilities in its’ API Security TOP 10 2019 [33], and multiple popularly used web application related frameworks have published CVEs pertaining to this vulnerability. The issue is also recognized within the Common Weakness Enumeration as *CWE-915: Improperly Controlled Modification of Dynamically-Determined Object Attributes.* [42]

Mass assignment vulnerabilities may be caused by unfamiliarity and insecure use of the utilized frameworks or underlying security issues with the framework itself. As a relatively well-known security issue, several solutions exist to combat this vulnerability, such as allow-lists, block-lists, and data transfer objects. [41] The general idea of all these solutions is to never allow unfiltered user input to be saved into the backend database of an application.

6.5.1 M-Files API

The M-Files API was tested against mass assignment vulnerabilities by doing manual exploratory testing on the different API endpoints and identifying places where the application could theoretically be vulnerable. These endpoints were then targeted with modified requests that would edit data that was normally inaccessible to the client. Responses to the requests and the changes they made to the backend databases were then verified to work as expected.

One issue was identified during the testing for mass assignment vulnerabilities; however, it did not end up being related to an actual mass assignment vulnerability. Regardless, the issue was reported to the developers and fixed.

Test automation coverage was also created to check for this vulnerability in the future as the support and development of the API is an ongoing process. Tests were created to identify if specific API methods allow clients to modify or add data, which they should not be able to edit. New test automation coverage is included within the CI pipeline of the application.

6.6 Security Misconfiguration

Security misconfiguration is an API vulnerability where the system is left in an insecure state due to configuration related issues. These issues may arise from, for instance, utilizing default configurations, incomplete or ad-hoc configurations or overly permissive security policies in the system. Depending on the type of vulnerability, attackers may utilize security misconfiguration related issues to gain knowledge of the system, its' users, or data contained within the application.

Finding security related misconfiguration issues can be difficult, particularly as the scale of the target system grows, and because of this, organizations utilize security scanners and other tools to manage their environment both on the software and the infrastructure level. These tools may, for instance, detect misconfigurations in server infrastructure and find vulnerable dependencies or integrations. [43]

There may also be difficulty in finding the right tools, and there needs to be clear instructions as to how they're being used and who is responsible for maintaining the system from a security perspective.

This chapter will describe several places where a security misconfiguration vulnerability may arise from.

6.6.1 Unpatched Systems

Unpatched systems may allow attackers to utilize known security flaws within a specific version of a given system or its' component. All backend systems should be kept up to date, as old software versions may contain vulnerabilities that allow potential attackers to exploit the system in various ways. Regular updates or automated updates are necessary to keeping any system secure in the long run.

6.6.2 Unprotected Files and Directories

Unprotected API endpoints, files or directories may make an API vulnerable to attackers. API implementations should have proper access controls in place for each API endpoint as well as all the backend systems that are used for providing the service. API users should only be able to access those resources for which they have been properly authorized. For instance, if a web application is utilizing an API which does not have proper access control implemented, a potential attacker may have access to other customer's data or administrative tools of the application.

6.6.3 Unhardened Images

Unhardened systems may pose a security risk to an API service. Proper system hardening should be done in order to protect the system from potential attackers by eliminating potential attack vectors and minimizing the system's attack surface. System hardening is a wide concept and good procedures for it include things such as keeping the system updated, developing strong access control mechanisms, utilization of firewalls, closing unnecessary ports, intrusion detection and prevention systems, and disabling unnecessary services. [44]

6.6.4 TLS Misconfiguration

Misconfigured transport layer security can cause an API service to be unprotected against malicious attackers. TLS may be missing entirely, or it can be outdated or misconfigured leading to the application being vulnerable. TLS has been updated throughout its' use and has multiple versions available, some of which are no longer considered secure. API implementations, just as any other web service, should utilize the newest versions of TLS which are considered secure. The newest version of TLS is 1.3 which was published in 2018, improving upon TLS 1.2 which is still considered secure [45]. Older TLS versions 1.0 and 1.1 have been proven to contain vulnerabilities and should no longer be used anywhere. Insecure TLS versions may lead to man-in-the-middle attacks between a user and the web service. [46]

6.6.5 CORS Policy and Security Headers

Missing or misconfigured CORS policies and security headers may pose a risk to any web service including API implementations. This chapter takes a look at the same-origin policy, CORS, and other HTTP security mechanisms.

Same-origin policy is a security mechanism built into modern web browsers that limits how documents and scripts can interact with other web servers. It limits possible attack vectors and isolates malicious web pages by preventing the malicious web application from interacting with other services used within the same browser session. [3]

Without the same-origin policy in place, a malicious website could run JavaScript within the browser to interact with other services, and combined with cookie-based authentication, exfiltrate sensitive data or perform operations on behalf of the user without their knowledge. The same-origin policy makes it safe for users to browse trusted and untrusted sites through multiple tabs within the same web browser session.

When the same-origin policy was originally introduced websites were much simpler in nature. Nowadays, web services have the need to interact with other services like external APIs and for this reason there have been a number of workarounds introduced to loosen the policy when appropriate.

One of these workarounds is Cross-Origin Resource Sharing (CORS), proposed originally in 2004 and accepted as an official W3C recommendation in 2014. CORS is a set of server response headers web applications can utilize to whitelist communication between different domains. CORS can be used among other things to whitelist request headers and methods, control cookie-based authentication and caching. [47]

One of the most important limitations of CORS policies is related to the combination of Access-Control-Allow-Origin and Access-Control-Allow-Credentials headers. CORS header Access-Control-Allow-Origin is a header which lists the trusted domains for which the same-origin policies are ignored. Access-Control-Allow-Origin specifies whether or not browsers are allowed to access the resource from a certain origin. Access-Control-Allow-Credentials header indicates if the web server allows cross-domain requests to utilize cookie-based authentication with requests. [3]

CORS policies limit the use of the forementioned headers such that setting "Access-Control-Allow-Credentials: true" fails if server has a wildcard policy in place, i.e., "Access-Control-Allow-Origin: *". However, CORS header misconfigurations exist, which may allow this combination of headers to be used. One of such configurations is utilizing a "reflective" wildcard policy, where a server will whitelist certain domains based on the requests it receives. [47]

The HTTP Strict Transport Security header (HSTS) is used to inform browsers that the site should always be accessed utilizing HTTPS. By setting the HTTP header "Strict-Transport-Security: max-age=<expire-time>", the user's browser will remember that the site is only accessible with HTTPS and will always utilize encrypted communication regardless of how the site was accessed. After the user has received this information, any insecure links to the site, or direct attempts to utilize HTTP communication, will result in the user being redirected to using HTTPS instead.

