FACULDADE DE TECNOLOGIA DE AMERICANA

Curso Superior de Tecnologia em Segurança da Informação

Renan Felipe Coglioni Graciano

Aplicação de técnicas de segurança no desenvolvimento de um aplicativo Android.

Americana, SP

2017

FACULDADE DE TECNOLOGIA DE AMERICANA

Curso Superior de Tecnologia em Segurança da Informação

Renan Felipe Coglioni Graciano

Aplicação de técnicas de segurança no desenvolvimento de um aplicativo Android.

Trabalho de Conclusão de Curso desenvolvido em cumprimento à exigência curricular do Curso Superior de Tecnologia em Segurança da Informação, sob a orientação do Prof. Diógenes de Oliveira.

Área de concentração: Segurança da Informação.

Americana, SP

2017

FICHA CATALOGRÁFICA – Biblioteca Fatec Americana - CEETEPS
Dados Internacionais de Catalogação-na-fonte

G756a GRACIANO, Renan Felipe Coglioni

Aplicação de técnicas de segurança no desenvolvimento de um aplicativo Android./ Renan Felipe Coglioni Graciano. – Americana: 2017.

45f.

Monografia (Curso de Tecnologia em Segurança da Informação) - - Faculdade de Tecnologia de Americana – Centro Estadual de Educação Tecnológica Paula Souza

Orientador: Prof. Ms. Diógenes de Oliveira

1. Segurança em sistemas de informação 2. Dispositivos móveis – aplicativos 3. Android – aplicativos 4. C# - linguagem de programação I. OLIVEIRA, Diógenes de II. Centro Estadual de Educação Tecnológica Paula Souza – Faculdade de Tecnologia de Americana

Renan Felipe Coglioni Graciano

APLICAÇÃO DE TÉCNICAS DE SEGURANÇA NO DESENVOLVIMENTO DE UM APLICATIVO ANDROID

Trabalho de graduação apresentado como exigência parcial para obtenção do título de Tecnólogo em Segurança da Informação pelo CEETEPS/Faculdade de Tecnologia – FATEC/ Americana.
Área de concentração: Segurança da informação.

Americana, 28 de junho de 2017.

Banca Examinadora:

Diógenes de Oliveira (Presidente)
Mestre
Fatec Americana

Ricardo Kiyoshi Batori (Membro)
Especialista
Fatec Americana

Ana Lucia Spigolon (Membro)
Graduada
Fatec Americana

RESUMO

O presente trabalho discorre sobre a necessidade de se considerar práticas de segurança no desenvolvimento de aplicativos para dispositivos móveis, através de recomendações baseadas em vulnerabilidades conhecidas e documentadas por conceituadas organizações. A partir de uma revisão bibliográfica sobre segurança da informação e estudos de vulnerabilidades no processo de desenvolvimento, buscamos aplicar algumas técnicas de segurança no desenvolvimento prático de um aplicativo Android construído com o Framework Xamarin e linguagem de programação C#, integrado a um sistema web existente.

Palavras Chave: Software seguro; Segurança da Informação; Dispositivos móveis; Android; Xamarin; C#.

ABSTRACT

This paper discusses the need to consider security practices in the development of mobile applications through recommendations based on vulnerabilities known and documented by reputable organizations. From a bibliographic review on information security and vulnerability studies in the development process, we seek to apply some security techniques in the practical development of an Android application built with the Xamarin Framework and C # programming language, integrated with an existing web system.

Keywords: *Secure software; Information Security; Mobile Devices; Android; Xamarin; C#.*

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

1. INTRODUÇÃO

Atualmente, com a facilidade que se tem de adquirir um dispositivo móvel e, instalar aplicativos de diferentes tipos, para uso no lazer ou profissional, que possibilitam a interoperabilidade entre sistemas de diferentes plataformas, abrem brechas para invasões mal-intencionadas, colocando em risco os dados pessoais ou mesmo os organizacionais, e assim, um simples aparelho móvel, aparentemente inofensivo, pode se tornar o responsável pelo declínio de uma organização.

Segundo Batori (2012), a segurança de uma aplicação, está fundamentada em três aspectos principais: a infraestrutura tecnológica, as pessoas envolvidas no desenvolvimento e a própria aplicação. Com base nessas afirmações, este trabalho buscou aplicar os conceitos e técnicas de segurança, visando minimizar a vulnerabilidade decorrente de métodos de programação inadequados, e assim, promover o desenvolvimento de um aplicativo Android, integrado a um sistema web na nuvem, voltado para a coleta e manutenção de dados relacionados a área do agronegócio.

Nos capítulos 2 e 3, foram revistos os conceitos básicos de segurança da informação, observando-se alguns dados de pesquisas relacionadas a vulnerabilidades de sistemas e os riscos envolvidos nesses desenvolvimentos. No capítulo 4 foram abordadas revisões sobre o ciclo de vida no desenvolvimento seguro, observando os principais aspectos relacionados ao desenvolvimento de aplicativos para dispositivos móveis.

No capítulo 5, aborda-se o desenvolvimento prático do aplicativo, apresentando as técnicas e métodos que foram utilizados para a manutenção e sincronização dos dados entre o aplicativo móvel e o servidor web, dando-se relevância aos métodos de mitigação de riscos e vulnerabilidades, sem comprometer o desempenho do sistema.

Como um todo, a pesquisa, tem como objetivos específicos, demonstrar o processo de identificação e mitigação de vulnerabilidades encontradas em um aplicativo Android. Através de uma análise ameaças apresenta-se as técnicas específicas para garantir que elas não possam ser exploradas.

2. SEGURANÇA DA INFORMAÇÃO

Com a constante evolução tecnológica que se presencia diariamente e, o crescente volume de informações geradas pelas organizações, considera-se muitas vezes que elas são os ativos mais valiosos de uma empresa, o que gera uma preocupação a respeito de sua segurança.

Conforme citado por Harris (2013) a maioria das empresas não quer gerenciar *firewalls*, sistemas de detecção de intrusão, criptografias, produtos *anti-malware* e, ainda ter que se preocupar com os diversos regulamentos de segurança (SOX, GLBA, PCI-DSS, HIPAA...). O que eles realmente querem é, desenvolver seu produto, vende-lo, e ir para casa.

