Detecção de API Scrapers Através do Fluxo de *Hyperlinks*

Ailton Santos Filho[1], **Eduardo L. Feitosa**[1]

[1]Instituto de Computação (IComp) - Universidade Federal do Amazonas (UFAM)
Manaus – AM – Brazil

{assf,efeitosa}@icomp.ufam.edu.br

Abstract. *Web APIs represent an expanding market and today respond to a significant portion of Internet traffic. With the increasing popularity of Web APIs, developers are increasingly faced with malicious agents and, in many cases, current solutions to prevent abuse of web APIs are not able to mitigate unauthorized extraction (leakage) of data. This work presents a new approach for detecting anomalous clients by extracting systematic information from RESTful APIs, based on hyperlink flow analysis.*

Resumo. *As APIs Web representam um mercado em expansão e hoje correspondem a uma porção significativa do tráfego na Internet. Com o aumento da popularidade das APIs Web, os desenvolvedores se deparam cada vez mais com agentes maliciosos e, em muitos casos, as soluções atuais para impedir o abuso dessas APIs não são capazes de impedir extração não autorizada (vazamento) de dados. Este trabalho apresenta uma nova abordagem para detectar clientes anômalos realizando extração sistemática de informações das APIs RESTful, baseada na análise do fluxo de hyperlinks.*

1. Introdução

Atualmente existe uma proliferação das APIs (*Application Programming Interface*) Web que vem sendo empregadas em diversas aplicações, por empresas de diferentes segmentos, como bancário, automotivo e de telecomunicações. Segundo relatório da Akamai (2019) - a provedora de soluções de segurança e rede para diversas companhias, como *Airbnb* e *IBM* -, 83% do tráfego de rede monitorado em 2018 era oriundo de APIs. Um aumento substancial quando comparado aos 47% computados em 2014. Em parte, esse aumento se deve às vantagens oferecidas pela adoção das APIs, como: redução no tempo e custo de desenvolvimento de aplicações Web e móveis, facilidade na integração com soluções de terceiros e no desenvolvimento de *microservices*.

Infelizmente, a adoção das APIs Web também atraiu a atenção de atacantes que vêm realizando campanhas maliciosas que variam desde vazamento de dados de empresas como o *Facebook* [Cimpanu, 2019], até a exploração de vulnerabilidades de execução de códigos arbitrários [MITRE Corporation, 2019]. A combinação de APIs que expõem dados sensíveis e *bots* (clientes autônomos) está levando a massivos vazamentos de dados, que ocorrem através da prática de API *scraping* - extração de informações através da simulação do processo de interação de um cliente legítimo.

Este artigo, relacionado a um trabalho de doutorado em andamento, descreve uma abordagem que utiliza análise dos fluxos de *hyperlinks* [Ivanchikj et al., 2018] para detectar *bots* que realizam API *scraping* contra serviços *RESTful*. A solução proposta consiste

em aplicar a diagramação do fluxo de *hyperlinks* (conhecida como *RESTalk* [Ivanchikj et al., 2018]) para modelar um cliente legítimo de uma API RESTful e verificar, em tempo de execução, os *hyperlinks* tomados pelos clientes do *Web Service*. Dessa forma, é possível identificar clientes anômalos que utilizam *conhecimentos externos* para acessar os recursos da API, não seguindo os fluxos estabelecidos pela modelagem.

2. Fundamentação Teórica

Esta seção introduz os conceitos inerentes a *Web Services REST*, a prática de *API scraping*, assim como as contramedidas existentes para mitigá-la, e por fim, o modelo de diagramação *RESTalk*.

2.1. Web Services RESTful

Web Services REST são aplicações Web construídas sob os princípios de design REST (*Representational State Transfer*), apresentados na tese de doutorado de Fielding (2000), que descrevem como uma aplicação Web deve responder as requisições de seus clientes, com o objetivo de promover a escalabilidade e robustez de sistemas *hypermedia*. Diz-se que uma aplicação é RESTful quando cumpre todas as regras (*constraints*) estabelecidas pelo padrão REST, sendo elas: Interface uniforme, Arquitetura cliente-servidor, *Stateless*, *Cacheabilidade* e Sistema em camadas.

Por definição, *Web Services* têm como objetivo prover funcionalidades (ou recursos) a um conjunto de clientes [Booth et al., 2004]. Não obstante, recursos são a base da abstração de informações no estilo de arquitetura REST, sendo que qualquer informação nomeável pode ser considerada um recurso, como: imagens, arquivos ou textos. A identificação de um recurso é feita através de URIs (*Uniform Resource Identifiers*). No caso de APIs REST acessíveis via Web, URIs são tipicamente URLs (*Uniform Resource Locators*), chamados também de *endpoints*. Cada resposta de um servidor *RESTful* contém uma lista de URIs de recursos acessíveis, indicando quais requisições podem ser feitas pelo cliente a partir daquele momento. Esse comportamento é definido pela *constraint* conhecida como HATEOAS (*Hypermedia As The Engine Of Application State*), uma das *constraints* centrais da arquitetura REST.

