

Como lidar com API Keys da forma mais segura possível



victor.melo@thoughtworks.com

Quem sou eu



Engenheiro de Software Sênior



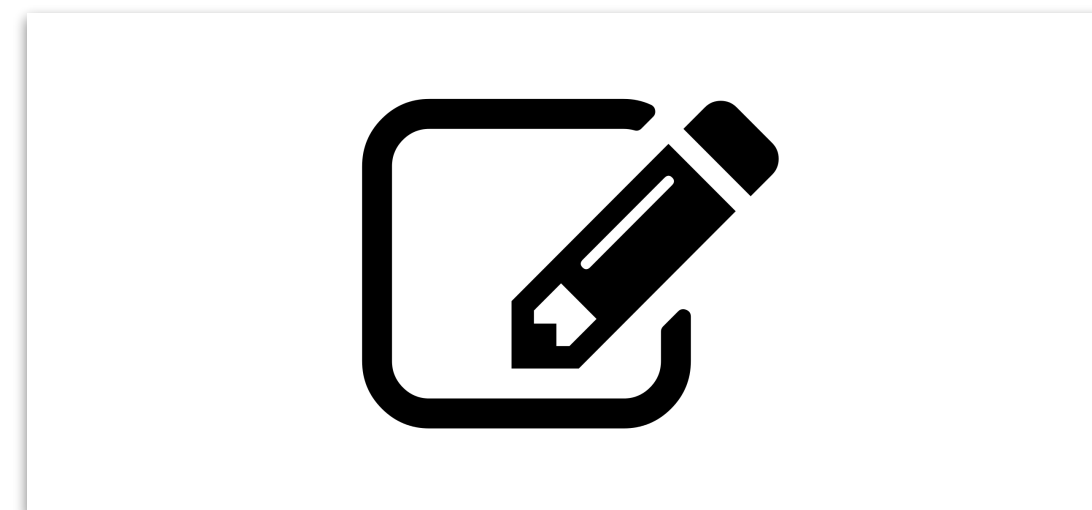
Formando em Ciência da Computação



Ex Aluno do Apple Developer Academy



Time is Money Calculator na App Store



<https://victor.dev.br>



[linkedin.com/in/vsmelo/](https://www.linkedin.com/in/vsmelo/)

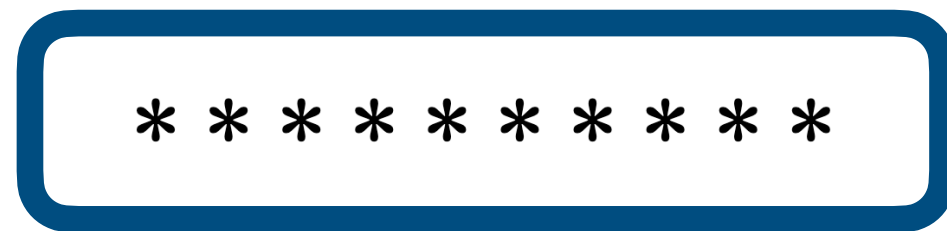
Segredos 🤫

O que são Segredos



Credencial digital usada para acessar algum recurso computacional

Exemplos de Segredos



Senhas



Certificados privados



Tokens de autorização



API keys

Vulnerabilidades 🙄🙄

Dois tipos de vulnerabilidade

(Na minha opinião)