HSTS is particularly useful for preventing possible man-in-the-middle attacks, where a potential attacker could capture the unencrypted HTTP request and redirect the victim to a false site where any they might enter sensitive data which will then be captured by the attacker. [3]

The Content Security Policy (CSP) header is an HTTP header, which when served with a web page allows the server provider to control how resources on that page can be loaded. [3] It functions as an added layer of security for web services against potential cross-site scripting and data injection attacks. Among other security policies, CSP makes it possible to reduce potential attack vectors by specifying what are valid sources for executable scripts.

For instance, Mozilla recommends setting the following CSP headers for API services: "Content-Security-Policy: default-src 'none'; frame-ancestors 'none'". This will disable any resources from being loaded as well as prevent the site from being embedded in HTML elements like "<iframe>". [48]

6.6.6 Verbose Error Messages

Excessively verbose error messages are a recognized software weakness that is often caused by security misconfiguration and poor error handling. Malicious users may force error messages to collect any information contained within them. If error messages are unhandled and users are returned too much information, an attacker may obtain details about how the underlying components of the system operate. Verbose error messages may reveal things such as stack traces, backend system information, or details related to database structure.

A good REST API implementation will respond with the correct HTTP status code and minimum required error messages to the end user. Stack traces should never be revealed as a result of errors. This way, implementation related details of the application will be kept secure from potential attackers. [49] This topic was also touched on earlier in chapter 6.3 while discussing excessive data exposure.

6.6.7 Vulnerable Dependencies

The use of vulnerable dependencies can be a critical security issue in today's systems. As a product grows larger and larger, the number of third-party dependencies and integrations grows, and organizations need to be aware and manage the risk that comes along with their use. Vulnerable and outdated components are well recognized as one of the highest security risks for web applications, but they relate to the underlying APIs as well, as they are a major part of today's web application implementations.

According to the OWASP Application Security Verification Standard (ASVS), outdated and insecure dependencies are the root cause of some of the largest most expensive attacks to date. OWASP also recommends that all components of an application are kept up to date, unnecessary features and configurations are removed, third-party assets are retrieved from a trusted secure provider, like a private content delivery network (CDN), and all third-party dependencies are listed in a software bill of materials (SBOM). [50]

The easiest and most reliable solution to ensure the secure use of third-party dependencies is to have automatic dependency checks in place within the build pipeline of an application. This way insecure dependencies can likely be detected and fixed before they get to the production version of an application. If the process is not automated and relies on irregular manual operations from developers, it may easily be forgotten or skipped due to increased workload. An automated dependency checking process is also recommended by ASVS [50].

There are multiple commercial and open-source tools available for running automated dependency checks with differing code language support capabilities and pipeline integrations. For instance, Snyk and Synopsys offer commercial tools for dependency checks and OWASP Depedency Checker is an open-source project that does the same. The basic idea of all of these tools is to automatically detect all components used within a given system and query them against a database of known vulnerabilities, such as the Common Vulnerabilities and Exposures (CVE) operated by The MITRE Corporation [51].

6.6.8 Unenforced HTTP Methods

HTTP protocol supports various methods that are intended for different types of operations. API implementations should make sure that these methods are enforced, as unenforced HTTP methods may cause issues when REST API endpoints are targeted with HTTP methods other than the ones originally intended.

Failure to close unnecessary services or HTTP methods is a common API security misconfiguration issue, which can cause harm as it may allow a malicious attacker to find, modify or delete sensitive information in ways that they shouldn't be able to. [52]

HTTP methods are also recognized as a potential security risk by The OWASP Foundation and their guide on web application security testing contains steps which can be taken to ensure unintended HTTP methods are protected against [53].

HTTP endpoints should be enumerated and tested against supported as well as unsupported HTTP methods to ensure their expected functionality in all cases. If a method is not supported, the web service endpoint should result in an error. According to RFC 7231, if a method is unrecognized or not implemented the server should respond with HTTP status code 501 (Not Implemented). If the method is supported but the caller is not authorized to access the resource the response should be 405 (Method Not Allowed). [2] This is recommendation is good practice, however, in reality applications may differ from this and implement their own procedures for handling these errors. For instance, the API may display an error caused by an unauthenticated or unauthorized access attempt or alternatively the user may be redirected to a login page. If the service endpoint responds with something other than an error or a redirect, the service may be vulnerable to the bypass of authentication or authorization. [53]

6.6.9 M-Files API

Each of the security misconfiguration examples described earlier in this chapter were reviewed for M-Files Cloud API to see if they apply to their product. When it comes to unpatched systems, the studied API implementation was considered to be secure, largely due to the use of various cloud services. M-Files API utilizes various Azure components heavily in its implementation, namely Azure SQL, Azure Functions, Azure Front Door, and Azure Web Application Firewall.