Porém, esses dias simples não existem mais e, as informações mantidas pelas organizações estão em constante ameaça de atacantes querendo rouba-las, sejam eles pessoas desconhecidas, ou até mesmo funcionários insatisfeitos.

A segurança da informação se resume no conceito da necessidade de garantir a confidencialidade, integridade e disponibilidade (tripé da segurança), de um ativo através de técnicas de proteção contra acessos não autorizados, negações de serviço, injeções, falsificações, mal-uso das informações e diversos outros tipos de ataque.

Detalha-se então o tripé da segurança da informação, da seguinte maneira proposta por Goodrich e Tamassia (2013):

- **Confidencialidade**: É a proteção da informação, de maneira que somente pessoas autorizadas tenham acesso a ela.

> No contexto de segurança de computadores, confidencialidade é evitar a revelação não autorizada de informação. Isto é, confidencialidade envolve a proteção de dados, proporcionando acesso àqueles que são autorizados a vê-los e não permitindo que outros saibam algo a respeito de seu conteúdo. (GOODRICH; TAMASSIA, 2013)

Manter a informação secreta, é a essência da segurança da informação, seja através de encriptação, controles de acesso, autenticações, autorização e até mesmo segurança física.

- **Integridade**: Garantia de que uma mensagem, dado ou informação, não tenha sido alterada entre a sua origem e o seu destino.

  Existem diversas maneiras de se comprometer a integridade de um dado. Porém, nem todas as vezes, através de algo malicioso, muitas vezes uma informação é comprometida involuntariamente. Pensando nisso, é importante que um sistema forneça medidas para garantia da integridade em quaisquer casos, através de cópias de segurança, somas de verificação, códigos de correção de dados entre outras maneiras.

- **Disponibilidade**: Garantia de que uma informação estará sempre disponível e possa ser modificada por aqueles que possuam autorização para fazer isso. Em muitos ataques, a disponibilidade das informações é o alvo. Através de negações de serviço, por exemplo, um *hacker* pode garantir que nenhum usuário de um banco, consiga efetuar transações em determinada hora do dia, ou até mesmo, por tempo indeterminado, exigindo quantias em dinheiro para a revogação do ataque.

  Um sistema seguro deve ser capaz de auxiliar no combate de ataques a disponibilidade, através de proteções físicas, redundâncias computacionais e muitas outras medidas.

3. DESENVOLVIMENTO SEGURO

Atualmente é essencial que os fornecedores e desenvolvedores de *software* abordem ameaças de segurança em seus projetos como importantes ferramentas para uma empresa. Segundo Batori (2012), a segurança de uma aplicação, possui três aspectos principais: a infraestrutura tecnológica, os aspectos humanos e a própria aplicação, levando em conta que a segurança em um software não é algo que se possa ver com facilidade, pois, diferente de outros requisitos como performance, quantidade de falhas e interface com usuário, a segurança é difícil de se quantificar ou qualificar.

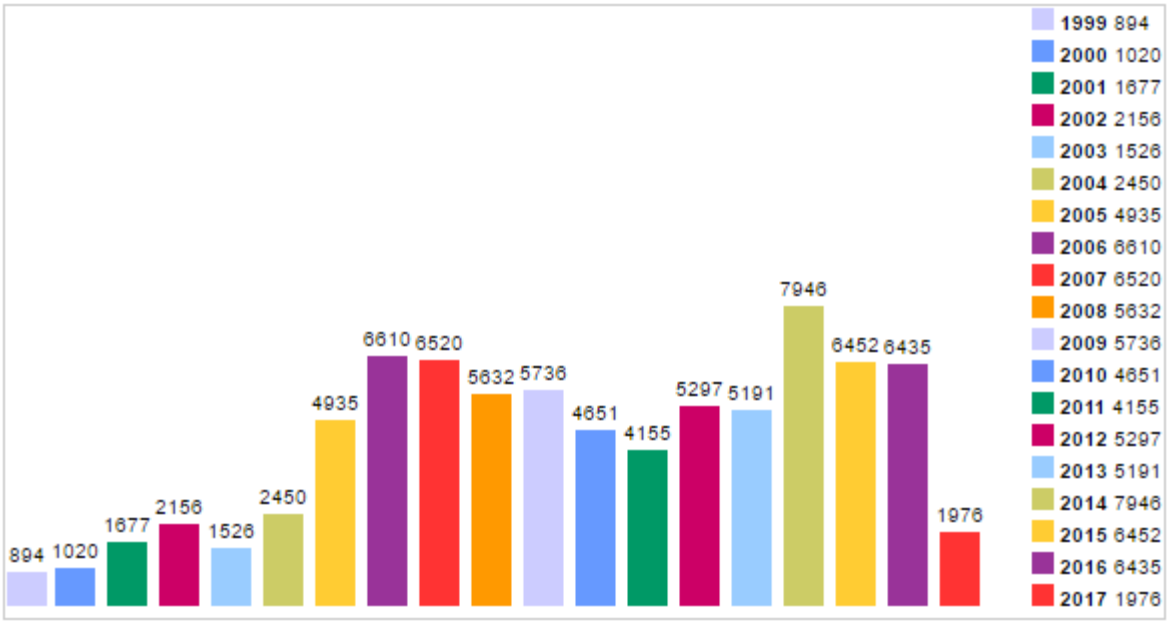
Contrapondo o autor Kiyoshi, os engenheiros de segurança e comunicações da Microsoft, Lipner e Howard (2005), acreditam que existam três aspectos na criação de um *software* mais seguro: o processo repetitivo, o treinamento de engenheiro e as métricas e responsabilidades.

Em outro aspecto, confirmando a visão do autor Kiyoshi (2012), os engenheiros afirmam que, a chave para suprir a demanda atual de segurança, é implementar processos repetitivos que forneçam de forma confiável, uma segurança aprimorada que seja possível de mensurar, para que se torne mais fácil sua quantificação e qualificação, de maneira a minimizar o número de vulnerabilidades existentes no design, na codificação e na documentação.

Diferente de outros recursos, não é possível garantir que uma aplicação tenha ou não segurança, pois, assim como, novas ferramentas ou tecnologias surjam constantemente para aprimorá-la, por outro lado, novas vulnerabilidades são introduzidas no mercado diariamente, capazes de atingir milhares de aplicações simultaneamente.