Vulnerabilidades em
Comunicação



Vulnerabilidades
Locais

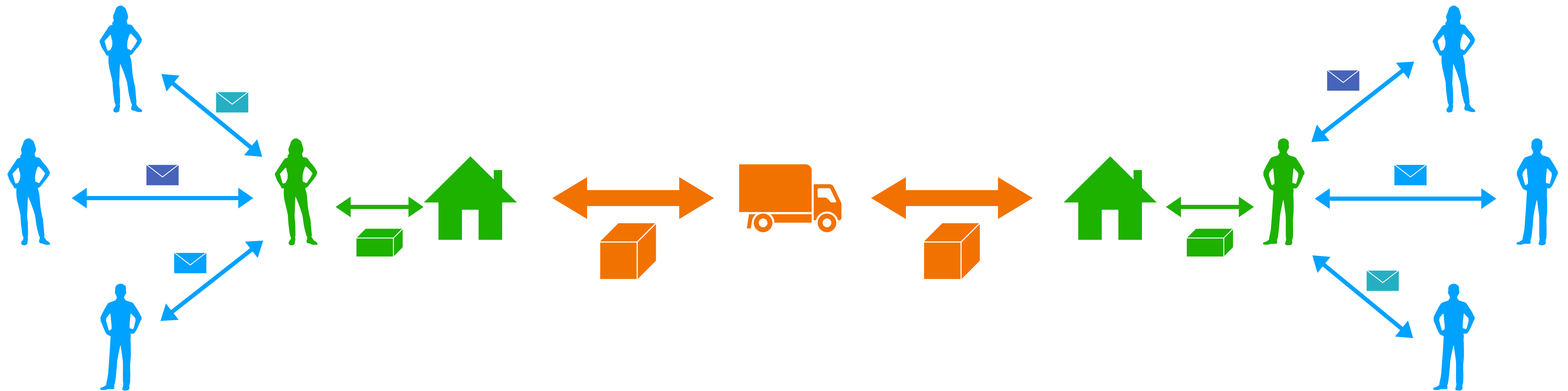
Camadas de Rede


Vulnerabilidade em Comunicação

Camada de Aplicação	Protocolos para comunicação entre aplicações especializadas HTTP, SMTP, FTP, Protocolos próprios
Camada de Transporte	Protocolos para comunicação entre processos TCP, UDP
Camada de Rede	Protocolos para comunicação entre sistemas finais (hosts) IP
Camada de Enlace	Protocolos para comunicação entre um novo e outro (roteadores) Ethernet, Wi-Fi
Camada Física	Protocolo para transmitir bits em espaço físico Protocolo para cabo de fio trançado, para fibra, ...

Analogia

Correios



 Camada de Aplicação

 Camada de Transporte

 Camada de Rede

Onde entra a segurança? 🤔

Camadas de Rede

Vulnerabilidade em Comunicação



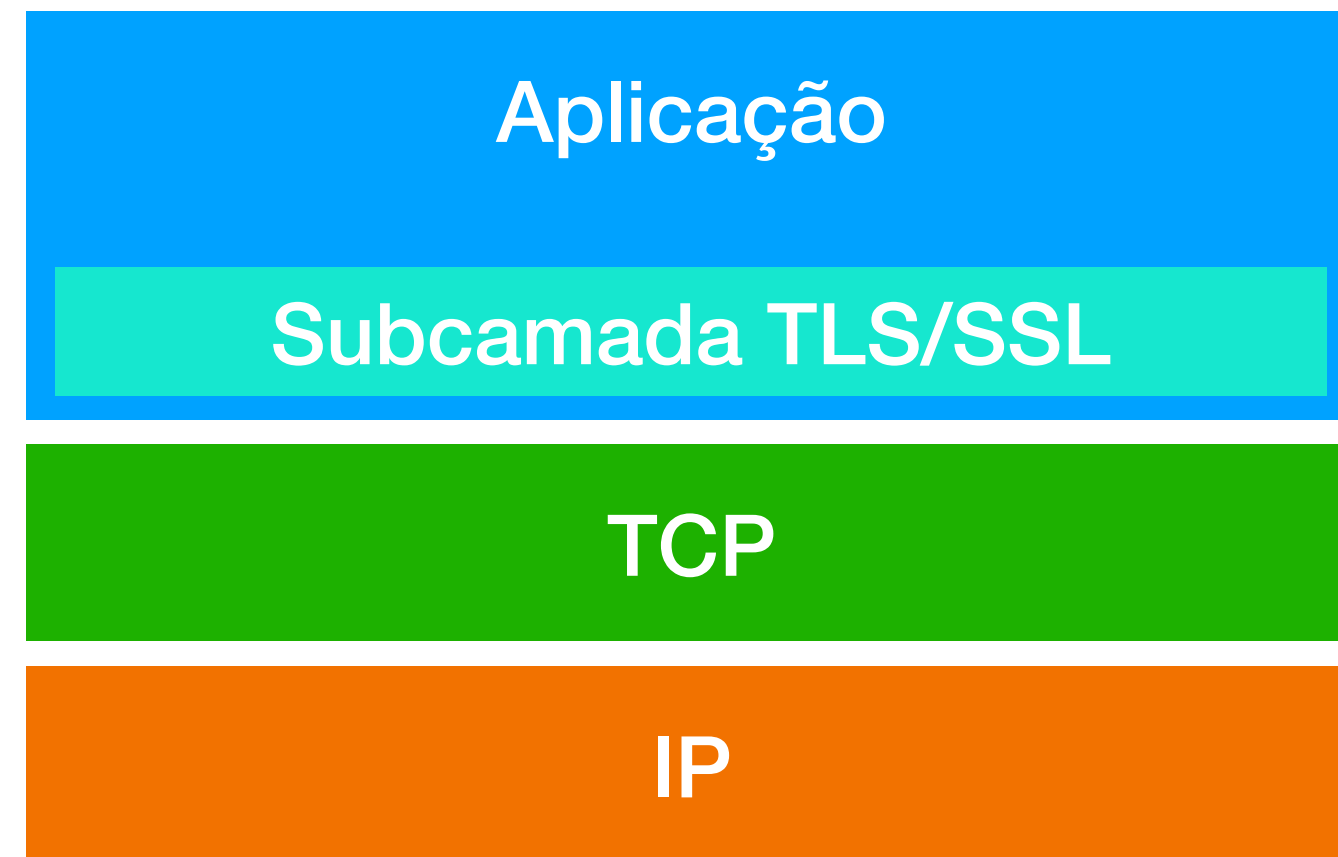
Protocolos para comunicação entre processos

UDP: simples. Não garante que um pacote enviado será recebida (carta perdida)

TCP: mais complexo. Garante entrega ordenada dos pacotes.

