ANÁLISE DE VULNERABILIDADES EM API

Silva, Allan D.; Viana, José A. A.

e-mail:
allan.silva62@fatec.sp.gov.br; jose.viana@fatec.sp.gov.br

Resumo: Empresas frequentemente usam aplicações que consomem APIs para comunicação e transmissão de dados. Programar uma API requer um alto nível de segurança, mas muitos desenvolvedores não consideram isso. Nesta pesquisa, será explorada uma falha lógica em uma API dentro de um laboratório de um *marketplace* com ferramentas *open-source*.

Palavras-chave: APIs. *Digital Security. Logical Flaw. API Development*.

Abstract: *Companies often use applications that consume APIs for communication and data transmission. Implementing an API requires a high level of security, but many developers do not consider this. In this research, a logical flaw in a marketplace laboratory will be explored using open-source tools.*

Keywords: *APIs. Digital Security. Logical Flaw. API Development*

1. Introdução

Uma API (*Application Programming Interfaces*) é um conjunto de métodos e padrões de comunicação que permitem que diferentes softwares se comuniquem entre si. Em outras palavras, é um canal de comunicação que permite que diferentes sistemas interajam uns com os outros. Essas interfaces podem ser usadas para muitas finalidades, desde compartilhar informações e funcionalidades entre aplicações, até permitir a integração de sistemas e serviços distintos. Elas são essenciais no desenvolvimento de softwares e aplicações modernas, pois permitem uma maior interconexão entre diferentes sistemas e serviços.

A utilização de uma API para a comunicação de aplicações modernas é bem comum, seja em uma aplicação pequena ou grande. Devido ao volume da utilização dessas interfaces, muitos vetores de ataques cibernéticos podem surgir através dessas portas de entrada, que muitas vezes não são realizados os testes de invasão para testar a segurança.

Uma vulnerabilidade pode resultar em uma série de problemas, desde o vazamento de informações confidenciais até o comprometimento de sistemas inteiros. Em que pode causar prejuízos financeiros, além de afetar a privacidade e a segurança dos usuários desses sistemas. Por isso, é fundamental que sejam adotadas medidas de segurança adequadas para mitigar esses riscos. Para os desenvolvedores de software, conhecer sobre a segurança de APIs é um aspecto fundamental e crítico no desenvolvimento de software. Ao entender tais aspectos, os desenvolvedores podem criar APIs mais seguras e robustas, garantindo a proteção de dados e informações sensíveis contra ameaças de segurança. Lembrando que, nenhum sistema é totalmente seguro, porém muitas brechas podem ser resolvidas com boas técnicas de programação.

2. Justificativa

Existem vários estudos de empresas renomadas que apontam para a falta de testes de segurança em APIs por parte dos desenvolvedores. Um exemplo é o relatório da empresa de segurança *Akamai Technologies*, que revelou que 83% das empresas não testam regularmente a segurança de suas APIs. Isso significa que a maioria das empresas não está tomando as medidas necessárias para proteger seus sistemas e dados contra possíveis ameaças de segurança.

Neste estudo é destacado a importância do teste de segurança de APIs e a necessidade de que empresas implementem medidas de segurança adequadas para proteger seus sistemas. Desta forma a implementação de um guia para mitigação de possíveis vulnerabilidades em APIs podem ajudar a desenvolvedores inexperientes a desenvolver aplicações mais seguras olhando para um aspecto lógico e não técnico.

3. Objetivo

O objetivo do trabalho é mitigar vulnerabilidades em APIs para garantir a segurança de sistemas que dependem do consumo dessas interfaces. Além disso, o trabalho visa fornecer técnicas para análise de vulnerabilidades para o desenvolvimento de soluções mais seguras e ajudar a prevenir ataques cibernéticos que possam resultar em vazamento de informações confidenciais, violação de privacidade, perda financeira e danos à reputação da empresa. O trabalho também busca fornecer conhecimentos e descobertas na área de segurança de APIs.

- Vetores de ataque em uma API;

- Enumeração de *Endpoints* vulneráveis;
- Demonstração do ataque em um *gateway* de pagamento de uma API;
- Ferramenta *Burp Suite.*

4. Fundamentação Teórica

Agora é representado toda a fundamentação teórica do funcionamento de uma API e as ferramentas utilizadas neste trabalho.

4.1. Estrutura de uma API: Requisição, Resposta e *Endpoint*

Entender a estrutura de uma API é essencial antes de realizar uma análise de segurança, nesta etapa é demonstrado a estrutura básica do ciclo de comunicação de uma API e a estrutura de *endpoints* que são utilizados no consumo das interfaces.

4.1.1. Requisição

Uma requisição é uma ação do cliente que solicita informações ou executa uma operação em um servidor. As requisições podem incluir várias informações, como o método HTTP utilizado, cabeçalho da requisição e os parâmetros de consulta ou o corpo da mensagem *(INTERNATIONAL BUSINESS MACHINES, 2023).*

As requisições são fundamentais porque permitem que os clientes acessem e manipulem recursos em um servidor. Sem requisições, a comunicação entre aplicativos seria muito limitada, e muitas funcionalidades importantes da *web* e das APIs não seriam possíveis. Além disso, as requisições podem incluir informações adicionais, como um *token* de autenticação, que permite que o servidor identifique o cliente e autorize o acesso aos recursos.