O gráfico apresentado na Figura 1 ilustra a situação da quantidade de vulnerabilidades documentadas pela CVE desde 1999:

Figura 1: Vulnerabilidades por ano.

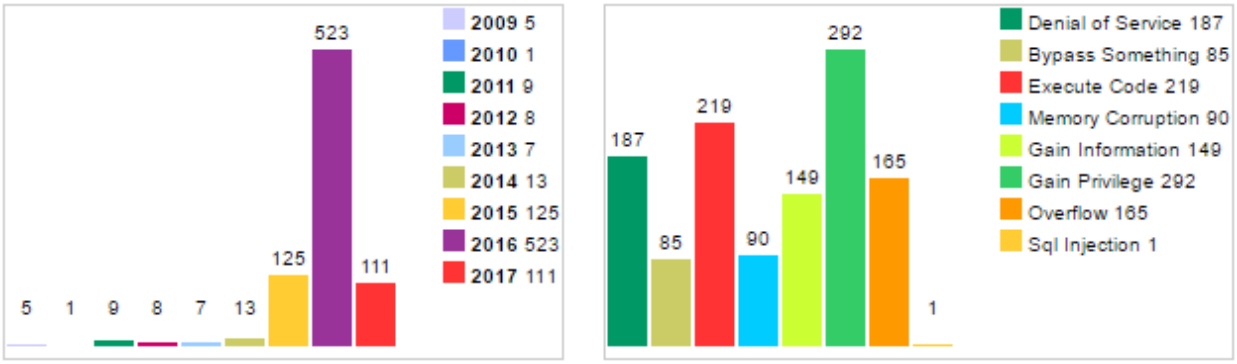


Fonte: CVE (2017)

Como é possível observar, a quantidade de vulnerabilidades descobertas a cada ano passa dos milhares, sendo que, independente da data da descoberta, é necessário um cuidado especial, pois, muitas delas, dependem do desenvolvimento seguro da aplicação, para que, antigas falhas não estejam presentes em softwares atuais. Dessa maneira, conclui-se que os números apresentados são acumulativos.

Dentre as 1976 vulnerabilidades já descobertas desde o início de 2017, até o dia desta pesquisa (21 de fevereiro de 2017), 5.61% delas são apenas do sistema Android, como pode ser visto na Figura 2, a qual apresenta dois (2) gráficos, sendo a quantidade de vulnerabilidades documentadas pela CVE desde 2009 e a quantidade de vulnerabilidades classificadas pelos seus tipos, como, por exemplo *Denial of Service* e *Sql Injection*:

Figura 2: Vulnerabilidades por ano & tipo.



Fonte: CVE (2017)

Independentemente da plataforma vulnerável, a importância do desenvolvimento seguro é crucial, pois, uma aplicação móvel que se conecta com a Internet para realizar operações, fica suscetível a ataques do lado cliente, como também do servidor. Ambas as partes podem contribuir para um ataque bem-sucedido.

3.1. RISCOS EM APLICAÇÕES MÓVEIS

Segundo Leavitt (2011), presidente da Leavitt Communications, em um mundo de comunicações, quanto mais uma tecnologia é usada, mais provável que ela se torne alvo dos *hackers*. E este é o caso da tecnologia móvel, mais especificamente os smartphones.

Levando isto em conta, o crescente número de dispositivos sendo utilizados mundialmente, torna a maioria das informações mais suscetíveis a ataques, pois, muitos usuários baixam aplicativos sem levar em conta se estes estão seguros ou não, fornecendo uma maneira para os *hackers* atacarem o dispositivo facilmente.

Com os aparelhos móveis sendo utilizados frequentemente para transações sensíveis, como, pagamentos, transferências de informações organizacionais e, até mesmo acesso às contas bancárias, os dispositivos se tornam muito mais atraentes

para os criminosos. O fato de muitas aplicações se conectarem à Internet, torna os ataques muito mais convenientes de serem executados.

O diretor de segurança, pesquisa e comunicação da McAffe, Dave Marcus, ressalta que, em uma rotina diária, a equipe da McAffe toma nota de mais de 55,000 novas peças de *malware* de dispositivos móveis.

Considerando o problema iminente, é necessário ter conhecimento das vulnerabilidades exploradas em aplicações móveis, para que seja plausível seu desenvolvimento seguro. A comunidade “Projeto Aberto de Segurança em Aplicações Web” ou OWASP as categoriza em dez (10) Riscos, numerados de um (1) a dez (10). Apresenta-se a seguir o nome e a breve explicação de cada um deles, baseando-se no estudo realizado em 2014.

- **Comunicação fraca com o servidor:** Vulnerabilidades de comunicação fraca, geralmente ocorrem quando um servidor disponibiliza uma *API* para consumo, que é implementada de maneira insegura.

  Através do dispositivo utilizado para o consumo da *API*, um atacante pode manipular a comunicação e realizar consultas, inserções, atualizações, ou deleções de dados no servidor de maneira maliciosa.

- **Armazenamento inseguro de dados:** A armazenagem insegura de dados, geralmente ocorre quando, os desenvolvedores possuem a errônea mentalidade de que, um *malware* nunca terá acesso ao sistema de arquivos do aparelho. Porém, sistemas de arquivos são facilmente acessados e, as organizações devem esperar que os *malwares* consigam inspecionar quaisquer locais de armazenamento, pois, a grande maioria, explora vulnerabilidades do sistema a fim da liberação de acesso *Root*.

  O usuário *Root* de um dispositivo, possui acesso a qualquer arquivo do sistema, independentemente de sua localização, assim, quando os dados não são devidamente protegidos, estarão sujeitos a manipulação.

- **Proteção insuficiente na camada de transporte:** A falta de proteção na camada de transporte, está relacionada, por exemplo, à maneira que um aplicativo se comunica com o servidor. Toda a informação compartilhada entre os dois pontos, está suscetível a monitoramento. Devido a isso, a importância de não compartilhar informações em texto claro, deve ser levada em conta.

  As aplicações móveis frequentemente deixam de lado a proteção no tráfego da rede, embora possam usufruir de SSL/TLS, ou, até mesmo, de quaisquer criptografias, para ao menos ofuscar as informações em caso de monitoramento ou interceptação.