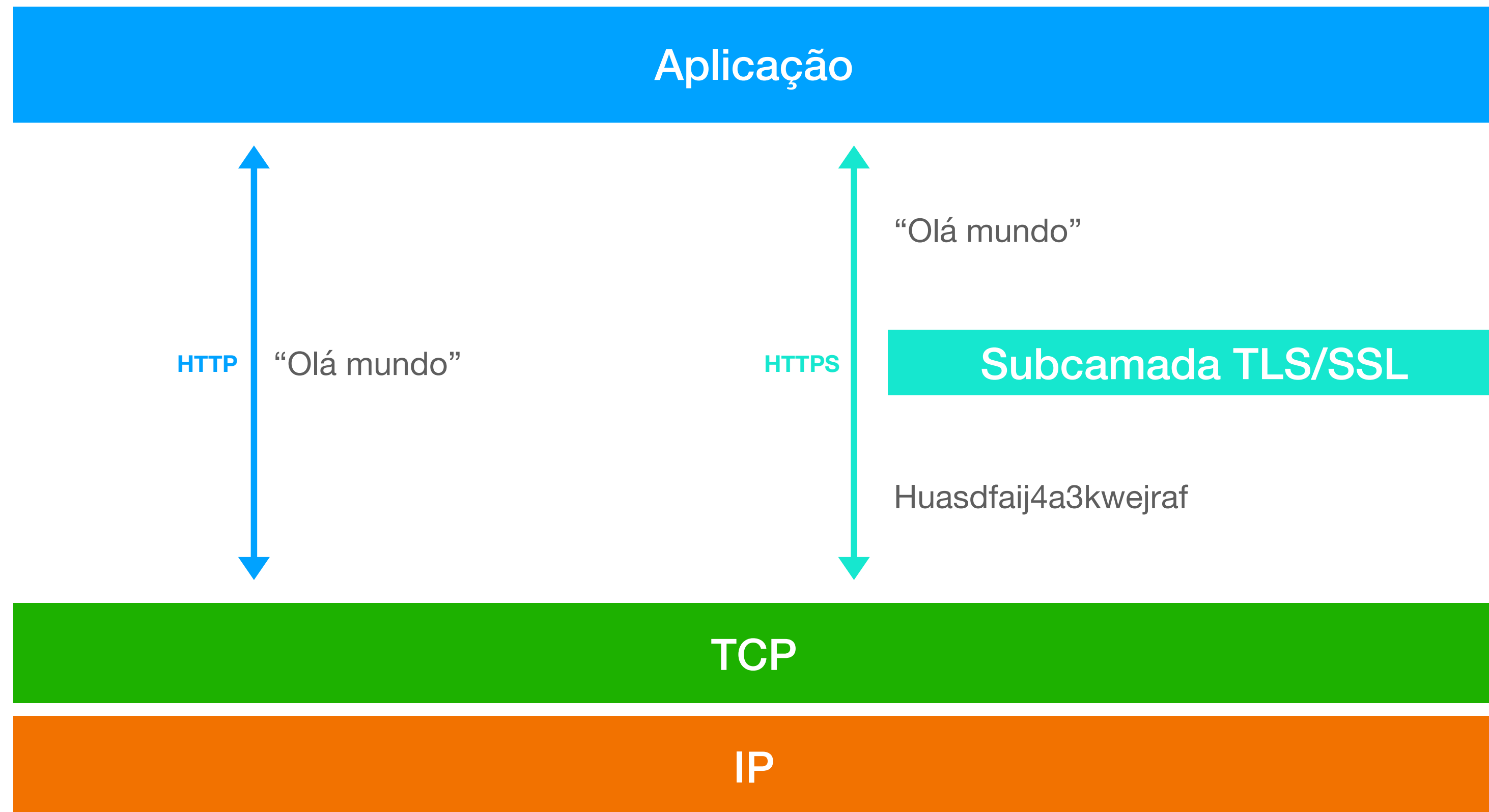
Nenhum protocolo garante o sigilo do pacote (carta é sempre enviada sem um lacre)