Azure SQL is a platform as a service (PaaS) database engine that is automatically kept up to date with the latest updates by Microsoft. Azure SQL always runs the most up-to-date version of Microsoft SQL Server and updating is done automatically by Microsoft. [11]

Azure Functions is a form of serverless computing services provided by Microsoft. The Azure functions runtime as well as the actual servers that process the computations are updated and maintained by Microsoft. [11]

Azure Front Door is a web application entry-point service, which efficiently routes client requests to application backend services. It works on Microsoft's global edge network and is used to create scalable web services. [11]

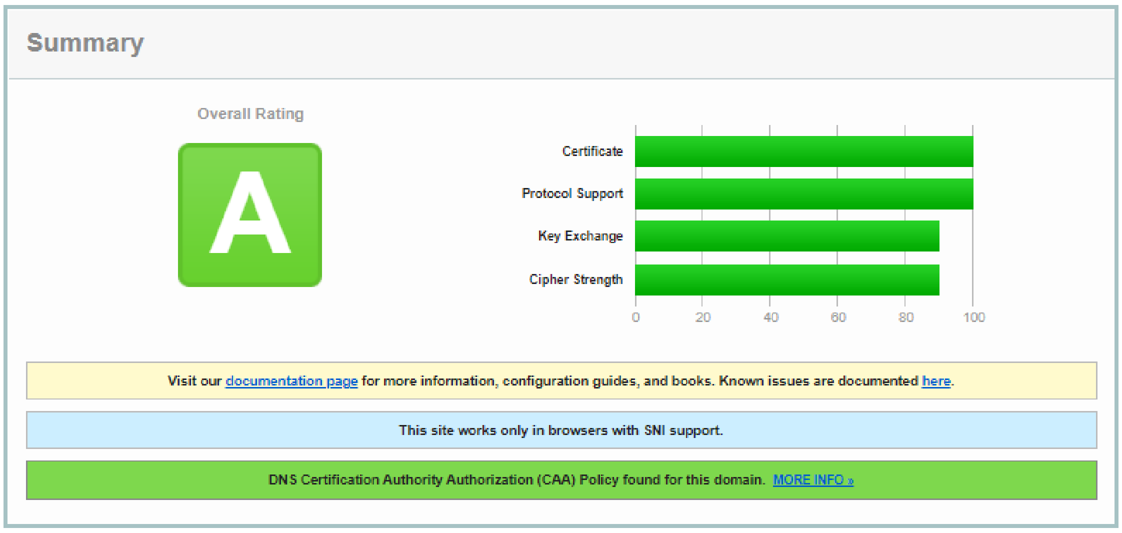
Azure Web Application firewall protects web applications from malicious attacks. It contains Microsoft's own regularly updated preconfigured rules used to protect web applications from attacks such as SQL injection and cross-site scripting, as well as allowing users to configure their own rules. [11]

Due to the heavy utilization of cloud services, unpatched systems are not a major concern when it comes to M-Files API implementation. If a system goes out-of-date and causes security issues, it will affect other Azure customers as well.

M-Files API was tested for the unprotected files and directories vulnerability by ensuring that the correct authentication and authorization mechanisms were in place. All M-Files Cloud Management API endpoints require both an API key and valid JWT authentication, which are provided in the HTTP headers of every API call. In addition to authentication, there are access right checks in place to make sure an authenticated user is properly authorized to access a given resource. For certain API functionality, IP-based restrictions are also in place.

For the unhardened images issue, the components of M-Files Management API were researched for potential misconfiguration issues. All cloud resources of the API are behind multi-factored AD authentication as well as IP-based restrictions. In addition, access to the resources is heavily limited and access to the resources are only granted to those who require it. In addition, the API services have firewalls that block malicious attackers and alerts are configured to notify of unexpected behaviour in the systems.

TLS security for the M-Files API was tested by running an automated security scan on different API environments. Qualys SSL Labs scan offers a way to check the configuration of any SSL web server against weak TLS configuration [54]. It provides an easy way to check for outdated TLS versions and weak ciphers, among other potential issues. SSL Labs scan gives as a result a graded report for the scanned target and points out if there were any issues or potential for improvement.



Visit our documentation page for more information, configuration guides, and books. Known issues are documented here.

This site works only in browsers with SNI support.

DNS Certification Authority Authorization (CAA) Policy found for this domain. MORE INFO »

Figure 11: SSL Labs Scan Result

As a result of these scans, M-Files API received a good grade, and it was verified that only secure TLS versions and ciphers were being utilized. M-Files API did not support insecure TLS versions such as 1.1. TLS 1.3 was not in use within the API implementation, which prevented the product from reaching the best possible grade. However, TLS 1.2 is considered to be secure, and TLS 1.3 support is not particularly difficult, if required in the future.

CORS policy settings and used security headers within M-Files API were reviewed as a part of testing for security misconfiguration issues. API headers were inspected manually by studying requests proxied through Burp Suite, as well as running an automated security headers scan tool offered by Probely [55]. The automated scan resulted in a bad grade and pointed out that certain security headers were not in place, however after further research, there were good reasons for why those headers were not necessary for the API implementation.

The API had all necessary CORS headers in place to enable the communication between it and the web applications that utilize it. M-Files API did not have CSP headers in place, however these were deemed not to be necessary, as the API cannot load resources from any untrusted sources. However, Mozilla does recommend these headers for REST APIs within their web security guide [48]. HSTS headers were also not in use, however attempting to utilize the API with HTTP will not work and results in an HTTP redirect response that automatically directs the user to use HTTPS instead. HSTS was considered not to be important, as users never call the API directly using their browser. For a potential man-in-the-middle attack to succeed as a result of missing HSTS headers, the original user would likely also be attempting something malicious.

Verbose error messages aren’t a major concern in M-Files Management API as system level error messages are never returned to the user as a result of their API calls. If an error occurs while processing an API request, the user will be returned a generic error message as opposed to a stack trace that might contain any revealing details about the underlying system or the API implementation itself. Unsuccessful requests are recorded elsewhere and the detailed errors they cause are only visible to those working in development and support tasks related to the API solution.

M-Files API was tested against vulnerable dependencies by running an automated dependency scanning tool, which went through all of its’ dependencies and queried them against the CVE vulnerabilities database. The API solution contained several hundreds of dependencies and within these there were two potential issues, however neither of these had a direct upgrade path that would fix the potential vulnerability. Both issues were brought by third -party components that utilized the vulnerable dependencies. Since there were no direct upgrades to fix the potential vulnerabilities, the issues were looked into and found to be relatively minor and likely to not appear in the product. Dependency related vulnerability scanning is not currently a part of the automated CI pipeline in the API development, but it was noted down as a future improvement and will be added at a later date.

Lastly, the API was studied for the security misconfiguration of unenforced HTTP methods. The API's endpoints were tested to not accept unintended HTTP methods. Any request with the wrong HTTP method resulted in HTTP error code 404 not found. Automated tests were added to the CI pipeline of the API as a part of this thesis and these tests will reveal if this becomes an issue in the future. In addition to manual testing and test automation, the underlying source code of the API implementation was reviewed and if the current practices are followed in the future, this vulnerability should never occur.