4.1.2. Resposta

A resposta é a mensagem enviada pelo servidor em resposta a uma solicitação de uma requisição. Essa mensagem pode conter os dados solicitados pelo cliente ou o resultado de uma operação executada no servidor. As respostas são importantes em uma API *RESTful* porque permitem que os clientes obtenham os dados, informações solicitadas ou saibam se uma operação foi bem-sucedida ou não. As respostas geralmente incluem um código de status HTTP, que indica se a solicitação foi bem-sucedida ou não, juntamente com os dados ou informações relevantes *(INTERNATIONAL BUSINESS MACHINES, 2023).*

4.2. *Endpoint*

De acordo com a *Cloudflare (2023),* um *endpoint* em uma API é um ponto de acesso específico para um recurso ou um conjunto de recursos em um servidor *web*. Em outras palavras, é uma URL (*Uniform Resource Locator*) que identifica um recurso específico na API e permite que os clientes acessem ou modifiquem esse recurso.

Por exemplo, em uma API de um sistema de e-commerce, pode haver um *endpoint* "/produtos" que permita que os clientes recuperem a lista de todos os produtos disponíveis na loja, um *endpoint* "/produtos/123" que permita que os clientes obtenham detalhes sobre um produto específico com *id* 123, e um *endpoint* "/produtos/123/comentarios" que permita que os clientes recuperem os comentários deixados por outros usuários sobre o produto com ID 123. Os *endpoints* geralmente são associados a um método HTTP específico, como *GET, POST, PUT ou DELETE*, que determina qual operação deve ser realizada no recurso identificado pelo *endpoint (MOZILLA, 2023).*

4.2.1. Protocolos de Comunicação HTTP e HTTPS

Os protocolos HTTP (*Hypertext Transfer Protocol*) e o HTTPS (*Hypertext Transfer Protocol Secure*) são usados para transferir dados pela internet. O HTTP é um protocolo cliente-servidor, onde o cliente envia uma solicitação para o servidor, que responde com uma resposta. Este protocolo utiliza métodos, como *GET, POST, PUT, DELETE*, para definir qual ação deve ser executada em um *endpoint.* Também é utilizado códigos de status para indicar o sucesso ou falha da solicitação, e cabeçalhos para fornecer informações adicionais *(CLOUDFLARE, 2023).*

HTTPS é uma evolução do HTTP que utiliza uma camada de criptografia SSL/TLS para proteger a transferência de dados entre o cliente e o servidor. Isso garante a privacidade e a integridade dos dados, impedindo que terceiros interceptem ou modifiquem as informações que estão sendo transmitidas *(CLOUDFLARE, 2023).*

4.3. Explorando o protocolo HTTP

Nesta seção é explorado o método HTTP e em seus métodos de comunicação e o seu cabeçalho entre as requisições.

4.3.1. Cabeçalho HTTP

O cabeçalho HTTP é uma parte fundamental das requisições entre cliente e servidor. Um cabeçalho contém informações adicionais a mensagem HTTP, como o tipo de conteúdo, o tamanho da mensagem, o método de autenticação, e os valores a serem entregues ao servidor ou cliente. Existem dois tipos de cabeçalhos HTTP: cabeçalhos de solicitação e cabeçalhos de resposta. Os cabeçalhos de solicitação são enviados pelos clientes para solicitar um recurso do servidor, enquanto os cabeçalhos de resposta são enviados pelos servidores para enviar informações de volta para o cliente.

Todos os dados que são comunicados entre o cliente e o servidor podem ser interceptados e alterados antes de chegar ao seu destino. O cabeçalho de uma requisição poderá ser alterado utilizando algum *proxy* de comunicação dentro da máquina do proprietário. Alterar um cabeçalho HTTP ajuda o atacante a descobrir novos parâmetros utilizados pelo servidor, ou mesmo manipulando a resposta para ver como ela funciona e mitigar algum ataque sobre aquela aplicação.

Um cabeçalho possui a estrutura de chave-valor, onde a chave serve de identificação para o local de destino e o valor é o item a ser utilizado para ser aplicado em alguma lógica.

4.3.2. Métodos HTTP

Os métodos HTTP são verbos que indicam qual ação que deve ser executada em um *endpoint* de uma requisição. Os principais métodos HTTP são:

- *GET*: solicita uma representação do recurso especificado no URL. É usado para obter informações do servidor;
- *POST*: envia dados para o servidor para que sejam processados. É usado para criar recursos ou realizar ações que alterem o estado do servidor;
- *PUT*: atualiza um recurso existente no servidor. É usado para substituir completamente o conteúdo do recurso com o conteúdo enviado na requisição;
- *DELETE*: remove o recurso especificado no URL. É usado para excluir um recurso do servidor;
- *HEAD*: é semelhante ao método GET, mas retorna apenas os cabeçalhos da resposta, sem o corpo da resposta;

- *OPTIONS*: solicita as opções de comunicação disponíveis para o recurso especificado. É usado para descobrir quais métodos e cabeçalhos são suportados pelo servidor para um determinado recurso.

Esses métodos são usados em conjunto com outros elementos de uma requisição HTTP, como o *URL*, o cabeçalho da requisição e o corpo da mensagem, para permitir a comunicação entre clientes e servidores em aplicações web *(MOZILLA, 2023)*.

4.3.3. Estrutura de dados transmitidos por uma API

A estrutura de dados transmitidos por uma API pode variar de acordo com o formato escolhido para a transmissão, como JSON ou XML, mas geralmente segue um padrão definido pela arquitetura REST.