Vulnerabilidade em Comunicação

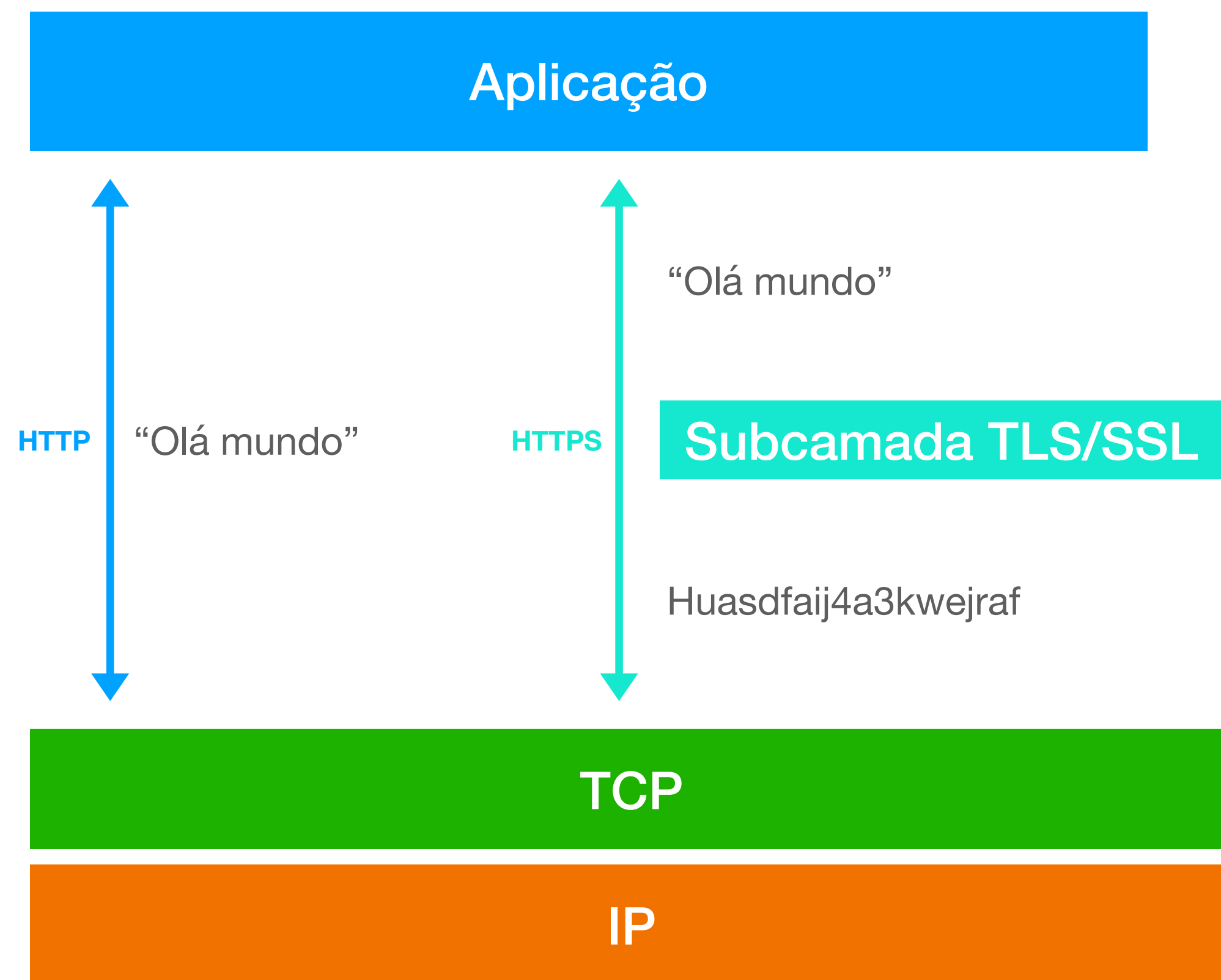


Subcamada SSL criptografa e descriptografa mensagens

Exemplo



Pontos de Vulnerabilidade



Vulnerabilidade local:

Conta que usuário instale um software malicioso ou que o desenvolvedor não tenha protegido os segredos em sua aplicação.



- Engenharia reversa
- Cavalo de Tróia
- Ransomware
- ...