6.7 Insufficient Logging and Monitoring

Logging is an important tool in any system and especially when it comes to security critical applications. Logs are a record of events that occur within a system. Logs are composed of log entries, which contain varying types of information related to an event that has occurred within the system. [56]

An event is a single occurrence within a system that includes a notion of time, the occurrence and other details related to the event and the environment in which it occurred. [57]

Typically, any system will have multiple different types of events and they are often categorized in various ways. For instance, system events, user related events and security related events may each have their own categories. Events may be generated by the system itself or as a result of user interaction with the application.

Application logging is a crucial component of a system when it comes to security events. Logs can provide invaluable data for identifying problems and security incidents, monitoring the status of the system, non-repudiation of events, and performance monitoring among other things. [58]

Establishing good logging practices for an application can be a difficult thing as at the time of developing the application, it may not always be clear what kind of logs are needed once it is taken into use. However, logs are important as they are often key to investigating issues within the system. Insufficient logging and log monitoring practices are a recognized vulnerability for API applications as well [33].

When thinking about good logging practices, it’s useful to keep in mind what the goal and primary purpose of logging actually is. Logs should be able to answer questions such as:

- What happened within the system?
- When did this event occur?
- Where in the system did the event occur?
- Who was involved in the event?
- Where did the involved user come from?

These are considered to be essential for any logging systems. In addition to the above, there are still other questions that an investigator would like to know, particularly in case of a security incident. Improving upon the previous, additional questions logs can help to answer are:

- Where do I get more information about the event?
- What is affected by the event?
- What could happen next?
- What else happened?
- What should be done about the event?

Being able to answer these questions form a good foundation for an effective logging system. However, the focus that should be put on a specific part of a logging system heavily depends on the type of application. [59] [60]

When it comes to application logs, they can loosely be divided into two types of logs: application logs and debug logs. Audit logs are meant for security teams and auditors and should always be enabled as they contain information related to attacks, faults, and user activities within the system. They should also be kept for a longer period of time in order to facilitate proper investigation following a security incident. [60]

Debug logs are intended for system operators and application developers and may not always be enabled, as they are not always necessary. They contain information about failures and errors within the system and are typically useful only for a limited period of time, as they are used to hunt down and fix bugs within the system during its development. [60]

As an example of what specific types of events should be logged, one can look at critical log entries that require immediate attention such as critical faults within the system, successful attacks, resource capacity, major system configuration changes, crashes, failed login attempts and unauthorized connections. There are also less critical accounting log entries like system status messages, blocked low impact attacks, start up and shutdown messages, successful logins, and routine configuration changes. [60]

Application logging does come with its own challenges. As the scale and complexity of a system grows and more events are being logged the management and usability of the logs becomes more and more challenging. Challenges related to application logging include the logs not being actively monitored, logs not being stored for long enough, difficulty to analyze logs, lack of prioritization, and logging only being enabled on some level of the system.

In order to retain control and usability, the logs should have some form of prioritization applied to them. One should log as much as they can and retain most of what they log, while actively analyzing and monitoring an even smaller portion of the logs [60].

As with any system, a logging system may also be attacked. This is not an uncommon occurrence, as the attacker may attempt to hide what they're doing within the system. Looking at the CIA triad, it is possible to examine how a malicious attacker could hurt a logging system. An attacker can exploit the confidentiality of a logging system if they are able to read sensitive log data. An attacker may attempt to hurt the integrity of the logs by altering, corrupting, or inserting false log data. The availability of the logs may also be in danger if an attacker is able to delete log data or disable the logging entirely.

When it comes to API implementations, they may be vulnerable, for instance, if the system does not produce logs at all or the logs are not sufficient, the log entries are not detailed enough, the integrity of the logs cannot be guaranteed, or log monitoring is not taken seriously. [33]

6.7.1 M-Files API

As a part of this thesis, M-Files Cloud Management API was reviewed for this vulnerability and the goal was to find out if insufficient logging and monitoring applies to the systems involved within the M-Files API solution.

Reviewing the logging practices consisted of going through all the different components within the system and verifying that logging was enabled, and appropriate settings were in place. As the API implementation consists of multiple components, it was key to ensure logging was in place on every level of the system, from the front-end entry-point all the way to the underlying backend database implementations.

The systems were tested by processing both successful and unsuccessful requests and reviewing what information was being logged, where it was logged and how it was stored. By using the application and doing various operations log events were being created and afterwards it was reviewed that there was enough information to trace what was happening in the system and who was behind these events.

After reviewing the logging practices, the product appeared to contain all the necessary information for investigating potential attacks against the API. Both successful and unsuccessful attacks leave enough of a trace in the system to reveal who was behind the attack and how an ongoing attack could be mitigated.

Log monitoring practices related to the API implementation were also reviewed, including configured alerts and log retention policies. Potential threats to the logging systems were also investigated, including whether or not the logs could be leaked, corrupted, deleted or disabled. The threats of alerts not being enabled, or triggered alerts being ignored were also considered. Some improvements related to log monitoring practices were proposed as a result of this investigation and have been taken to use as a result.

Overall, M-Files Cloud Management API can be considered to be sufficiently protected against both *CWE-778: Insufficient Logging* [61] and *CWE-223: Omission of Security-relevant Information* [62], as long as logs remain available, they're stored long enough for investigation, and necessary alerts are enabled.

6.8 Improper Assets Management

Improper assets management is a vulnerability in which an API system is left exposed to potential attackers due to inadequate practices in maintaining an organization’s API inventory. It is recognized as one of the most common and severe security issues regarding APIs by The OWASP Foundation and can have a severe impact on the security of sensitive data contained within the system or on the system itself [33].

Organizations can maintain several large-scale API systems, and this can cause issues when managing the API inventory. Old unpatched and forgotten APIs or API endpoints can cause a considerable security risk to the organization as they may contain unintended functionality, lacking security configurations or unpatched software that can provide an easy entry point for a potential attacker. During API development organizations may also maintain multiple versions of the same API for different purposes, like testing environments, development environments as well as earlier versions or beta releases. These can be a major security concern if not properly managed.