- **Vazamento de dados indesejados:** Vazamentos de dados, geralmente ocorrem quando, o desenvolvedor inadvertidamente armazena-os, em lugares nos quais o dispositivo possui um fácil acesso, seja através de outros aplicativos, ou mesmo manualmente.

  É fácil a detecção de tal risco, basta apenas inspecionar todas as localizações públicas do dispositivo, ou seja, que podem ser acessadas por quaisquer aplicativos, em busca de informações sensíveis.

- **Fraca autorização e autenticação:** A fraca autorização e autenticação, está presente exclusivamente, pela falta da implementação de checagem de usuários e proteção de seus dados de acesso.

  Uma fraca, ou, uma autenticação não presente, pode permitir que um usuário mal-intencionado execute funcionalidades, por exemplo, no *webservice* privado da aplicação.

  Em aplicativos móveis, um sistema de autenticação irá gerar diferentes ramificações as quais devem ser consideradas em tempo de desenvolvimento, como a autenticação *off-line* e a sincronização, devido ao fato que, autenticações em dispositivos móveis diferenciam-se de aplicações web, na qual, possui o usuário conectado à Internet todo o tempo.

- **Criptografia quebrada:** A exploração de uma criptografia quebrada é referente a sua reversão. Uma vez que um atacante consegue reverter uma criptografia, ele tem acesso aos dados em texto claro.

  Existem duas (2) maneiras na qual uma criptografia pode ser “quebrada”, a primeira, é a presença de vulnerabilidades em código, já conhecidas ou não, que permitam que um atacante possa explorá-las através de *exploits* (scripts maliciosos específicos para explorar alguma vulnerabilidade). A segunda, é a força bruta, na qual o atacante descobre a informação oculta, através de tentativa e erro, porém, esta é uma técnica muito custosa em questão de tempo e recursos.

- **Injeções no lado cliente:** Injeções do lado cliente, são resultados da execução de códigos maliciosos através do aplicativo. Geralmente, o dado é manipulado, não tratado, e acaba sendo processado pelo servidor.

  Na melhor das hipóteses, o código é executado dentro do escopo de acesso e permissão que o usuário mal-intencionado possui, mas, na pior, o código é executado com permissões privilegiadas, causando um impacto muito mais severo.

- **Decisões de segurança baseadas em *inputs* inseguros:** Os *inputs* estão presentes em todas as aplicações, uma URL onde exista um ID de uma notícia, por exemplo, é considerado um *input*; os campos de um formulário de cadastro, também são considerados e, até mesmo os campos escondidos entram nesta classificação.

  A partir do momento, que um *input* aceita dados de um usuário do sistema, ele está suscetível a ser uma superfície de ataque.

  Basear-se exclusivamente em tais *inputs* para uma decisão de segurança, gera um grande risco, visto que, na falta de uma filtragem, um *hacker* pode interceptar o envio de formulários ao servidor, manipular os dados, e executar instruções maliciosas remotamente, adquirir permissões mais altas das que foram atribuídas a ele, enviar arquivos maliciosos, ou, até mesmo, executar comandos diretamente no servidor.

- **Tratamento impróprio de sessões:** Em questão de facilitar transações através de aplicativos móveis e servidores, muitas arquiteturas utilizam chaves de sessão para garantir a conexão em chamadas assíncronas em protocolos como *HTTP* e *SOAP*. O processo para tal comportamento, é a notificação de uma chave aleatória de sessão, à qual será utilizada nas futuras transações entre os dois pontos. Isso garante que o aplicativo informe que está autenticado como um certo usuário.

  O risco em tal tratamento de sessão ocorre quando a chave é compartilhada involuntariamente, possibilitando que o atacante a use para forjar sua autenticação.

- **Falta de proteção binária:** A proteção binária em um aplicativo, garante a segurança do código, quanto a do cliente, a uma variedade de riscos corporativos e exposição de propriedades intelectuais. A falta da proteção binária resulta em um software, que possa ser analisado, revertido por engenharia reversa, e modificado por um atacante, interferindo em sua integridade.

Com os dez (10) riscos apresentados e, baseando-se em dados coletados da comunidade Projeto Aberto de Segurança em Aplicações Web (OWASP), foi produzida a Tabela 1 de análise de riscos, enfatizando o nível de exploração, a prevalência, a detectabilidade e o impacto do risco.

Nota-se uma presença considerável de riscos com impactos altos, ressaltando a importância da segurança em aplicações móveis, pois, muitas vezes, a falta de controle sob o dispositivo pode trazer um risco alto para as corporações.

Tabela 1: Análise de riscos a partir de dados da OWASP.

Descrição	Exploração	Prevalência	Detectabilidade	Impacto
Comunicação fraca com Servidor	Fácil	Comum	Média	Severo
Armazenamento inseguro de dados	Fácil	Comum	Fácil	Severo
Proteção Insuficiente na camada de transporte	Difícil	Comum	Fácil	Moderado
Vazamento de dados indesejados	Fácil	Comum	Fácil	Severo
Fraca autorização e autenticação	Fácil	Comum	Fácil	Severo
Criptografia quebrada	Fácil	Comum	Fácil	Severo
Injeções no lado cliente	Fácil	Comum	Fácil	Moderado
Decisões de segurança baseados em inputs inseguros	Fácil	Comum	Fácil	Severo
Tratamento impróprio de sessões	Fácil	Comum	Fácil	Severo
Falta de proteção binária	Media	Comum	Fácil	Severo

Fonte: Próprio autor

3.2. DIRETRIZES DE CODIFICAÇÃO SEGURA

A fim de mitigar ou minimizar impactos dos riscos supracitados acima, é necessário um conhecimento de diretrizes que aprimorem a segurança das aplicações. Segundo a comunidade “Projeto Aberto de Segurança em Aplicações Web” ou OWASP as diretrizes se dividem em oito (8) sessões, classificando-as entre diversos controles.

Numerados de um (1) a oito (8), apresenta-se a seguir, uma lista com os nomes das sessões de um estudo realizado em 2016. A seleção das julgadas mais importantes, foi feita pelo próprio autor desta pesquisa:

- **Autenticação e gerenciamento de senhas:** A autenticação e gerenciamento de senhas é, a garantia da identidade de um usuário interagindo com o sistema.