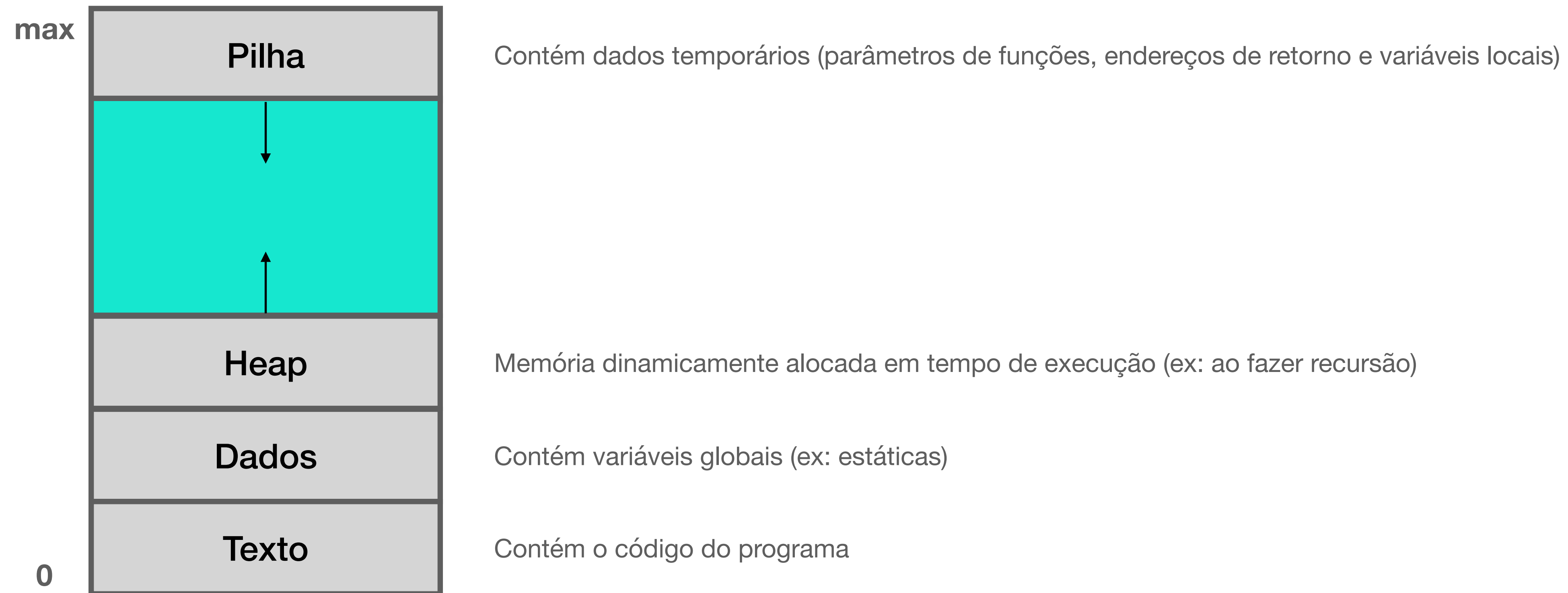
Vulnerabilidade na comunicação:

Conta que não esteja usando a subcamada TLS/SSL (ou esteja usando uma criptografia fraca)

- Rogue Access Point
- Spoofing
- Sniffing
- ...