JSON (*JavaScript Object Notation*) é um formato de dados bastante utilizado em aplicações web para transferência de informações entre o cliente e o servidor. É um formato de dados estruturado em pares de chave-valor, onde as chaves são *strings* que representam os nomes dos campos e os valores podem ser de diversos tipos, como *strings*, números, objetos, *arrays*, entre outros *(W3SCHOOLS, 2022)*.

XML (*Extensible Markup Language*) é uma linguagem de marcação que permite estruturar dados em um formato de *tags* que definem elementos e atributos, e que dão informações adicionais sobre esses elementos. No entanto, o XML é considerado uma linguagem mais verbosa e complexa quando comparada ao JSON, o que pode tornar o seu processamento mais lento e demandar mais recursos do sistema. Além disso, a sua estrutura mais complexa também pode dificultar a sua leitura e manutenção *(W3SCHOOLS, 2022)*.

4.4. Ferramentas de Exploração

Nesta subseção é representado as ferramentas utilizadas para exploração de uma API. Parte das ferramentas são *open-source* e podem ser encontradas nos repositórios dentro do *GitHub*.

4.4.1. *Burp Suite*

Está *é* uma ferramenta de teste de segurança para aplicações *web*, utilizada por profissionais de segurança cibernética. Ela é composta por várias ferramentas que trabalham juntas para executar testes de penetração em aplicativos *web*. Algumas das funcionalidades do

Burp Suite incluem interceptação de solicitações HTTP, teste de vulnerabilidades automatizados, manipulação de dados nas requisições, análise de desempenho, entre outras.

Além disso, o *Burp Suite* é altamente personalizável, permitindo que os usuários desenvolvam suas próprias extensões. Isso torna a ferramenta adequada para analistas de segurança da informação integrar com outras ferramentas e ampliar os modos de testes.

4.4.2. Ffuf

O Ffuf é uma ferramenta *open-source* usada para a descoberta de diretórios e parametros ocultos em uma aplicação web. Essa ferramenta pode ser usada tanto para testes de segurança quanto para fins de exploração. A ferramenta é executada na linha de comando e suporta várias configurações, incluindo a configuração para especificar o tipo de palavra-chave para pesquisa, o número máximo de conexões simultâneas, o tempo limite de solicitação, entre outros *(KALI, 2023)*.

Esta ferramenta é capaz de detectar diretórios e arquivos que normalmente não são visíveis em uma aplicação web, como aqueles que não foram indexados pelo mecanismo de busca do site. Ele também pode ser usado para descobrir *endpoints* ocultos em uma API, o que é particularmente útil para testes de segurança em aplicações web baseadas em API.

4.4.3. Wordlist

De acordo com o website *IMaster (2023)* uma *wordlist* é uma lista de palavras ou combinações de caracteres usadas para fazer ataques de força bruta. Ela é usada para criar possíveis valores de entrada que podem ser enviados para a API, com o objetivo de descobrir senhas, chaves de API, nomes de usuários, entre outros dados confidenciais.

A *wordlist* pode ser criada manualmente, por exemplo, por meio da combinação de palavras comuns ou por meio de dicionários de senhas conhecidas. Existem também várias *wordlists* disponíveis gratuitamente na internet, como a *SecList*, que contém uma grande variedade de palavras e combinações de caracteres comumente usados em senhas e chaves de API. Cada *wordlist* pode ser usada em conjunto com ferramentas de teste de segurança, como o *Burp Suite* e o Ffuf, para automatizar ataques de *brute force*.

5. Trabalhos Similares

Este trabalho tem como base alguns trabalhos similares já realizados por outros estudantes, com isso foram escolhidos três artigos análogos a vulnerabilidades de APIs.

O trabalho realizado por Sampaio (2021) aborda todas as 10 vulnerabilidades de aplicações *web* colocadas pela OWASP (2017) de forma prática. O autor estrutura o trabalho em 3 fases sendo elas: demonstração, correção & mitigação. Além de demonstra boas práticas de desenvolvimento para evitar tais falhas nas aplicações.

Em relação ao desenvolvimento seguro e boas práticas o trabalho de Carneiro Neto (2020) foi essencial, pois é abordado todo o ciclo de desenvolvimento de uma API *Rest*. É explorado todo o conceito do protocolo HTTP e é demonstrado um ciclo de desenvolvimento seguro de uma *Rest* API. Neste trabalho é abortado toda a teoria de uma API, no entanto, mais profundamente e voltado para a mitigação de vulnerabilidades lógicas.

Por último ficou o trabalho de Reis Silva (2022) que é representado uma linha de desenvolvimento seguro em sistemas *web* modernos. O trabalho demonstra como uso de linguagens de programação pode influenciar na segurança das aplicações e como é abordado a segurança para métodos de autenticação do usuário final, além de trazer o uso da computação em nuvem. Diferente dos outros trabalhos, este é mais teórico sem abordagens técnicas, mas é importante para concretização dos conceitos de segurança de modo geral.

Os trabalhos mencionados trouxeram contribuições importantes para este artigo, pois contribuem nos conceitos fundamentais para o desenvolvimento do trabalho. Utilizá-los como base foi fundamento para elaboração de um ciclo de desenvolvimento e conhecimento de novos termos técnicos. Assim unindo-se os elementos em comum abordados pelos autores lidos, formou-se uma base de como desenvolver a construção do laboratório presente neste trabalho de graduação.