In order to combat this vulnerability, organizations should keep a clear separation of production and non-production environments. API versions maintained for purposes like testing or development should never interact with actual customer data like their production versions. The production API and its databases should have no connection any non-production environments to ensure the safety of the sensitive data contained within. API versions used for development operations may contain, for instance, bugs or default configurations that leave the system vulnerable to potential attackers.

In addition to separating the different environments, the access to each system should be properly controlled. Each environment should have their own access policies and only authorized personnel should be able to access each environment. For instance, development and testing operations should be kept separate from any production environments, unless absolutely necessary. Unless the goal of the API system is to be freely accessible, all API environments should require authentication and not be accessible to anyone. Security by obscurity is a bad policy and organizations should not rely on no one finding the hosted non-production environments.

Documentation is also key in API development. It is critical to have the entire API inventory properly documented, as the personnel working with the API can change within an organization, and without documentation, this can lead to unmanaged and forgotten APIs or endpoints. This should include an update policy for the APIs as well as a retirement plan for those services which are no longer in active use.

6.8.1 M-Files API

The M-Files API was reviewed regarding this vulnerability, and it was verified that proper API management practices were being followed. Proper separation of production and non-production environments including access control to different systems was being maintained. Security control mechanisms were in place for all existing environments and proper documentation was being kept.

6.9 Lack of Resources and Rate Limiting

API implementations that are not protected with rate limits and denial of service (DoS) mitigation may be vulnerable to various attacks such as denial of service and brute force attacks. An unprotected API can receive an excessive number of requests that cause the API backend implementation to experience heavy CPU load, memory, or storage issues. With a cloud-computation-based API implementation this can also lead to excessive costs arising from unnecessary request processing.

Common protection mechanisms for these types of vulnerabilities include rate limiting based on identifying the attacker by their client IP address, API key or other authentication mechanisms. APIs can also impose IP restrictions and maximum HTTP request payload sizes to further protect themselves against attackers.

6.9.1 M-Files API

Testing M-Files API for lack of resources and rate limiting consisted of reviewing what forms of protection were in place and simulating a denial-of-service attack against the system to find what affect it would have on the system.

Simulating a denial-of-service attack scenario was done by running repeated DoS attacks with Burp Suite’s Intruder tool against various API endpoints. The testing was done in a separate testing environment so as not to cause possible harm to active users of the API or the services that rely upon it. Using the Intruder tool, requests of various sizes were generated and sent repeatedly at a higher rate than can ever be expected to need to be handled by the system under its normal operation. A single Burp Suite instance supported sending up to 999 concurrent requests. With the delay between the client sending the request and the server returning the response, this usually resulted to be around 600–700 requests per second, however the actual number of requests per second depended upon which API endpoint was being targeted.

M-Files Cloud Management API contains IP-based rate limiting. The API implementation was tested both with and without rate limiting restrictions in place, as an authenticated user and an unauthenticated user. If rate limits were in place, the user was limited to a certain number of requests per minute, and any requests that exceeded that limit were not processed and a HTTP 403 response was returned to the client. Unauthenticated users are blocked from accessing the API early on in the handling of the requests and as such do not cause increased load or slow down the system.

The API was able to handle the stress of the simulated denial of service attack in all scenarios and there was no visible slowdown of the system even without rate limits in place, likely because of the reliability and performance brought by the automatically scaling Azure Functions cloud service it is based on.

As a result of this testing, it was revealed that the tested API did not limit the HTTP payload size of the requests in any manner, and it was possible to send requests with a large JSON body that the API would process for a long period of time. Verifying this issue, the API was called with a request body JSON containing 1.2 million rows and the processing of that requests took several minutes. This improvement has since been acted upon and the issue has been fixed. Another improvement that was suggested is that the API would move to tracking users based on their authentication, as a real DoS attack would likely come from a single authenticated user that is using multiple IP ad-dresses.

7. RESULTS

Security testing for the M-Files Cloud API was done through manual evaluation of recognized API security vulnerabilities utilizing the existing tools described in chapter 5. In addition to manual testing, these tools also offered automated scans that were utilized in evaluation whether or not a specific vulnerability exists within the system under test. Manual testing of the API resulted in several findings in the form of potential security violations within the API and suggested improvements.

The goal of this thesis was also to improve the existing test automation coverage to account for security issues in the future. This way the results of this thesis provide continuous value in the API development process going forward. The test automation improvements made during work on this thesis aim to detect introduced vulnerabilities, help increase efficiency and minimize wasted development time.

7.1 Automated Security Testing Improvements

Test automation coverage of the M-Files API code project was improved during this thesis by introducing new security-focused test cases and improving existing test cases. The test cases were added to an existing test project written in C# using the NUnit Framework [63]. The test cases within this project are executed regularly and whenever changes are made to the API.

Security focused test case count within the test automation project was increased from 231 to 4556 unique test cases. Test cases brought both new coverage for security issues and improved coverage for issues such as: SQL injections, path traversal, API key usage, authentication tokens, authorization, mass assignment vulnerabilities, and API enumeration. Testing for these issues was done in the URL paths of API requests, query parameters, request headers and request body.

Typically, the test cases send customized requests to the API and evaluate the responses to see if the response is as expected. If unintended changes are made to the API that change the functionality, the test cases will fail, and the failures can be investigated to see if a potential security issue has been introduced with the changes.

The overall number of new test cases in the project could have been increased much further, however, instead of aiming to do extensive fuzzing of the API, it was decided that the test automation should rather focus on specific and realistic issues that could potentially threaten the security of the system. Instead of creating tens of thousands of new test cases, the test case count was kept relatively low and the value of each test case higher. This helped to keep the overall run time of the test project in control.

7.2 Issues and Improvements

The testing done during this thesis resulted in reporting 10 security-related issues to the developers of the API system. These issues have since been dealt with and fixed within the API implementation.

The breakdown of the reported issues within the API is as follows: 1 issue related to security misconfiguration, 3 injection issues, 1 issue related to broken authorization, 1 issue related to excessive data exposure, 1 rate limiting issue, 1 case of improper assets management and 2 cases of unintended API functionality.