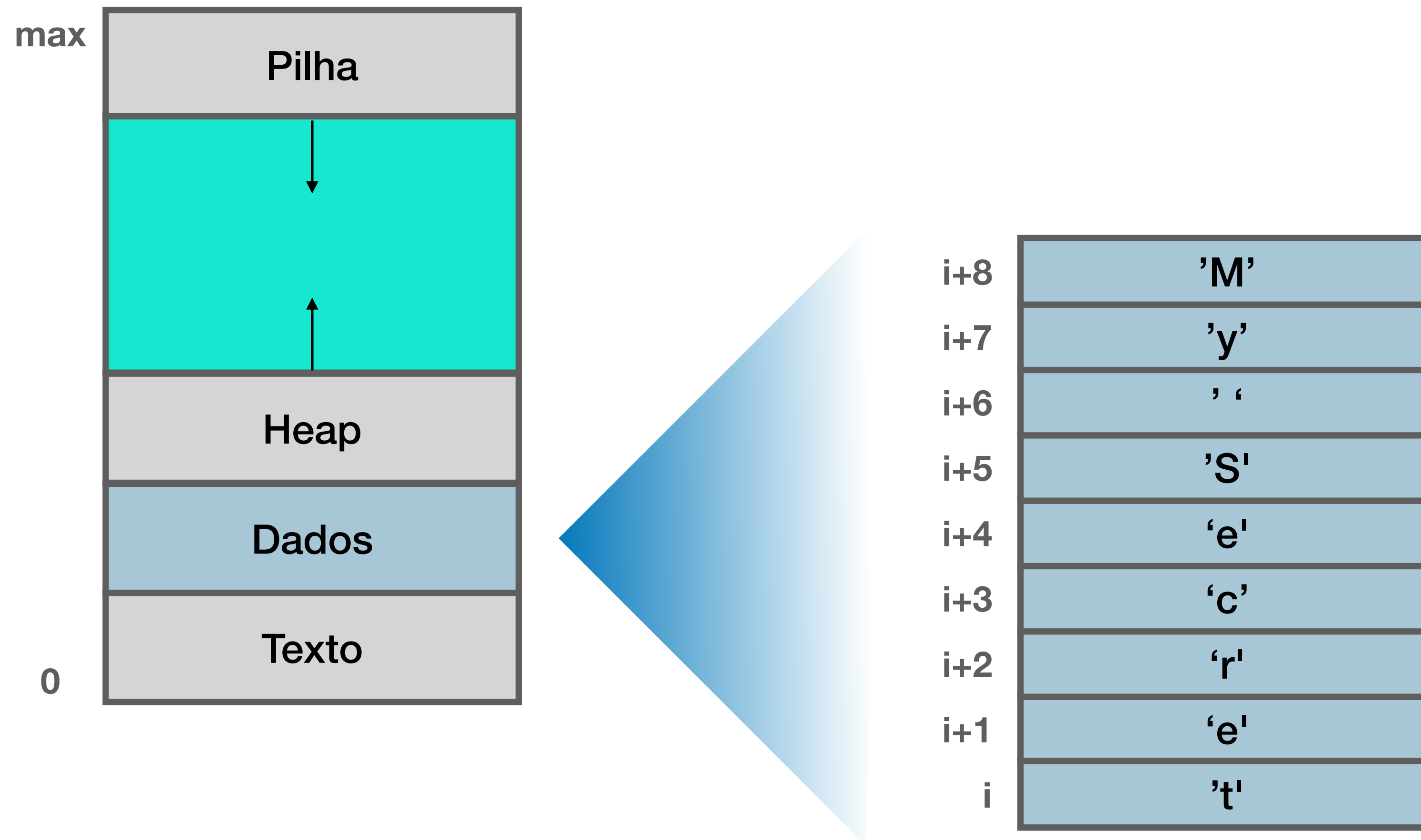
Engenharia Reversa

Processo em Memória Virtual



Engenharia Reversa

Processo em Memória Virtual



Como podemos nos proteger? 🤯

OWASP Top 10 Mobile Risks

Top 10 Mobile Risks - Final List 2016

- [M1: Improper Platform Usage](#)
- [M2: Insecure Data Storage](#)
- [M3: Insecure Communication](#)
- [M4: Insecure Authentication](#)
- [M5: Insufficient Cryptography](#)
- [M6: Insecure Authorization](#)
- [M7: Client Code Quality](#)
- [M8: Code Tampering](#)
- [M9: Reverse Engineering](#)
- [M10: Extraneous Functionality](#)

How Do I Prevent 'Code Tampering'?

The mobile app must be able to detect at runtime that code has been added or changed from what it knows about its integrity at compile time. The app must be able to react appropriately at runtime to a code integrity violation.

The remediation strategies for this type of risk is outlined in more technical detail within the [OWASP Reverse Engineering and Code Modification Prevention Project](#).

Android Root Detection Typically, an app that has been modified will execute within a Jailbroken or rooted environment. As such, it is reasonable to try and detect these types of compromised environments at runtime and react accordingly (report to the server or shutdown). There are a few common ways to detect a rooted Android device: Check for test-keys

- Check to see if build.prop includes the line `ro.build.tags=test-keys` indicating a developer build or unofficial ROM

Check for OTA certificates

OWASP Guides



Input Validation Testing

Testing for Reflected Cross Site Scripting (OTG-INPVAL-001)

Testing for Stored Cross Site Scripting (OTG-INPVAL-002)

Testing for HTTP Verb Tampering (OTG-INPVAL-003)

Testing for HTTP Parameter pollution (OTG-INPVAL-004)

Testing for SQL Injection (OTG-INPVAL-005)

Oracle Testing

MySQL Testing

SQL Server Testing

Testing PostgreSQL (from OWASP BSP)

MS Access Testing

Testing for NoSQL injection

Testing for LDAP Injection (OTG-INPVAL-006)

Testing for ORM Injection (OTG-INPVAL-007)

Testing for XML Injection (OTG-INPVAL-008)

Testing for SSI Injection (OTG-INPVAL-009)

Testing for XPath Injection (OTG-INPVAL-010)

IMAP/SMTP Injection (OTG-INPVAL-011)

Testing for Code Injection (OTG-INPVAL-012)

Testing for Local File Inclusion

Testing for Remote File Inclusion

Testing for Command Injection (OTG-INPVAL-013)

Testing for Buffer overflow (OTG-INPVAL-014)

Testing for Heap overflow

Testing for Stack overflow

Testing for Format string

Testing for incubated vulnerabilities (OTG-INPVAL-015)

Testing for HTTP Splitting/Smuggling (OTG-INPVAL-016)

Caso Real 🌐

Problema

```
final class Config {  
    static let APIKey: String = {  
        #if DEBUG  
        return "Bla bla bla API Key de Debug"  
        #else  
        return "Bla bla bla API Key de Producao"  
        #endif  
    }()  
}
```

Segredos hardcoded

Problema

- Segredos enviados para o repositório.
- Segredos expostos na área de dados do processo.