  A Limpeza dos endereços de memória contendo as senhas logo após o *hash* desses valores serem calculados; a utilização de *salt* de senhas sempre que possível; a informação ao usuário do nível de segurança de sua senha no momento do registro; a garantia de que cada usuário utilize um *salt* único; todas são diretrizes que devem ser considerados para o aprimoramento do gerenciamento das senhas e autenticações dos usuários.

- **Ofuscação de código:** “Ofuscação de código é o método mais viável para prevenir engenharia reversa. ” (HADA, 2000, p. 444, tradução próprio autor).

  A técnica de ofuscar, pode ser realizada, através de, por exemplo, implementação de técnicas *anti-debugging* e desativação de *logs*, permitindo que um aplicativo esteja, em partes, prevenido contra engenharia-reversa, ou que, ao menos, aumente o nível de habilidade requerida de um atacante.

- **Comunicação segura:** Comunicação segura é, a garantia que o aplicativo possui de que está enviando e recebendo informações de uma maneira segura. O que garante tal segurança, pode ser resumido no, uso de certificados confiáveis quando utilizado SSL/TLS, na aceitação de dados apenas de uma comunicação autorizada e, na utilização de chaves de autenticação somente geradas pelo servidor.

- **Armazenamento de dados e proteção:** Devido ao fato dos aplicativos serem executados em dispositivos móveis, existe uma grande probabilidade de perda, ou roubo, com isso, é necessário considerar diversos fatores na hora do armazenamento e proteção de dados do usuário, como, armazenar dados sensíveis somente no servidor, não guardar informações temporárias em locais de fácil acesso no sistema, remover as chaves de criptografia da memória *RAM* durante o ciclo de vida do aplicativo e, **nunca** salvar senhas em texto claro.

- **Controles do servidor:** Independente de um aplicativo ser executado em dispositivos móveis, o servidor que se comunica com o mesmo, necessita de segurança também, pois, o atacante pode aproveitar da comunicação garantida entre os dois, para explorar falhas remotamente.

  Garantir que o servidor passe por um processo de *hardening,* o qual realize o mapeamento e mitigação das ameaças, atualizando e corrigindo serviços utilizados; assegurar-se que os *logs* do sistema estão devidamente configurados e funcionando; limitar a largura de banda utilizada pelos clientes e, rejeitar quaisquer transações ou comunicações que não apresentem os pré-requisitos, como chaves de segurança, podem aprimorar a segurança do servidor.

- **Gerenciamento de sessões:** Para a garantia de que a aplicação gerencie as sessões de maneira correta, é importante verificar o estado da sessão em cada tela do aplicativo para se certificar que o usuário esteja autenticado no sistema, descartar e limpar toda a memória associada aos dados do usuário, assim que a

sessão for terminada e fazer com que as chaves de autenticação sejam revogáveis no lado do servidor.

- **Uso de bibliotecas ou códigos de terceiros:** O uso de bibliotecas ou códigos de terceiros é comum na área de desenvolvimento, pois, aceleram e auxiliam em tarefas mais complexas as quais o *software* necessite realizar, reduzindo consideravelmente o tempo de desenvolvimento.

  Para garantia de segurança na integração de códigos externos, é necessário a verificação crucial de que o código vem de uma fonte confiável, existe um suporte contínuo para tal e, não contenha nenhum tipo de *backdoor*.

  Acompanhe todos os *frameworks,* bibliotecas e *API's* utilizadas, de maneira a tomar conhecimento de quaisquer atualizações existentes e, garantir a validação de todos os dados provenientes de tais códigos.

4. CICLO DE VIDA DO DESENVOLVIMENTO SEGURO (SDL)

O SDL ou “Ciclo de vida do desenvolvimento seguro”, envolve modificar os processos de desenvolvimento de um *software* de maneira a aprimorar a segurança do produto final, não revisando totalmente o processo, mas adicionando pontos importantes de se considerar.

De acordo com Lipner e Howard (2005), como já exposto, a chave para suprir a demanda atual de segurança, é implementar processos repetitivos que forneçam de forma confiável, segurança aprimorada mensurável. Dessa maneira, os fornecedores de *software,* devem fazer a transição para um processo de desenvolvimento mais rigoroso que se concentre, de forma mais ampla, na segurança. Esse processo tem o objetivo de minimizar o número de vulnerabilidades de segurança existentes no design, na codificação e na documentação, além de detectar e remover essas vulnerabilidades o mais cedo possível, durante o ciclo de vida do desenvolvimento.

4.1. SD3+C

O SDL orienta-se em quatro (4) princípios, levando em consideração a segurança desde o design e o padrão para a compilação do *software*, até a implantação e a comunicação, para auxílio em determinar onde é necessário a preocupação com a segurança.

- **Seguro por design:** Procura garantir que, a arquitetura, o design e, a estrutura do aplicativo sejam executadas de maneira a proteger quaisquer informações que ele processe e, resista a ataques.

- **Seguro por padrão:** Na segurança por padrão, encontra-se a conscientização de que nenhum *software* é totalmente seguro e, deve se considerar que o aplicativo apresente falhas, de maneira que, os desenvolvedores minimizem os

possíveis danos de exploração de alguma falha existente, por exemplo, limitando privilégios de uso.

- **Seguro na implantação:** Garantir que o aplicativo contenha ferramentas e, orientações que auxiliem os usuários finais a utiliza-lo com segurança, além de facilitar uma possível atualização.

- **Comunicações:** O princípio da comunicação, se deve ao fato de que, os desenvolvedores e administradores do *software*, devem comunicar-se de maneira aberta com os usuários finais em caso de descoberta de vulnerabilidades, para garantir que, executem ou instalem *patches* de correções.

Conforme Lipner e Howard (2005) enfatizam, todos os elementos do SD3+C impõem requisitos para um desenvolvimento seguro, porém, os dois primeiros, seguro por design e, seguro por padrão, fornecem as maiores vantagens, pois, têm o objetivo de impedir a introdução de vulnerabilidades e, minimizar a superfície do ataque, respectivamente.

4.2. PD3+C

Em analogia com o SD3+C, o qual preocupa-se com a segurança, o PD3+C também conhecido como SD3+C para privacidade, é igualmente dividido em quatro (4) princípios, preocupando-se com a privacidade do *software*.