6. Metodologia

Este trabalho usou como metodologia de pesquisa o estudo bibliográfico, focado na pesquisa de conceitos através de *websites* e trabalhos similares. Nesta seção de metodologia é demonstrado todo o ciclo de desenvolvimento. Diferente de um *pentest* real, durante a fundamentação não será seguido todos os passos recomendados e será uma exploração mais direta, como geralmente é realizado por profissionais especializados.

Para a realização da pesquisa é utilizado um laboratório que representa um *e-commerce* contendo uma única falha de segurança lógica no método de pagamento de sua API. O laboratório foi desenvolvido com *NextJS* utilizando também o *gateway* de pagamento da *Stripe,* e pode ser baixado no GitHub para a realização de testes locais. Ter um laboratório com esse nível de simulação é algo essencial para os profissionais se familiarizarem com as técnicas necessárias.

Como segunda etapa da fundamentação teórica, será focado na fase de reconhecimento que é uma das mais importantes e intensas a ser seguida, seja em um *pentest* real ou em laboratórios simulados, pois é nela que se encontra todos os pontos fracos de uma aplicação. Será utilizado a combinação de diversas ferramentas de reconhecimento para demonstrar como um profissional de segurança da informação deve realizar a fase de reconhecimento de uma API para posteriormente encontrar os pontos vulneráveis da aplicação. O intuito principal desta fase é encontrar possíveis parâmetros, url ou páginas vulneráveis.

Por fim a fase de exploração será a última. Esta etapa acontece quando é finalizado a fase de reconhecimento e já possui alguns pontos da aplicação que tem parâmetros com possibilidade de serem atacados. Está é a parte em que será necessário realizar as combinações das ferramentas citadas como fundamentação teoria, tais essas ferramentas como o *BurpSuite*, Cliente HTTP e o *Ffuf*.

7. Desenvolvimento

Nesta seção, será abordado uma análise prática da falha de segurança lógica proposta neste trabalho. Toda a utilização de ferramentas e conceitos citados no trabalho são colocados em prática para habituar o leitor com os conceitos básicos de mitigar falhas de segurança em uma aplicação.

7.1. Laboratório

Antes de explorar a falha será demonstrado como realizar a instalação do laboratório para que seja testado localmente. Para começar a instalação é necessário ter o *NodeJS* instalado na máquina e o *Git* para realizar o download do projeto através do *GitHub*. Depois de certificar que os requisitos estão sendo atendidos é hora de iniciar. Seguindo como primeiro processo de configuração, o leitor deverá realizar o clone do repositório (Figura 1).

Figura 1: *Clonando o laboratório*

```
PS C:\Users\55179\Desktop> git clone https://github.com/allandiegoasilva/cyber-store.git
```

Fonte: *Autoria própria.*

Para continuar a instalação do projeto é necessário realizar a instalação de todas as dependências do laboratório. As dependências são necessárias para baixar as bibliotecas que está aplicação utiliza, tais dependências servem para diversas finalidades como uma aplicação de estilos ou mesmo para comunicação HTTP de outros clientes. Para realizar a instalação acesse a pasta criada e execute o comando representado (Figura 2).

Figura 2: *Acessando a pasta do projeto e instalando as dependências*

```
PS C:\Users\55179\Desktop> cd .\cyber-store\
PS C:\Users\55179\Desktop\cyber-store> npm install --force
```

Fonte: *Autoria própria.*

Depois que as dependências são finalizadas o projeto poderá ser iniciado (Figura 3). Ao iniciar o projeto, é criado um cliente HTTP para ser consumido de uma aplicação web e poderá ser acessado através de um navegador. Toda a parte de backend desta aplicação ficará ao lado do servidor que é tratado automaticamente pelo framework *NextJS*.

Figura 3: *Inicializando o laboratório no ambiente local*

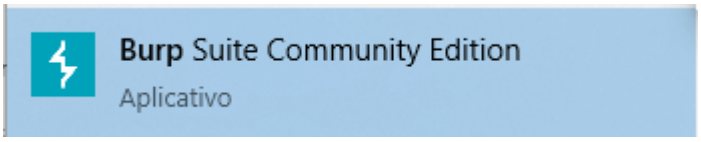
```
PS C:\Users\55179\Desktop\cyber-store> npm run dev

> gift-store@0.1.0 dev
> next dev

- ready started server on 0.0.0.0:3000, url: http://localhost:3000
```

Fonte: *Autoria própria.*

7.2. Fase de Reconhecimento

Como citado anteriormente a fase de reconhecimento é a mais demorada e cansativa, já que é nela que precisa do máximo de atenção nos mínimos detalhes. Então para começar o processo de reconhecimento é necessário ter o *BurpSuite* instalado na máquina local e pronto para uso, após isso basta inicializá-lo (Figura 4).

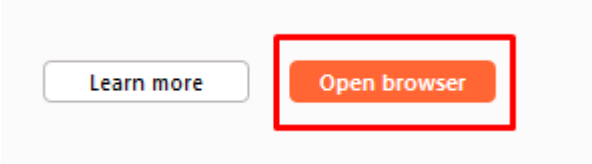
Figura 4: *Inicializando o BurpSuite*



Fonte*: Autoria própria.*

Depois de inicializar o programa, basta acessar o *browser* que já vem embutido (Figura 5) no programa. Este browser irá servir como *proxy* enquanto é realizado a navegação pelo site, com as funcionalidades do *Burp* será possível identificar *endpoints* ocultos ou parâmetros descontinuados que ainda são usados pela aplicação e que não foram removidos do código.