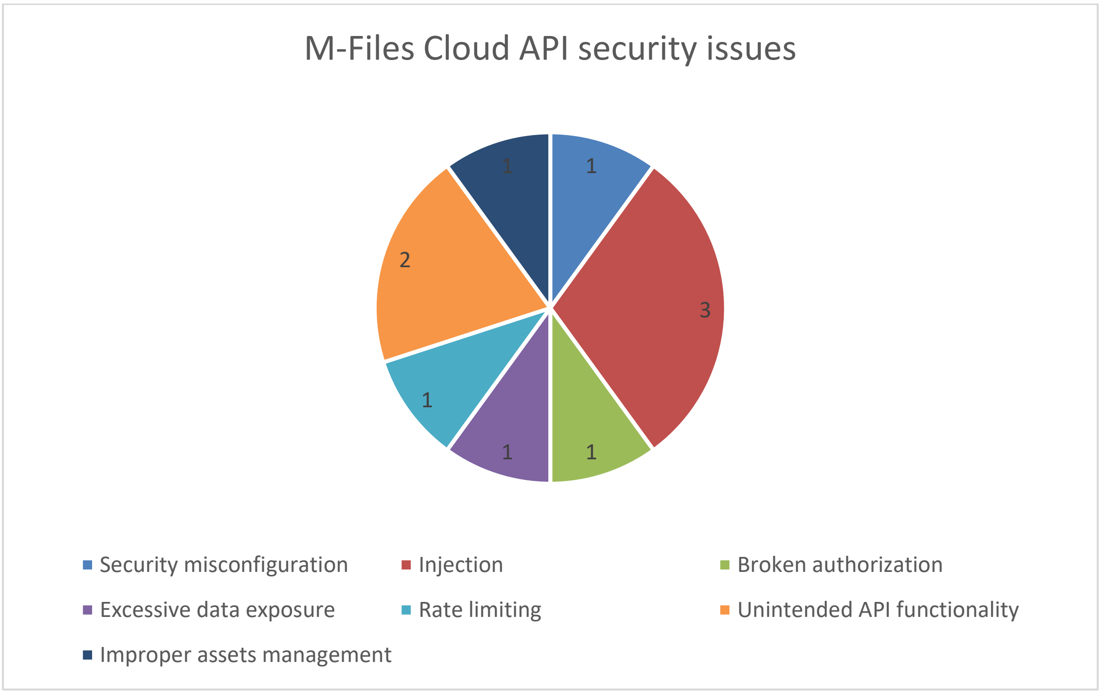


Figure 12: Discovered API issues

In addition to the issues found within the API implementation, potential security improvements were also noted for the future development of the API. These were related to logging policies, security alerts, and rate limiting policies.

8. CONCLUSIONS

The purpose of this master’s thesis was to study and identify known API vulnerabilities as well as to find ways that an existing API solution can be tested for those vulnerabilities. Extensive security testing was during the making of this thesis for the M-Files API solution, resulting in uncovering previously unknown issues within the implementation, proposing new improvements and improving the security test automation coverage. The goal of this chapter is to give answers to the primary research questions presented in this thesis.

8.1 What are the most significant vulnerabilities and attacks against REST API implementations?

During this thesis, research was done on API security related papers and other sources to find common vulnerabilities which the M-Files system should be evaluated against. In researching the material for this thesis, it was found that the list of API-specific security vulnerabilities produced by The Open Web Application Security Project was often referred to in other sources, and it contained the most critical and commonly found API security related vulnerabilities.

The API vulnerabilities discussed in detail during this thesis were:

- Broken authentication
- Broken authorization
- Excessive data exposure
- Injection
- Mass assignment
- Security misconfiguration
- Insufficient logging and monitoring
- Improper assets management
- Lack of resources and rate limiting

In chapter 6 of this thesis, research was done into how these common vulnerabilities may occur, why they are common, what can cause them, and for each issue testing was done to study if the M-Files API implementation was found to be vulnerable.

8.2 How is the M-Files API implementation protected against API vulnerabilities?

The M-Files Cloud API is currently in use as well as an ongoing development effort. It is used to handle sensitive data and perform important operations for the normal function of customer's data storage solutions. The importance of security for this product cannot be overstated.

In the research and testing done during this thesis, several previously unknown issues were uncovered, reported, and fixed within the API implementation, improving the overall security of the system. The total number of reported security issues in the API implementation during this thesis was 10. Additionally, some other security-related improvements were proposed to be potentially implemented at a later date.

Despite resulting in multiple findings of different types, discussed in detail in chapter 7.2, the overall severity of these issues was considered to be relatively low, and for an attacker to have taken advantage of these issues, they would have to have had a deep understanding of the system and the M-Files product as a whole.

From a security standpoint, the heavily cloud-based M-Files API implementation was found to be in a positive state overall, and good development practices were being followed during its development. However, as the findings of this thesis suggest, there is still room for some improvement, especially if the scale of the API continues to grow and its role becomes more important.

Security-related unit test automation coverage of the API solution was vastly improved during this thesis and is currently at an acceptable level to detect the most common API issues that can be tested for in this way. Further investments towards security of the M-Files API solution could be targeted towards the integration of other security solutions, such as code analysis tools to automated build pipelines of the system.

8.3 How can the development process ensure the security of a REST API?

Steps to the ensure the security of a software product should be taken early in its development process. The process can start with initiating a threat modelling and risk analysis processes, which will then help recognize potential security risks and vulnerabilities that should be taken into account during the actual coding process.

The development of a REST API should have a heavy focus on testing for potential security threats. The testing can be done in various ways through manual and automated methods. Security of an API implementation can involve, for instance, unit testing and the utilization of security testing tools, such as code analysis tools and fuzzing tools.

To ensure the security of a system throughout its continuing development and maintenance work, the security testing process should be automated as much as possible. In practice, this should involve integrating as much of the testing as possible into the CI/CD process of an application.

Security of the development process may also be improved by educating the developers of the system. The developers should have knowledge of common security risks, vulnerabilities, and bad coding practices. This way many of the potential security issues can be prevented as early as possible within the development process, improving its overall efficiency. It is also good practice to have dedicated security and quality assurance experts working on and testing the product, as they can often provide valuable feedback to the developers of the system.