- **Privacidade por design:** Procura fornecer de maneira apropriada, avisos sobre dados coletados, armazenados ou compartilhados, para que o usuário possa tomar suas decisões.

  Além de coletar o mínimo necessário de dados sensíveis para uma particular necessidade e, proteger o armazenamento e a transferência de dados

através de criptografias, o controle de usuário, a fim de gerenciar privacidade de uso caso desejado, também se engloba nesta categoria.

- **Privacidade por padrão:** Na privacidade por padrão, leva-se em conta o envio do aplicativo com configurações padrões conservadores, que obtenha o consentimento do usuário antes da coleta ou transferência de qualquer dado.

  Para evitar acessos não autorizado, recomenda-se a proteção dos dados pessoais armazenados nas listas de controle de acesso.

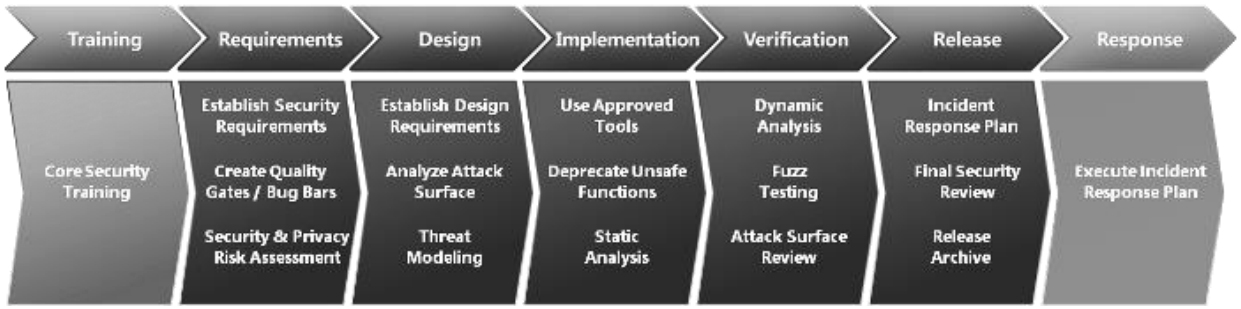
- **Privacidade na implantação:** Procura garantir mecanismos de privacidade para o usuário, para que possam estabelecer e, manter políticas de privacidade de suas informações.

- **Comunicações:** A comunicação na sessão de privacidade, auxilia na transparência, publicação e, no estabelecimento de uma equipe de resposta a privacidade, publicando declarações em locais apropriados, envolvendo os meios de comunicação convencionais como White Papers e documentações para orientação.

4.3. O PROCESSO DO SDL

Com base no pensamento de Lipner e Howard (2005), o ciclo de desenvolvimento seguro é dividido em sete (7) fases de implementação, ilustradas na Figura 3.

Figura 3: O processo de desenvolvimento padrão da Microsoft.



Fonte: Microsoft ([s.d.]).

- **Fase de treinamento (Pré-SDL):** No início do processo SDL, é necessário que todos os membros envolvidos no desenvolvimento, recebam um treinamento apropriado de maneira a se informar sobre os conceitos básicos de segurança e, as atuais tendências no ramo.

  É recomendado que os desenvolvedores participem de pelo menos um treinamento de segurança todo ano e, que sejam encorajados a buscar uma educação complementar sobre segurança e privacidade.

  Um estudo completo deve envolver conceitos básicos e avançados, requisitos e recomendações de segurança, recomendações de privacidade e recursos, todos, envolvendo redução de superfície de ataque, modelagem de ameaças, vulnerabilidades técnicas, metodologias de testes, análise de riscos entre muitos outros.

- **Fase de requisitos:** Durante a fase de requisitos, é necessário a designação de um supervisor de segurança, o qual servirá como contato e orientação durante o planejamento.

  Aqui é necessário que todo o plano seja revisado, para que se crie recomendações e, garanta que a equipe de segurança planeje os recursos apropriados.

  O supervisor de segurança é encarregado do aconselhamento sobre marcos de segurança e critérios de saída que serão exigidos.

  Nesta fase se considera como a segurança será integrada no processo de desenvolvimento do *software*, identificando os objetivos-chave para maximizar a segurança e, minimizar a quebra de cronograma. Deve-se também considerar, como será a integração dos recursos de segurança, atrelados a outros *softwares* que, provavelmente serão utilizados no decorrer do desenvolvimento. Mesmo estando sujeito a alterações, é necessário que a perspectiva da equipe sobre os objetivos, desafios e planos de segurança, reflitam nos documentos produzidos.

  Embora alguns requisitos serem identificados na modelagem das ameaças, aconselha-se que a articulação detalhada dos planos seja produzida

precocemente, para garantir que nenhum seja desconsiderado, ou descoberto em última hora.

- **Fase de design:** Na fase de design são identificados a estrutura e os requisitos gerais do *software,* em uma perspectiva de segurança, através de:

  - Definição das diretivas de design e arquitetura de segurança, de maneira geral, sempre identificando quais componentes são essenciais para a segurança. Aplicação de privilégios mínimos para minimizar a superfície de ataque, uma boa organização em camadas, de maneira que, componentes em uma camada mais alta, podem depender de serviços de camadas inferiores, mas nunca o contrário.
  - Documentação dos elementos da superfície de ataque do *software*, por exemplo, documentar como o aplicativo não atinge a segurança perfeita
  - Realização da modelagem de ameaças componente por componente, identificando por quais interfaces os ativos são acessados e, qual a probabilidade de ocorrerem danos, sempre estimando o risco. Após a identificação, é necessário elaborar as contramedidas que diminuam a intensidade do risco. Desta maneira, existe um auxilio com a área de testes de segurança e revisão de código, afim de capturar as possíveis ameaças.
  - Definir critérios de fornecimento complementar, os quais devem ser atendidos antes do lançamento do aplicativo. O processo de desenvolvimento, deve localizar e remover vulnerabilidades antes que sejam relatadas, evitando que a equipe as corrija depois e acabe gerando diversos patches de atualizações para os usuários.

- **Fase de implementação:** Na fase de implementação, realiza-se um teste e uma integração no aplicativo, para remover ou evitar falhas de segurança, auxiliando na redução da probabilidade de que existam vulnerabilidades em versões finais lançadas para os clientes.