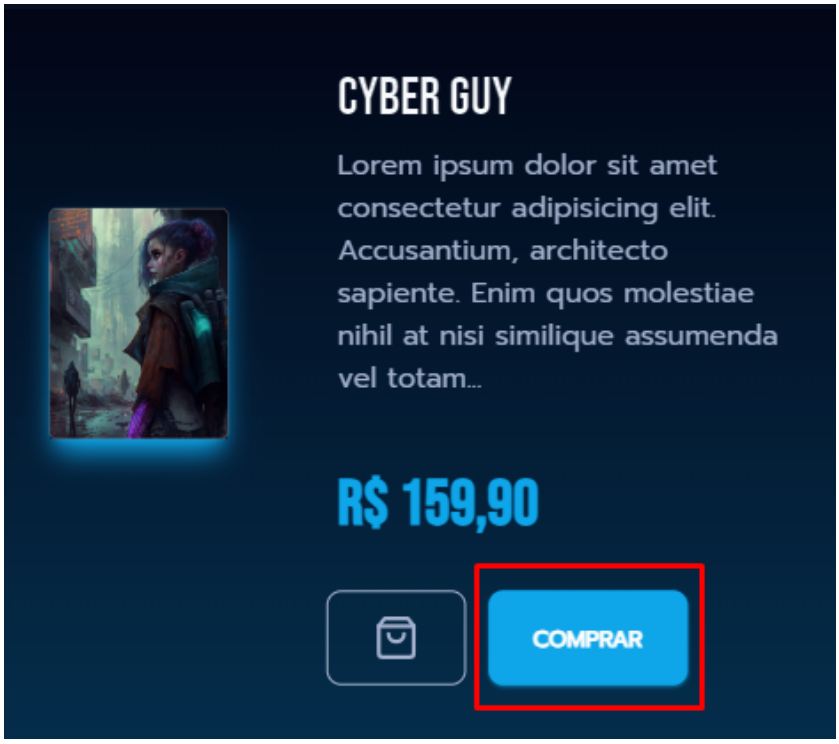
Figura 5*: Inicializando o navegador do BurpSuite*



Fonte: *Autoria própria.*

Ao abrir o navegador, o processo de navegação e reconhecimento pode ser iniciado. Abra o site local inicializado através do endereço “http://localhost:3000” isso irá permitir visualizar o laboratório. Depois que o laboratório foi acessado através do navegador do *BurpSuite*, agora é necessário clicar em alguns botões da aplicação para entender o processo de requisições do site. Para realizar esse processo de interação será realizado um clique no botão “comprar” de qualquer produto (Figura 6).

Figura 6: *Clicando no botão "comprar"*



Fonte: *Autoria própria.*

Quando é clicando para efetuar uma compra é realizado uma requisição *POST* para o servidor (Figura 7), essa requisição pode ser analisada ao acessar a aba "*target*" do *BurpSuite*, nesta aba contém todo o histórico de navegação pelo website.

Figura 7: *Analisando o histórico do BurpSuite*

http://localhost:3000	POST	/api/checkout	✓	200	572	JSON

Fonte: *Autoria própria.*

Agora para entender os dados que são enviados e retornados do serviço de API, basta acessar a requisição com a ferramenta dando dois cliques sobra a requisição. Ao acessar esta requisição é notável que os dados estão sendo enviados para o servidor em um formato *JSON* (Figura 8). Provavelmente o *web site* está fazendo essa requisição para processar a compra, neste processo é comum haver uma comunicação com o *gateway* de pagamento diretamente, caso não haja é retornando um link com a página de pagamento do próprio *gateway*. Muitas empresas fazem isso, pois torna mais rápido integrar várias formas de pagamento pelo site, ou seja, contrata uma empresa externa para realizar o trabalho mais complexo.

Figura 8: *Payload do endpoint de checkout*

```
POST /api/checkout HTTP/1.1
Host: localhost:3000
Content-Length: 316
sec-ch-ua:
Accept: application/json, text/plain, */*
Content-Type: application/json
sec-ch-ua-mobile: ?0
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/114.0.5735.134 Safari/537.36
sec-ch-ua-platform: ""
Origin: http://localhost:3000
Referer: http://localhost:3000/

{
  "products":[
    {
      "title":"Cyber Guy",
      "description":"Descrição do cyber Gy",
      "price":159.9,
      "discount":0,
      "color":"pink",
      "image":"/img/image-1.png",
      "type":"guy",
      "colors":{
        "linerBg":"to-sky-500/25",
        "shadow":"shadow-sky-500",
        "btnBgColor":"bg-sky-500",
        "btnShadowColor":"shadow-sky-500/50",
        "textPriceColor":"text-sky-500"
      }
    }
  ]
}
```

Fonte: *Autoria própria.*

Depois que o servidor processa estes parâmetros é retornado outro *payload (*Figura 9*)*. O formato da resposta contém o link de pagamento que leva o usuário até o *geteway* da *Stripe*.