8.4 Value of this thesis

During this thesis, in depth research was done on different API security vulnerabilities and the M-Files API was thoroughly tested against them. As described in earlier chapters, this study resulted in multiple findings within the system. These findings were reported to the developers of the API and fixed as a result, improving the overall security of the M-Files Cloud API solution. Without this thesis, these findings may not have been made and further functionality may have been built upon vulnerable components.

Through the research done for this thesis as well as discussing and fixing the issues found during this thesis, the development and quality assurance teams at M-Files were provided with new knowledge and material to further improve the overall development operations from a security point of view.

9. REFERENCES

[1] D. Gourley and B. Totty, HTTP: The Definitive Guide, Sebastopol, California: O'Reilly, 2002.

[2] R. Fielding and J. Reschke, Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content, Internet Engineering Task Force, 2014.

[3] Mozilla Corporation, "MDN Web Docs," [Online]. Available: https://developer.mozilla.org/en-US/. [Accessed 30 January 2022].

[4] A. P. Felt, R. Barnes, A. King, C. Palmer, C. Bentzel and P. Tabriz, "Measuring HTTPS Adoption on the Web," in *Proceedings of the 26th USENIX Security Symposium*, Vancouver, 2017.

[5] T. Berners-Lee, L. Masinter and M. McCahill, "Uniform Resource Locators (URL)," December 1994. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc1738.

[6] R. T. Fielding, Architectural Styles and the Design of Network-based Software Architecture, Irvine: University of California, 2000.

[7] ProgrammableWeb, "JSON is Clearly the King of API Data Formats in 2020," 3 April 2020. [Online]. Available: https://www.programmableweb.com/news/json-clearly-king-api-data-formats-2020/research/2020/04/03. [Accessed 12 February 2022].

[8] M. Jones, J. Bradley and N. Sakimura, "JSON Web Token (JWT)," May 2015. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7519. [Accessed 5 March 2022].

[9] M-Files Corporation, "Intelligent Information Management," [Online]. Available: https://www.m-files.com/. [Accessed 22 January 2022].

[10] M-Files Corporation, "M-Files System Overview," [Online]. Available: https://www.m-files.com/user-guide/2018/eng/system_overview.html. [Accessed 8 April 2022].

[11] Microsoft Corporation, "Microsoft technical documentation," [Online]. Available: https://docs.microsoft.com/en-us/. [Accessed 5 March 2022].

[12] M-Files Corporation, "M-Files Knowledge Base," [Online]. Available: https://kb.cloudvault.m-files.com/. [Accessed 20 December 2021].

[13] A. B. S. R. I. H. Touhid Bhuiyan, "API vulnerabilities: current status and dependencies," *International Journal of Engineering & Technology,* vol. 7, no. 2.3, pp. 9-13, 2018.

[14] D. Gruber, "Modern Application Development Security," Enterprise Strategy Group, 2020.

[15] Akamai, "API: The Attack Surface That Connects Us All," *Computer Fraud & Security,* vol. 2021, no. 9, pp. 1-20, 2021.

[16] A. Munsch and P. Munsch, "The Future of API (Application Programming Interface) Security: The Adoption of APIs for Digital Communications and the Implications for Cyber Security Vulnerabilities," *Journal of international technology and information management,* vol. 29, no. 3, pp. 24-45, 1 July 2020.

[17] A. Singh and K. Chatterjee, "Cloud security issues and challenges: A survey," *Journal of Network and Computer Applications,* vol. 79, pp. 88-115, 2017.

[18] T. Bar, "Notifying our Developer Ecosystem about a Photo API Bug," Meta, 14 December 2018. [Online]. Available: https://developers.facebook.com/blog/post/2018/12/14/notifying-our-developer-ecosystem-about-a-photo-api-bug/. [Accessed 5 March 2022].

[19] R. Kovatch, "InfoSec Write-ups," Medium, 27 October 2020. [Online]. Available: https://infosecwriteups.com/the-youtube-bug-that-allowed-uploads-to-any-channel-3b41c7b7902a. [Accessed 5 March 2022].

[20] C. Osborne, "GitLab patches Elasticsearch private group data leak bug," 7 October 2020. [Online]. Available:

https://www.zdnet.com/google-amp/article/gitlab-patches-elasticsearch-private-group-access-leak/. [Accessed 5 March 2022].

[21] M. Zhao, A. Laszka and J. Grossklags, "Devising Effective Policies for Bug-Bounty Platforms and Security Vulnerability Discovery," *Journal of Information Policy,* vol. 7, no. 1, pp. 372-418, 2017.

[22] M. Felderer, M. Büchler, M. Johns, A. D. Brucker, R. Breu and A. Pretschner, "Security Testing: A Survey," *Advances in Computers,* vol. 101, pp. 1-51, 2016.

[23] I. Schieferdecker, J. Grossmann and M. Schneider, "Model-Based Security Testing," *Electronic Proceedings in Theoretical Computer Science,* vol. 80, pp. 1-12, 2012.

[24] Microsoft Corporation, "Microsoft Security Development Lifecycle," [Online]. Available: https://www.microsoft.com/en-us/securityengineering/sdl. [Accessed 7 March 2022].

[25] Y. Pan, "Interactive Application Security Testing," in *International Conference on Smart Grid and Electrical Automation (ICSGEA)*, 2019.

[26] PortSwigger, "Burp Suite - Application Security Testing Software," [Online]. Available: https://portswigger.net/burp. [Accessed 19 April 2022].

[27] The OWASP Foundation, "OWASP Zed Attack Proxy (ZAP)," [Online]. Available: https://www.zaproxy.org/. [Accessed 5 March 2022].

[28] A. Tyler, "The JSON Web Token Toolkit v2," [Online]. Available: https://github.com/ticarpi/jwt_tool. [Accessed 30 January 2022].

[29] The MITRE Corporation, "CVE-2015-2951," 7 April 2015. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-2951. [Accessed 19 April 2022].

[30] The MITRE Corporation, "CVE-2016-10555," 29 October 2016. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-10555. [Accessed 19 April 2022].