2.2. API Scraping

API scraping é uma variação do termo *Web scraping*, utilizado na literatura para definir uma técnica de software para extrair informações de Websites, através da simulação do processo humano de exploração de páginas Web [Vargiu and Urru, 2013]. Segundo Mitchell (2014), trata-se de uma prática tão antiga quanto a própria Internet. As informações extraídas são utilizadas para diversos fins, como agregação de conteúdo e análise de propagandas, dentre outros tipos de processamentos de dados possíveis. Usualmente, essas informações são reunidas em uma nova página Web [Vargiu and Urru, 2013].

No que tange APIs Web especificamente, Jawad (2017) define *Web API scraping* como o ato de coletar um conjunto substancial de dados de uma API Web sem o consentimento dos fornecedores dessa API. Aplicações que realizem *scraping* são conhecidas como *scrapers* e funcionam através da implementação de clientes autônomos (*bots*) das fontes de informação exploradas. A construção de um *scraper* consiste na engenharia reversa da API Web, a fim de identificar os *endpoints* que proveem as informações desejadas

e quais os mecanismos envolvidos na obtenção da informação, como parâmetros aceitos pela API, *tokens* solicitados, entre outros. Os conhecimentos obtidos são embutidos em um *bot* que fará o *scraping* de fato.

Com a popularização das APIs Web, surgiram diversas soluções para a realização de *API Scraping* [Lospinoso, 2017; Zaslavskiy, 2019; Mitchell, 2014], incluindo soluções pagas [Scraper API, 2019], com o objetivo de facilitar o processo de extração de informações, reduzir o tempo de desenvolvimento de *bots* e evadir as contramedidas existentes.

As contramedidas existentes dividem-se em duas abordagens possíveis: a limitação da quantidade de requisições que um cliente pode fazer a um *Web Service* em um espaço de tempo (*throttling*) e a detecção de clientes anômalos. A primeira abordagem pode ser utilizada como estratégia para minimizar os efeitos dos *bots* e garantir que o *Web Service* não seja sobrecarregado. Ao atingir o limite máximo pré-estabelecido de chamadas, o cliente é obrigado a aguardar uma janela de tempo definida pelo fornecedor da aplicação para voltar a realizar chamadas e ser atendido [Chandramouli, 2019]. Para a implementação do *throttling*, é necessário definir duas políticas: como vai se dar a identificação de cada cliente e a política de limitação de acesso ao recurso.

Simpson (2019) analisou as políticas de *throttling* de três grandes mantenedoras de APIs (*GitHub, LinkedIn e Bitly*) para identificar as práticas adotadas por cada uma. Como resultado, o autor detectou que os três serviços adotam a identificação de clientes por meio de *tokens* (*API Keys*). A limitação do máximo de requisições que um cliente pode fazer, no entanto, é feita por hora, para a primeira empresa, por dia, para a segunda, e por minuto, hora e mês, para a terceira. Apesar do uso de *tokens* ser recomendado como uma medida de controle de acesso a recursos, especialmente contra o abuso de APIs Web, segundo a OWASP, não se pode depender exclusivamente dessa tecnologia, uma vez que *tokens* podem ser comprometidos com relativa facilidade [OWASP, 2019].

Quanto a detecção de *bots*, Jawad (2017) afirma que pode ser difícil distinguir clientes legítimos de ilegítimos através dos padrões de comportamento, uma vez que podem existir diferentes tipos de clientes legítimos consumindo APIs Web de diferentes formas (navegador, aplicação *mobile*, *IoT*, dentre outros). A autora definiu empiricamente diversas características comportamentais que podem ser utilizadas para identificar um *API scraper*, como: taxa de requisições constante ou com pequena variação por um longo período de tempo, alta taxa de requisições por segundo, modificações sistemáticas na URI em requisições contínuas, dentre outras. Essas características foram empregadas em três algoritmos de aprendizagem de máquina supervisionado para verificar se era possível detectar *bots* de API *scraping* realizando obtenções agressivas de dados.

Embora capaz de detectar comportamentos anômalos de clientes, a solução proposta por Jawad (2017) requer a obtenção de conjuntos de dados das requisições dos clientes de determinado *Web Service* para o treinamento e validação dos algoritmos de aprendizagem de máquina, uma vez que vários padrões de comportamento são inerentes ao serviço em questão. Ademais, quando ocorrerem modificações em *endpoints* (adição ou exclusão) é necessário realizar um novo treinamento.