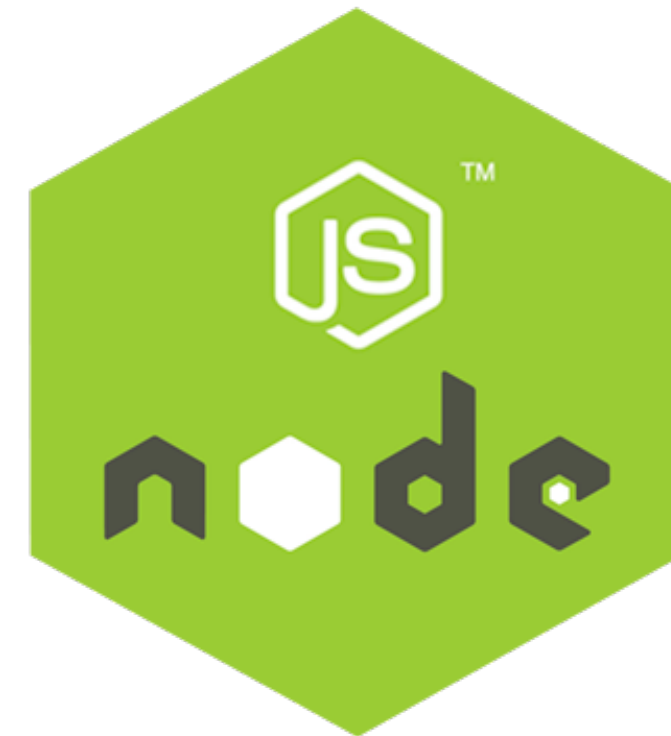
```
final class Config {  
    static let APIKey: String = {  
        #if DEBUG  
        return "Bla bla bla API Key de Debug"  
        #else  
        return "Bla bla bla API Key de Producao"  
        #endif  
    }()  
}
```

Segredos hardcoded

Soluções Consideradas



AWS Secrets Manager



Buscar das nossas APIs



Solução inspirada no arquivo .env

Soluções Consideradas



AWS Secrets Manager

Implementação mais complexa pra nossa arquitetura



Buscar das nossas APIs

Implementação mais complexa pra nossa arquitetura



Solução inspirada no arquivo .env

Soluciona problema com repositório, mas não com engenharia reversa

Ação Tomada



Solução inspirada no arquivo .env

- Arquivo **json** compartilhado por canal seguro.
 - Script executado em tempo de build, parseando o json e gerando arquivo swift com segredos ofuscados.
- (Em Android, o arquivo final gerado é um binário em outra linguagem, como C ou similar).

Demo



Conclusão

- A forma mais segura é a forma que mais dificulta o vazamento.
- Pra conhecer como dificultar, precisa conhecer as vulnerabilidades.
 - <https://owasp.org/www-project-mobile-top-10/>
- A implementação depende das restrições do projeto.

Conclusão

Sejam curiosos 🤔

how to handle secrets mobile

All Videos Images News Shopping More Tools

About 51,200,000 results (0.67 seconds)

[https://medium.com › advanced-ios-engineering › mobi...](https://medium.com/advanced-ios-engineering/mobi...)
Mobile Secrets. Handle secrets the secure way with ease
Oct 5, 2019 — In this article, I will show how to use **Mobile Secrets** gem alongside with GPG to **handle app secrets** the most secure way. **Mobile Secrets** is ...

[https://guides.codepath.com › android › Storing-Secret-...](https://guides.codepath.com/android/Storing-Secret-...)
Storing Secret Keys in Android | CodePath Android Cliffnotes
Hidden as constants in source code. The simplest approach for storing **secrets** in to keep them as resource files that are simply not checked into source **control**.

[https://www.appdome.com › no-code-data-encryption](https://www.appdome.com/no-code-data-encryption)
How to Store Encrypted Secrets in Android & iOS Memory
Appdome's Storing in Protected Memory enables you to protect those **secrets** by storing them in the **mobile** app encrypted memory. This Knowledge Base article ...

[https://hackernoon.com › hands-on-mobile-api-security...](https://hackernoon.com/hands-on-mobile-api-security...)
Hands On Mobile API Security: Get Rid of Client Secrets
May 3, 2017 — The NASA API calls are **handled** in the `src/api/nasa.js` module. When the proxy

Conclusão

How Bad Can It Get? Characterizing Secret Leakage in Public GitHub Repositories

Michael Meli
North Carolina State University
mjmeli@ncsu.edu

Matthew R. McNiece
North Carolina State University
Cisco Systems, Inc.
mrmcniec@ncsu.edu

Bradley Reaves
North Carolina State University
bgreaves@ncsu.edu

Abstract—GitHub and similar platforms have made public collaborative development of software commonplace. However, a problem arises when this public code must manage authentication secrets, such as API keys or cryptographic secrets. These secrets must be kept private for security, yet common development practices like adding these secrets to code make accidental leakage frequent. In this paper, we present the first large-scale and longitudinal analysis of secret leakage on GitHub. We examine billions of files collected using two complementary approaches: a nearly six-month scan of real-time public GitHub commits and a public snapshot covering 13% of open-source repositories. We focus on private key files and 11 high-impact platforms with distinctive API key formats. This focus allows us to develop conservative detection techniques that we manually and automatically evaluate to ensure accurate results. We find that not only is secret leakage pervasive — affecting over 100,000 repositories — but that *thousands* of new, unique secrets are leaked every day. We also use our data to explore possible root causes of leakage and to evaluate potential mitigation strategies. This work shows that secret leakage on public repository platforms is rampant and far from a solved problem, placing developers and services at persistent risk of compromise and abuse.