  Com base na modelagem de ameaças, realizada na fase de design, os desenvolvedores se dedicam especialmente para correção de código, dando prioridade para as ameaças de alto risco, juntamente com a área de testes, que deve garantir que tais ameaças foram bloqueadas. Existem quatro (4) elementos aplicados nesta fase:

  - Padrões de codificação e testes, que ajudem a evitar introdução de falhas que possam gerar vulnerabilidades e, garantir que os testes detectem falhas e não apenas que funções e recursos do *software* funcionem de maneira correta.
  - Ferramentas de testes de segurança, incluindo difusão, as quais ofereçam entradas inválidas para o *software*, para maximização da probabilidade de detecção de erros que possam gerar vulnerabilidades.
  - Ferramenta de análise de código para detecção falhas de código que resultem em vulnerabilidades, por exemplo, saturação de *buffer* de números inteiros e variáveis não inicializadas.
  - Revisão de código que complemente os testes de segurança, realizada através de desenvolvedores treinados no exame do código, para a possível detecção de vulnerabilidades.

- **Fase de verificação:** Nesta fase é onde o aplicativo já está concluído e, entra na etapa de testes por usuários. Durante estes testes, a equipe realiza um esforço de segurança para revisão do código, além da que já foi feita na fase de implementação, garantindo que esta tenha como objetivo a versão final do *software* e, tenha a oportunidade de revisar códigos que foram introduzidos na fase anterior.

- **Fase de lançamento:** A fase de lançamento, é quando seu aplicativo já está pronto para o consumo dos usuários e, os desenvolvedores já estão prontos para o que acontecer. Um dos principais conceitos desta fase, é o planejamento, mapeando um plano de ação, juntamente com uma revisão final da segurança e da privacidade.

- **Resposta (Pós-SDL):** A fase de resposta, garante que, após o lançamento do *software*, a equipe de desenvolvedores, esteja disponível para responder a quaisquer problemas de segurança ou privacidade que demandem uma resposta. É recomendado também, que se desenvolva um plano de respostas que inclua as preparações para um potencial *patch* pós-lançamento.

5. ESTUDO DE CASO

Para fins de avaliação da importância do desenvolvimento seguro, técnicas de programação foram aplicadas na criação de um aplicativo Android denominado “EquinoGestor Móvel off-line” para um sistema web existente denominado EquinoGestor®, destinado ao gerenciamento dos processos de criação e manejo de equinos, criado pela empresa AgroRoxa® Software para o Agronegócio. A proposta do aplicativo “off-line” é permitir que o usuário possa realizar os registros de procedimentos veterinários e de reprodução através do celular, independentemente de estar ou não conectado à Internet. Posteriormente, quando houver conexão com a Internet, o aplicativo realiza a sincronização automática dos dados criptografados entre o dispositivo móvel e o servidor web, garantindo assim a disponibilidade das informações em todas as plataformas.

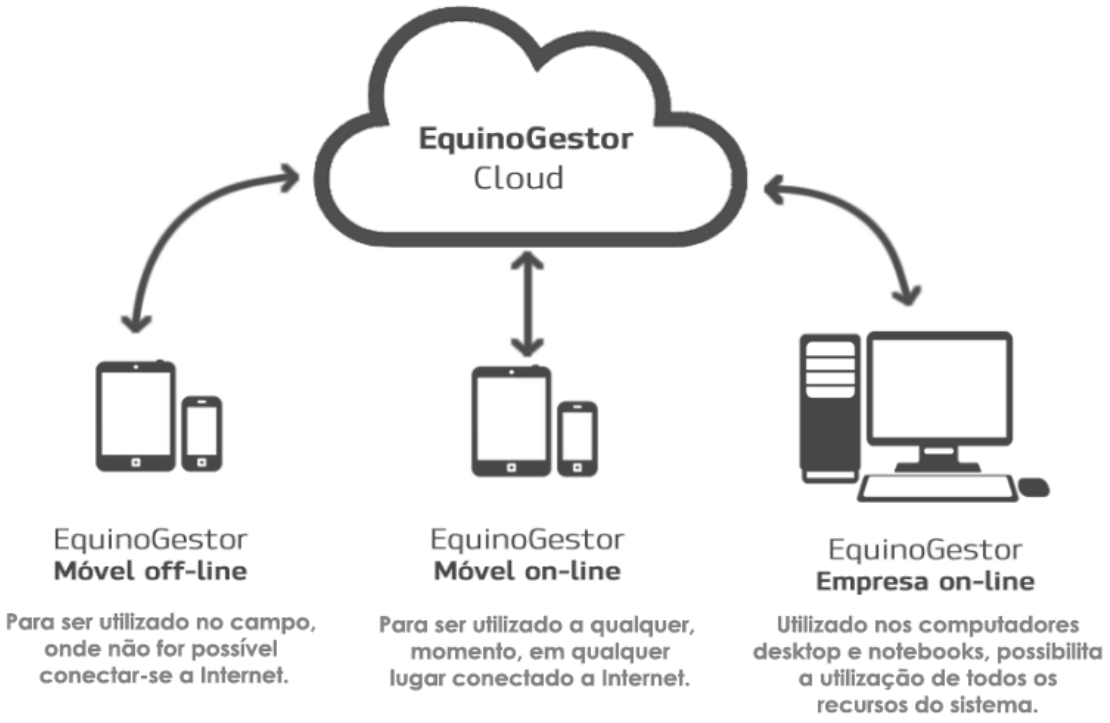


Figura 4: Utilização do EquinoGestor.

Fonte: Documentação EquinoGestor App

Os três tipos de utilização do sistema, garantem que o usuário consiga gerenciar seus dados através do celular e do navegador. No celular, independe da presença da Internet, devido ao sistema de sincronização implementado.

Percebeu-se a necessidade de tal funcionalidade, devido ao grande número de clientes que não possuem sinal de Internet em seus haras, por serem em locais remotos.