2.3. *RESTalk*

Devido ao *REST* ser um estilo de arquitetura, não há um modelo padrão para realizar a modelagem dos serviços e clientes. Por isso, surgiram diferentes ferramentas para proje-

tar essas aplicações: por diagramas (como UML, *Business Process Model and Notation* (BPMN) e *RESTalk*) e por documentação (*Swagger* e *Blueprint*, por exemplo) [Ivanchikj et al., 2018].

Ivanchikj et al. (2018) propuseram o modelo de diagramação *RESTalk*, com o objetivo de representar graficamente todas as interações que podem ocorrer durante uma *conversação RESTful*. Segundo os autores, em aplicações *RESTful*, a *constraint* HATEOAS garante que, apesar de ser o cliente quem inicia uma conversação, é o servidor quem guia o cliente a partir daí, fornecendo as próximas possibilidades de requisições, através da lista de URIs embutidas em cada resposta. Esse comportamento é modelado pela diagramação *RESTalk*, através do que os autores chamaram: fluxo de *hyperlinks*.

3. Detecção de *API Scraping* Através do Fluxo de *Hyperlinks*

A capacidade da diagramação *RESTalk* de modelar as interações entre cliente e servidor em uma conversação *RESTful* legítima, através dos fluxos de *hyperlinks*, foi o que motivou a sua adoção neste trabalho. O fluxo de *hyperlinks* permite determinar como os identificadores dos recursos (URIs) são descobertos pelos clientes a partir de respostas anteriores enviadas pelo servidor.

A hipótese investigada neste trabalho é a de que a comparação dos fluxos de *hyperlinks* com as requisições realizadas por um cliente torna possível distinguir requisições que foram enviadas utilizando conhecimentos externos (*out-of-band* ou *hardcoded*) sobre os recursos. Para demonstrar que é possível utilizar o fluxo de *hyperlinks* para a detecção de *API Scraping*, foi realizado um experimento baseado no exemplo de *scraping* apresentado por Lospinoso (2017), onde um *endpoint* não documentado de uma aplicação real [Riot Games, 2018], que fornece os recursos *PLAYER_HISTORY*, tem informações extraídas através da ferramenta de *scraper* desenvolvida pelo autor, conhecida como *Abrade*. Após a extração das informações pelo *bot*, um dos clientes legítimos da API, disponível online[1], foi executado para que fosse determinado o fluxo de *hyperlinks* esperado de um cliente legítimo. A interações cliente-servidor de ambos os casos foram obtidas através da ferramenta de *proxy* `Burp Suite` [PortSwigger, 2019].

Em ambos os testes foi estabelecido arbitrariamente que a quantidade de cinco recursos seria obtida, por ser suficiente para ilustrar a diferença de comportamento. A Figura 1 apresenta um diagrama de sequência das requisições realizadas, construído a partir das requisições e respostas capturadas pelo *proxy*, onde o lado assinalado com **(a)** é referente ao cliente legítimo e o lado assinalado com **(b)** ao *bot*. Em **(a)**, o cliente obtém o recurso *GAME* (*endpoint game*), através da requisição (1), para acessar a lista dos recursos *PLAYER_HISTORY*(*endpoint player_history*) acessíveis e obtê-los através das requisições (2), (3), (4), (5) e (6). Em **(b)**, as interações com o servidor são baseadas em informações obtidas de fora da *RESTful conversation*. Não há propagação de informações vindo do servidor, uma vez que a requisição que informa a lista de recursos *PLAYER_HISTORY* disponíveis não foi feita (requisição ao *endpoint game*), ou seja, a partir das requisições-respostas observadas, não é possível determinar a procedência das informações utilizadas pelo cliente nas requisições (1), (2), (3), (4), (5).

[1]Disponível em `https://matchhistory.na.leagueoflegends.com/en/#page/landing-page`

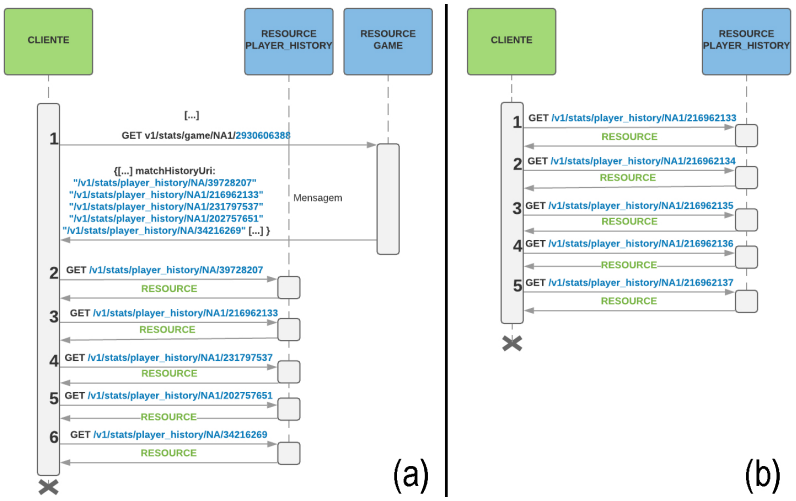

Figura 1. Diagrama de Sequência das requisições realizadas: Cliente legítimo X Bot