I. INTRODUCTION

Since its creation in 2007, GitHub has established a massive community composed of nearly 30 million users and 24 million public repositories [1], [11], [55]. Beyond merely storing code, GitHub is designed to encourage public collaborative

leaked in this way have been exploited before [4], [8], [21], [25], [41], [46]. While this problem is known, it remains unknown to what extent secrets are leaked and how attackers can efficiently and effectively extract these secrets.

In this paper, we present the first comprehensive, longitudinal analysis of secret leakage on GitHub. We build and evaluate two different approaches for mining secrets: one is able to discover 99% of newly committed files containing secrets in real time, while the other leverages a large snapshot covering 13% of all public repositories, some dating to GitHub's creation. We examine millions of repositories and billions of files to recover hundreds of thousands of secrets targeting 11 different platforms, 5 of which are in the Alexa Top 50. From the collected data, we extract results that demonstrate the worrying prevalence of secret leakage on GitHub and evaluate the ability of developers to mitigate this problem.

Our work makes the following contributions:

- **We perform the first large-scale systematic study across billions of files that measures the prevalence of secret leakage on GitHub by extracting and validating hundreds of thousands of potential secrets.** We also evaluate the time-to-discovery, the rate and timing of removal, and the prevalence of co-located secrets. Among other findings, we find thousands of

2018 ACM/IEEE 40th International Conference on Software Engineering: Software Engineering in Practice

Protecting Million-User iOS Apps with Obfuscation: Motivations, Pitfalls, and Experience

Pei Wang*
pxw172@ist.psu.edu
The Pennsylvania State
University

Dinghao Wu
dhw@ist.psu.edu
The Pennsylvania State
University

Zhaofeng Chen
chenzhaofeng@baidu.com
Baidu X-Lab

Tao Wei
lenx@baidu.com
Baidu X-Lab

ABSTRACT

In recent years, mobile apps have become the infrastructure of many popular Internet services. It is now fairly common that a mobile app serves a large number of users across the globe. Different from web-based services whose important program logic is mostly placed on remote servers, many mobile apps require complicated client-side code to perform tasks that are critical to the businesses. The code of mobile apps can be easily accessed by any party after the software is installed on a rooted or jailbroken device. By examining the code, skilled reverse engineers can learn various knowledge about the design and implementation of an app. Real-world cases have shown that the disclosed critical information allows malicious parties to abuse or exploit the app-provided services for unrightful profits, leading to significant financial losses for app vendors.

One of the most viable mitigations against malicious reverse engineering is to obfuscate the software before release. Despite

ACM Reference Format:

Pei Wang, Dinghao Wu, Zhaofeng Chen, and Tao Wei. 2018. Protecting Million-User iOS Apps with Obfuscation: Motivations, Pitfalls, and Experience. In *ICSE-SEIP '18: 40th International Conference on Software Engineering: Software Engineering in Practice Track, May 27–June 3, 2018, Gothenburg, Sweden*. ACM, New York, NY, USA, 10 pages. <https://doi.org/10.1145/3183519.3183524>

1 INTRODUCTION

During the last decade, mobile devices and apps have become the foundations of many million-dollar businesses operated globally. However, the prosperity has drawn many malevolent attempts to make unjust profits by exploiting the security and privacy loopholes in popular mobile software.

In recent years, we noticed that security breaches targeting mo-

2018 25th Asia-Pacific Software Engineering Conference (APSEC)

An Empirical Study of SDK Credential Misuse in iOS Apps

Haohuang Wen
School of Software Engineering
South China University of Technology
Guangzhou, China
onehouwong@gmail.com

Juanru Li
Lab of Cryptology and Computer Security
Shanghai Jiao Tong University
Shanghai, China
jarod@sjtu.edu.cn

Yuanyuan Zhang, Dawu Gu
Lab of Cryptology and Computer Security
Shanghai Jiao Tong University
Shanghai, China
{yyjess, dwgu}@sjtu.edu.cn

Abstract—During the development of web-based mobile apps, third-party SDKs (Software Development Kit) are frequently used to facilitate the integration of certain functionality such as push notification and mobile payment. Unfortunately, security issues are often considered as a second-tier problem and app developers are prone to implement apps with SDK misuses. Among those typical SDK misuses, the misuse of credentials is the one that introduces serious security threats. A credential is a set of unique information (e.g., APP ID, App Token, etc) allocated to a specific developer to help app authenticate the identity. However, if not properly used, the credential can be easily obtained by attackers and leads to not only the leak of confidential information of mobile developers but also direct

Since credentials are often the only authentication information for many web services, mobile developers need to properly manage them and should be extra vigilant about credential security. Unfortunately, the use of mobile SDKs often weakens this assumption. For one thing, SDK providers often publish vague instructions on how to use credentials, leading to mistakenly embedded and protected credentials. For another, even if a correct guide of credential management is published, it involves many aspects of protection and is often very complex. Developers still face various challenges in implementing a secure protection scheme. As a result, many

Obrigado!



LinkedIn®



[linkedin.com/in/vsmelo/](https://www.linkedin.com/in/vsmelo/)