[31] T. McLean, "Critical vulnerabilities in JSON Web Token libraries," Auth0, 21 8 2020. [Online]. Available: https://auth0.com/blog/critical-vulnerabilities-in-json-web-token-libraries/. [Accessed 19 April 2022].

[32] GitHub Security Lab, "Security Code Scan - static code analyzer for .NET," [Online]. Available: https://security-code-scan.github.io/. [Accessed 30 January 2022].

[33] The OWASP Foundation, "OWASP API Security - Top 10," 2019. [Online]. Available: https://owasp.org/www-project-api-security/. [Accessed 19 April 2022].

[34] N. Madden, API Security in Action, Shelter Island: Manning Publications, 2020.

[35] The MITRE Corporation, "CWE-200: Exposure of Sensitive Information to an Unauthorized Actor," 19 July 2006. [Online]. Available: https://cwe.mitre.org/data/definitions/200.html. [Accessed 19 April 2022].

[36] The MITRE Corporation, "CWE-213: Exposure of Sensitive Information Due to Incompatible Policies," 19 July 2006. [Online]. Available: https://cwe.mitre.org/data/definitions/213.html. [Accessed 19 April 2022].

[37] The MITRE Corporation, "CWE-209: Generation of Error Message Containing Sensitive Information," 19 July 2006. [Online]. Available: https://cwe.mitre.org/data/definitions/209.html. [Accessed 19 April 2022].

[38] The OWASP Foundation, "Injection Flaws," [Online]. Available: https://owasp.org/www-community/Injection_Flaws. [Accessed 28 January 2022].

[39] G. Deepa and P. S. Thilagam, "Securing web applications from injection and logic vulnerabilities: Approaches and challenges," *Information and software technology,* vol. 74, pp. 160-180, 2016.

[40] M. A. M. Yunus, M. Z. Brohan, N. M. Nawi, E. S. M. Surin, N. A. M. Najib and C. W. Liang, "Review of SQL Injection : Problems and Prevention," *INTERNATIONAL JOURNAL ON INFORMATICS VISUALIZATION,* vol. 2, no. 3, pp. 215-219, 2018.

[41] The OWASP Foundation, "Mass Assignment Cheat Sheet," [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Mass_Assignment_Cheat_Sheet.html. [Accessed 2 February 2022].

[42] The MITRE Corporation, "CWE-915: Improperly Controlled Modification of Dynamically-Determined Object Attributes," 26 January 2013. [Online]. Available: https://cwe.mitre.org/data/definitions/915.html. [Accessed 19 April 2022].

[43] M. Isbitski, API Security for Dummies, Hoboken: John Wiley & Sons, Inc., 2022.

[44] BeyondTrust, "Systems Hardening," [Online]. Available: https://www.beyondtrust.com/resources/glossary/systems-hardening. [Accessed 1 February 2022].

[45] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," August 2018. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc8446. [Accessed 1 February 2022].

[46] A. Prodromou, "TLS Security 6: Examples of TLS Vulnerabilities and Attacks," 31 March 2019. [Online]. Available: https://www.acunetix.com/blog/articles/tls-vulnerabilities-attacks-final-part/. [Accessed 1 February 2022].

[47] D. Petty, "The Not-So-Same-Origin Policy," July 2017. [Online]. Available: https://www.ise.io/wp-content/uploads/2018/03/ise_same-origin-policy_whitepaper.pdf. [Accessed 1 February 2022].

[48] Mozilla Foundation, "Web Security," [Online]. Available: https://infosec.mozilla.org/guidelines/web_security. [Accessed 22 January 2022].

[49] A. Lamba, "API Design Principles & Security Best Practices," *Cybernomics,* vol. 1, no. 3, pp. 21-25, 2019.

[50] The OWASP Foundation, "OWASP Application Security Verification Standard," [Online]. Available: https://owasp.org/www-project-application-security-verification-standard/. [Accessed 1 February 2022].

[51] The MITRE Corporation, "Common Vulnerabilities and Exposures," [Online]. Available: https://cve.mitre.org/. [Accessed 1 February 2022].

[52] V. Li, "API Security 101: Security Misconfiguration," 17 August 2021. [Online]. Available: https://blog.shiftleft.io/api-security-101-security-misconfiguration-eb9efed80ebe. [Accessed 1 February 2022].

[53] The OWASP Foundation, "OWASP Web Security Testing Guide," [Online]. Available: https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/02-Configuration_and_Deployment_Management_Testing/06-Test_HTTP_Methods. [Accessed 1 February 2022].

[54] Qualys, ”Qualys SSL Labs,” [Online]. Available: https://www.ssllabs.com/. [Haettu 19 April 2022].

[55] Probely, "Security Headers," [Online]. Available: https://securityheaders.com/. [Accessed 22 January 2022].

[56] K. Kent and M. Souppaya, Guide to Computer Security Log Management, National Institute of Standards and Technology, 2006.

[57] The CEE Editorial Board, "Common Event Expression," May 2010. [Online]. Available: https://cee.mitre.org/docs/CEE_Architecture_Overview-v0.5.pdf. [Accessed 19 April 2022].

[58] The OWASP Foundation, "Logging Cheat Sheet," [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Logging_Cheat_Sheet.html. [Accessed 21 January 2022].

[59] A. Chuvakin and G. Peterson, "How to Do Application Logging Right," *IEEE Security & Privacy,* vol. 8, no. 4, pp. 82-85, July 2010.

[60] A. A. Chuvakin, K. J. Schmidt and C. Phillips, Logging and Log Management, Saint Louis: Elsevier Science & Technology Books, 2012.

[61] The MITRE Corporation, "CWE-778: Insufficient Logging," 2 July 2009. [Online]. Available: https://cwe.mitre.org/data/definitions/778.html. [Accessed 19 April 2022].

[62] The MITRE Corporation, "CWE-223: Omission of Security-relevant Information," 19 July 2006. [Online]. Available: https://cwe.mitre.org/data/definitions/223.html. [Accessed 19 April 2022].

[63] C. Poole and R. Prouse, "NUnit," [Online]. Available: https://nunit.org/. [Accessed 14 April 2022].