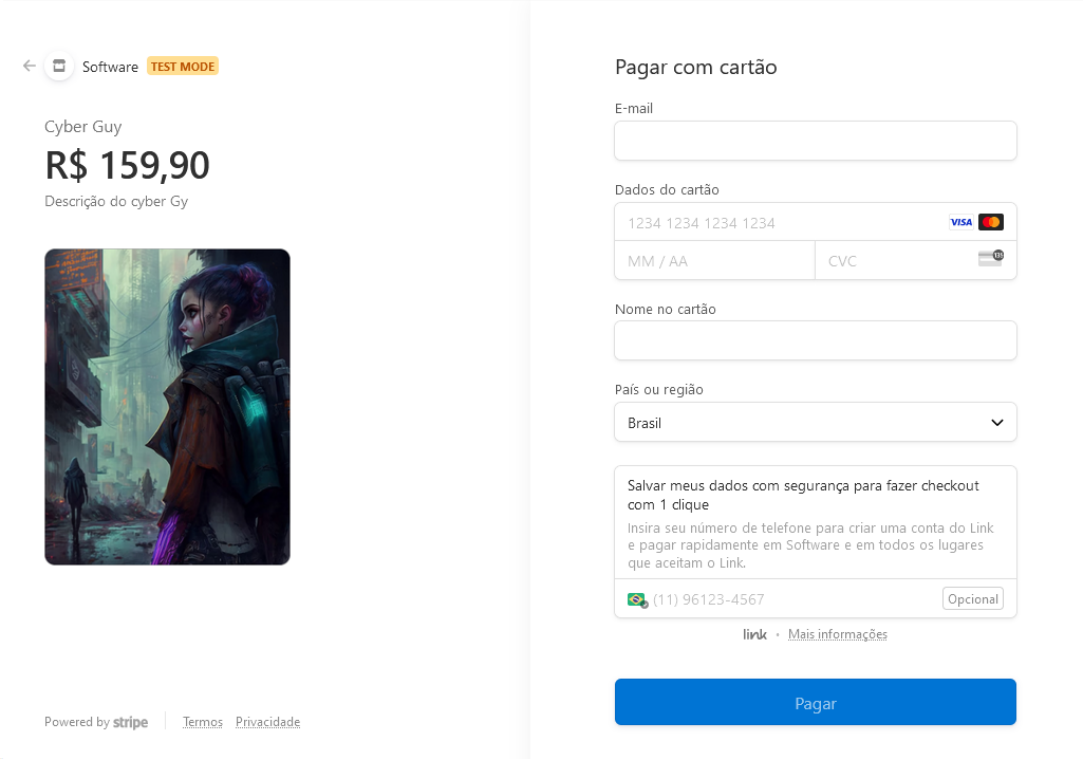
Figura 9: *Resposta do endpoint de checkout*


```
HTTP/1.1 200 OK
content-type: application/json; charset=utf-8
etag: "hp90wfhbnnao"
vary: Accept-Encoding
date: Fri, 23 Jun 2023 04:10:12 GMT
connection: close
Content-Length: 384

{
  "success":true,
  "uri":
  "https://checkout.stripe.com/c/pay/cs_test_a14M6pbkUykHFDqyR5gHRuBVUwUE
  gGlJb5LgnACsNzzDvlpVT2DSMjNplb#fidkdWx0YHwnPydlblpxYHZxWjA0SzEydX9DcVBB
  fOhgPXI8cTJOSX9kTG9oQXJXYlFlYTNhM39jMlVONOtgNUdDbmxoQlx8SkFBdm9TZlR8fEh
  IRlx0TUFATkRtaWJHMmFmRGZBcHdUMGAxNTV0U2FxajM0YicpJ2N3amhWYHdzYHcnP3F3cG
  ApJ2lkfGpwcVF8dWAnPyd2bGtiaWBabHFgaCcpJ2BrZGdpYFVpZGZgbWppYWB3dic%2FcXd
  wYHgl"
}
```


Fonte: Autoria própria.

Com o link anterior retornado o site redireciona o usuário para outra página, está página é a de checkout (Figura 10) e possuí as informações necessárias para pagamento e do produto.

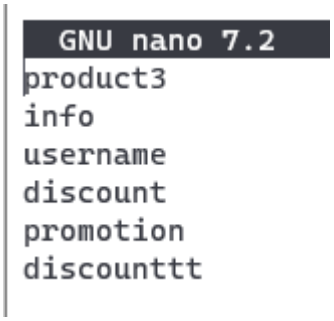
Figura 10: *Página de checkout*



Fonte: *Autoria própria.*

Até o presente momento do reconhecimento foi identificado um *endpoint* de compra que permite o usuário comprar um item dentro da loja. No entanto olhando este funcionamento aparentemente está tudo correto, mas como um analista de segurança tem que pensar fora da caixa então terá que tentar encontrar parâmetros vulneráveis dentro da aplicação. Para realizar está análise primeiramente deve criar a *wordlist* dos possíveis parâmetros vulneráveis dentro do *payload* de *checkout (*Figura 11*)*.

Figura 11: Criação da wordlist



Fonte: *Autoria própria.*

Na *wordlist* anterior foi utilizada apenas seis palavras, mas em um ambiente real é utilizado milhares e até centenas de combinações. Agora será necessária uma combinação da ferramenta Ffuf com a *wordlist* criada para descobrir se não há nenhum parâmetro oculto no *payload* de pagamento. Para isso será enviado o mesmo *payload* através do método *POST* e filtrando apenas resultados 401 de resposta do servidor, com isso é inserido um parâmetro "*FUZZ*" na chave do *payload* com o valor, "valor", a ideia desta manobra é encontrar qualquer outro tipo de parâmetro. Então o execute.