Através da análise manual das requisições-respostas é possível determinar a dependência entre os recursos *game* e *player_history*, pois o primeiro fornece as informações necessárias para acessar o segundo. No entanto, a análise manual de todo o conjunto de requisições-repostas feitas por todos os clientes de um *Web Service* torna essa abordagem impraticável em ambientes reais. Todavia, ao construir o diagrama `RESTalk`, mostrado na Figura 2, a dependência entre *endpoints* e o fluxo legítimo de *hyperlinks* é verificável de maneira direta. Quando adotada a diagramação `RESTalk` em fase de projeto, a análise do fluxo de *hyperlinks* oferece como vantagens: poder ser feita de forma automatizada, em tempo de execução ou *offline* (através de registros das requisições realizadas), ser independente de atualizações nos *endpoints* das APIs *RESTful* e não necessitar de dados de requisições-respostas anteriores.

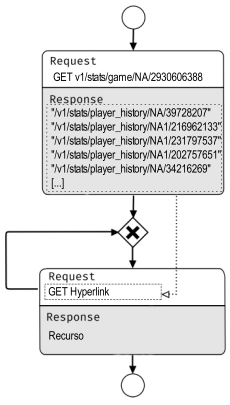

Figura 2. RESTalk: Fluxo de Hyperlinks do Cliente Legítimo

4. Trabalhos Futuros

Como atividade futura, pretende-se desenvolver um protótipo que realize a verificação da procedência das URIs sendo manipuladas pelos clientes de maneira autônoma. Uma vez que a adoção do RESTful é significantemente inferior a do REST [Rodríguez et al.,

2016], pretende-se expandir a solução proposta para *web services REST*. Ademais, como apresentado por Jawad (2017), diversos comportamentos nas requisições feitas podem ser incorporadas no processo de detecção de *bots*, como taxa de requisições no tempo, uma vez que o objetivo do agente malicioso é adquirir acesso ao maior montante de dados possível, no menor tempo possível.

5. Agradecimentos

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Código de Financiamento 001

Referências

Akamai (2019). State of the Internet / Security: Retail Attacks and API Traffic. *5*.

Booth, D., Haas, H. and Mccabe, F. (2004). Web Services Architecture.

Chandramouli, R. (2019). Security Strategies for Microservices-based Application Systems. [Online; https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-204-draft.pdf].

Cimpanu, C. (2019). Turkey fines Facebook for December 2018 API bug | ZDNet.

Fielding, R. T. (2000). Architectural Styles and the Design of Network-based Software Architectures. University of California, Irvine.

Ivanchikj, A., Pautasso, ·. C. and Schreier, S. (2018). Visual modeling of RESTful conversations with RESTalk. Software & Systems Modeling *17*, 17.

Jawad, D. (2017). Detection of Web API Content Scraping An Empirical Study of Machine Learning Algorithms. [Online; http://www.nada.kth.se/~ann/exjobb/dina_jawad.pdf].

Lospinoso, J. (2017). Abrade, a high-throughput web API scraper. [Online; https://lospi.net/cpp/developing/software/2017/09/15/abrade-web-scraper.html].

Mitchell, R. (2014). Web Scraping with Python. O'Reilly.

MITRE Corporation (2019). Common vulnerabilities and exposures - CVE-2019-5678.

OWASP (2019). REST Security Cheat Sheet - OWASP. [Online; https://www.owasp.org/index.php/REST_Security_Cheat_Sheet].

PortSwigger (2019). Burp Suite Scanner - PortSwigger.

Riot Games (2018). Riot Developer Portal. [Online; https://developer.riotgames.com/api-methods/].

Rodríguez, C., Baez, M., Daniel, F., Casati, F., Trabucco, J. C., Canali, L. and Percannella, G. (2016). REST APIs: A large-scale analysis of compliance with principles and best practices.

Scraper API (2019). Scraper API. [Online; https://www.scraperapi.com/].

Simpson, J. (2019). Everything You Need To Know About API Rate Limiting | Nordic APIs |. [Online; https://nordicapis.com/everything-you-need-to-know-about-api-rate-limiting/].

Vargiu, E. and Urru, M. (2013). Exploiting web scraping in a collaborative filtering-based approach to web advertising. Artificial Intelligence Research *2*.

Zaslavskiy, A. (2019). API Scraping in the Real World. [Online; https://www.codementor.io/blog/api-scraping-5fq1gtd4ah].