Figura 12: *Executando o Ffuf com a wordlist*

```
┌──(kali㉿kali)-[~/Documents/nft-store]
└─$ ffuf -w wordlist.txt -u https://cyber-store.allandiego.com.br/api/checkout -X POST -H "Co
ntent-Type: application/json" -d '{"FUZZ":"valor", "products":[{"title":"Cyber Robot","descri
ption":"Descrição do cyber Robot","price":324.9,"discount":0,"color":"purple","image":"/img/i
mage-2.png","type":"robot","colors":{"linerBg":"to-purple-500/25","shadow":"shadow-purple-500
","btnBgColor":"bg-purple-500","btnShadowColor":"shadow-purple-500/50","textPriceColor":"text
-purple-500"}}]}' -mc 401
```

Fonte: *Autoria própria.*

O Fuff agora retorna código de status filtrado e que um parâmetro *discount* dentro do *payload* foi encontrado (Figura 13), sabe-se que o parâmetro foi encontrado pela linha "*FUZZ*". Este parâmetro agora pode ser explorado através de um cliente HTTP.

Figura 13: *Saída do Ffuf*

```
[Status: 401, Size: 72, Words: 7, Lines: 1, Duration: 1916ms]
    * FUZZ: discount

:: Progress: [6/6] :: Job [1/1] :: 3 req/sec :: Duration: [0:00:03] :: Errors: 0 ::
```

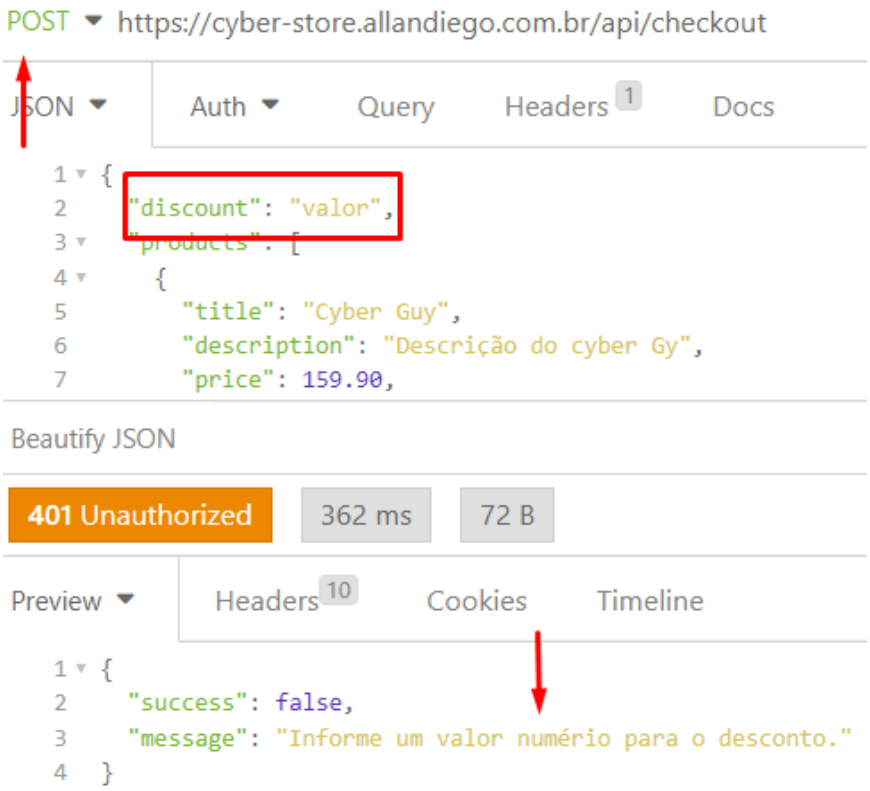
Fonte*: Autoria própria.*

A partir de agora, foi encontrado um parâmetro que provavelmente já foi utilizado dentro da aplicação em algum momento do ciclo de vida dela. Só de analisar o nome dele imagina se que é relacionado ao valor do produto. Então, nesta parte do reconhecimento já tem uma base que será explorada na próxima seção do trabalho e com isso finaliza a fase de reconhecimento.

7.3. Explorando a vulnerabilidade

Nesta seção é explorado o parâmetro de *discount* encontrado no processo de reconhecimento. Antes de começar a realizar esta análise é necessário ter um cliente HTTP, neste caso será utilizado o *insominia* para fazer as requisições para o *endpoint* de pagamento. Para começar abra o seu cliente HTTP, coloque o método *POST* e o parâmetro encontrado com o valor que deu errado no Ffuf, e faça a requisição (Figura 14).

Figura 14: *Parâmetro discount com o valor inválido*



Fonte: *Autoria própria.*

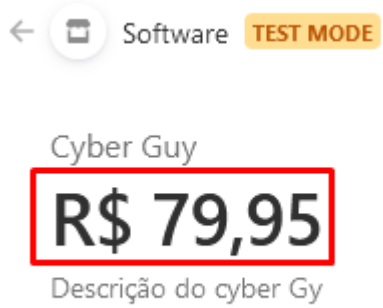
Note que o parâmetro requisita um valor numérico para ser enviado no *JSON*. Com isso é necessário ajustar o valor e ver como o *endpoint* vai se comportar. Ao alterar o parâmetro para o valor válido, o servidor irá retornar novamente a *URL* de da página de *checkout*. Perceba também o valor atual do produto que é de R$ 159,00 (Figura 15).

Figura 15: *Parâmetro discount com valor válido*

Fonte: *Autoria própria.*

Após trocar o valor do parâmetro descoberto e pegar o *link* da página de *checkout*, é ideal acessar a página de pagamento e ver o que pode ter impactado no produto quando este parâmetro é informado. Neste caso ao ver o resultado nota-se que o valor foi alterado para 50% do original (Figura 16).

Figura 16: *Valor do produto alterado pelo parâmetro discount*



Fonte: *Autoria própria.*

Para alguns desenvolvedores notar este tipo de falha é um desafio, mas o problema está no parâmetro que foi descontinuado da aplicação e, no entanto, continuou lá. Então com essa análise fica claro que é sempre bom revalidar as funcionalidades principais do sistema e rever lógicas e parâmetros que foram descontinuados em algum momento da aplicação. Está falha encontrada reflete diretamente nos preços dos produtos, caso estivesse on-line, os atacantes poderiam comprar produtos de graça, principalmente se os atendimentos forem automáticos e não passam por revisão humana.

Encontrar uma falha assim dentro de uma aplicação é perceptível que a equipe de desenvolvedores não entenda está parte de segurança e acabam prejudicando a empresa indiretamente.

8. Resultados e Discussões

Em resumo, este trabalho teve como objetivo analisar uma vulnerabilidade em uma API, com destaque para um laboratório que inclui uma falha lógica. Durante a pesquisa, foi identificado uma falha que permite o atacante causar dano financeiro para a empresa. Sendo assim, ficou evidente que as APIs estão sujeitas a uma variedade de riscos, incluindo vazamento de informações confidenciais e comprometimento de sistemas. Então através do laboratório foi

possível ilustrar como uma vulnerabilidade pode ser explorada e quais os possíveis impactos para a segurança e integridade dos sistemas.

Com base nos resultados obtidos, é fundamental enfatizar a importância de adotar medidas de segurança adequadas ao desenvolver e utilizar APIs. Isso inclui a implementação de práticas robustas, bem como a realização de testes de segurança e análise de vulnerabilidades de forma regular. Além disso, é crucial que os desenvolvedores e profissionais de segurança estejam cientes das vulnerabilidades comuns que podem afetar as APIs e que estejam atualizados sobre as melhores práticas e soluções de segurança disponíveis.

9. Conclusões

Por fim, conclui-se com este trabalho que um programador sem experiência com segurança da informação consiga realizar testes de segurança em uma API. O trabalho contribuí diretamente para auxiliar profissionais menos experientes em segurança da informação a testar suas aplicações de modo que não fiquem vulneráveis. E lembra a importância da revisão da lógica de uma API, pois as falhas nem sempre são técnicas, por exemplo a que foi abordada neste trabalho trata-se de uma falha lógica que impacta no setor financeiro da empresa, então é sempre bom lembrar deste detalhe para prosseguir nos testes.

10. Referências

CARNEIRO NETO, José André. **Um mapeamento de práticas em projetos de APIs REST**. 2020. 62 f. Tese (Doutorado) - Curso de Ciência da Computação, Universidade Federal de Pernambuco, Recife, 2020.

CLOUDFLARE. Endpoints. *In*: **Endpoints**. [*S. l.*], 2023. Disponível em: https://developers.cloudflare.com/fundamentals/global-configurations/lists/lists-api/endpoints/. Acesso em: 22 jun. 2023.

CLOUDFLARE. **HTTP x HTTPS: Quais são as diferenças?** [*S. l.*], 2023. Disponível em: https://www.cloudflare.com/pt-br/learning/ssl/why-is-http-not-secure/. Acesso em: 22 jun. 2023.

IMASTER. **Quebrando a banca com força bruta!** [*S. l.*], 2017. Disponível em: https://imasters.com.br/desenvolvimento/quebrando-banca-com-forca-bruta. Acesso em: 22 jun. 2023.

INTERNATIONAL BUSINESS MACHINES. **Http Request**. [*S. l.*], 7 jun. 2023. Disponível em: https://www.ibm.com/docs/en/cics-ts/5.3?topic=protocol-http-requests. Acesso em: 22 jun. 2023.

INTERNATIONAL BUSINESS MACHINES. **Http Responses**. [*S. l.*], 7 jun. 2023. Disponível em: https://www.ibm.com/docs/en/cics-ts/5.3?topic=protocol-http-requests. Acesso em: 22 jun. 2023.

KALI. **Ffuf**. [*S. l.*], 8 mar. 2023. Disponível em: https://www.kali.org/tools/ffuf/. Acesso em: 22 jun. 2023.

LI, Vickie. **Bug Bounty Bootcamp**: The Guide to Finding and Reporting Web Vulnerabilities. [*S. l.*: *s. n.*], 2021.

LOZANO, Carlos. **Bug Bounty Hunting Essentials**. [*S. l.*: *s. n.*], 2018.

MOZILLA. Http Request methods. *In*: MOZILLA. **Http Request methods**. [*S. l.*], 10 abr. 2023. Disponível em: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods. Acesso em: 22 jun. 2023.

SAMPAIO, Felipe Ferreira. **Uma análise prática das principais vulnerabilidades em aplicações web baseado no top 10 OWASP**. 2021. 61 f. Tese (Doutorado) - Curso de Redes de Computadores, Universidade Federal do Ceará, Quixadá, 2021.

W3SCHOOLS. **JSON vs XML**. [*S. l.*], 2022. Disponível em: https://www.w3schools.com/js/js_json_xml.asp. Acesso em: 22 jun. 2023.

YAWORSKI, Peter. **Real-World Bug Hunting**: A field guide to web hacking. [*S. l.*: *s. n.*], 2019.