A sincronização dos dados e toda a comunicação com o servidor, ocorre através de um *webservice,* desenvolvido especificamente para esta aplicação, utilizando HTTP como meio de transporte dos dados serializados em JSON. Uma mensagem de comunicação segue o padrão:

```
{
        “Mensagem” :
        {
                “Sucesso” : true,
                “Codigo” : 10,
                “Mensagem” : “Mensagem relevante”,
                “DataHora” : “Data e hora da mensagem”
        },
        Dados relevantes ...
}
```

A “Mensagem” é um campo comum entre todas as transações, na qual o dispositivo pode distinguir se houve algum erro no processamento dos dados, autenticação, ou qualquer outra comunicação. Os “Dados relevantes”, são quaisquer

tipos de informação solicitadas pelo aplicativo, sejam os dados de sincronização, ou mesmo as informações da autorização de acesso, na autenticação do usuário.

5.1. CONFIDENCIALIDADE

A confidencialidade do aplicativo, se deve ao fato de que toda a sua utilização é protegida por um sistema de *login,* para a garantia de que, apenas usuários **registrados** utilizem o aplicativo e, também, um sistema de *check-in* para assegurar-se que o usuário está **autorizado** a utiliza-lo.

O usuário só pode se conectar ao aplicativo, após enviar suas credenciais de acesso, vale lembrar que é permitido o registro de um novo usuário através do aplicativo.

A autenticação do usuário sempre é executada quando se abre o aplicativo e, uma sessão de acesso não exista. Esta checagem independe da tela em que o usuário está, por que, todas as telas herdam o processo de criação, da chamada *BaseActivity,* a qual contém a seguinte verificação:

```
if(!allowNoLogin)
     {
         UserSession userSession = new UserSession(this);
         if (!userSession.IsLoggedIn())
         {
             userSession.SendToLoginActivity();
             Finish();
         }
     }
 }
```

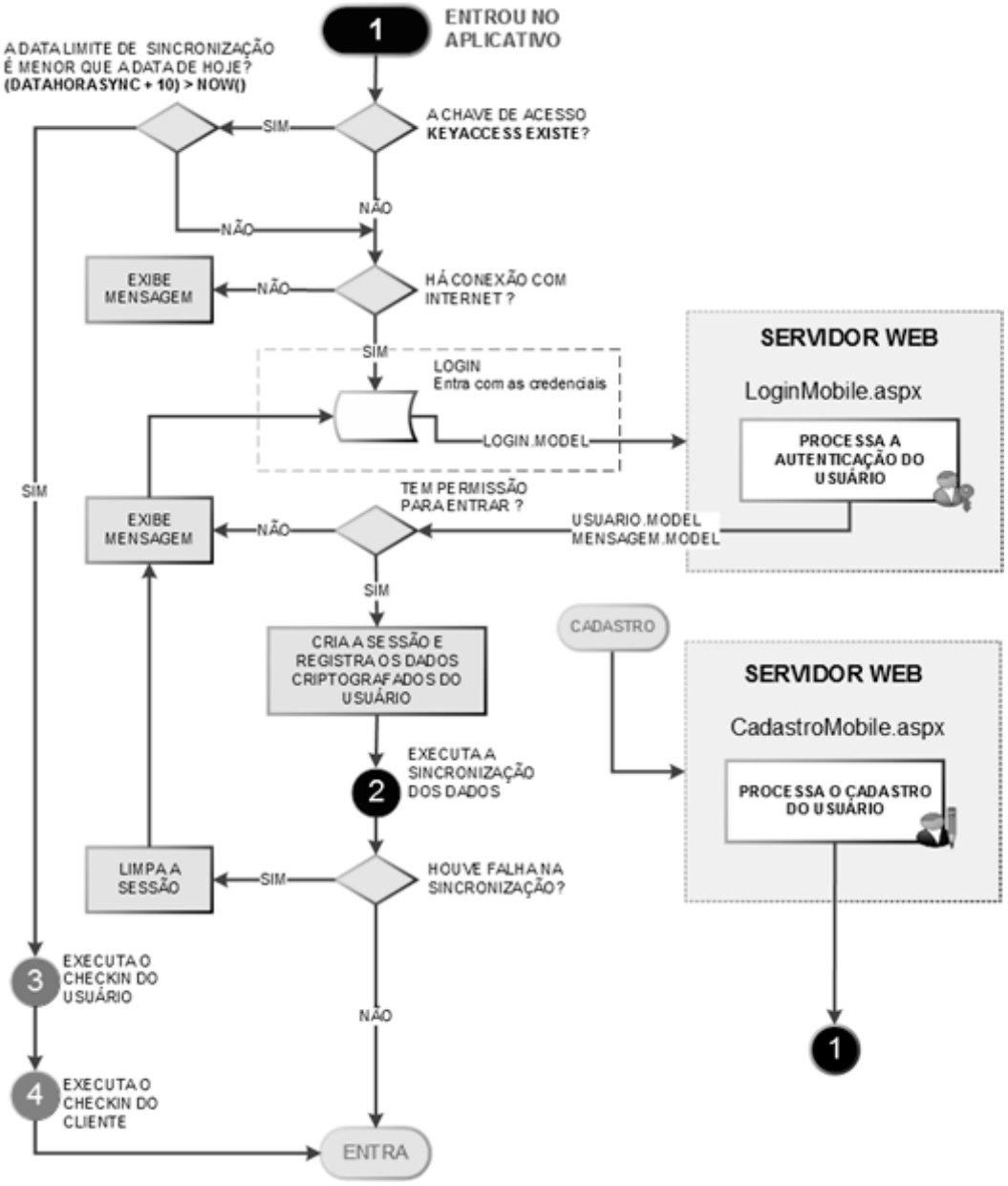
Primeiramente, é verificado se a página permite acesso de um usuário não autenticado, caso não permita, uma instância de *UserSession* é criada para a verificação de que o mesmo está autenticado ou não, o redirecionamento para a página de *login* é executado caso os requisitos não sejam atendidos e, a tela atual é finalizada.

Quando o usuário já se encontra conectado ao aplicativo e, uma sessão de acesso já está registrada em seu dispositivo, ocorre a validação da data de expiração da sincronização. Desta maneira garante-se que o usuário sincronize seu aplicativo pelo menos, de 10 em 10 dias, prevenindo que ele tente, de alguma maneira, burlar a autorização de seu usuário e, garantir que seus dados continuem atualizados.

Independente da conexão com a Internet, a autorização de acesso ocorre todas as vezes em que o usuário abre o aplicativo, passando pelo processo de *check-in,* seja online, através do servidor ou offline, com base na sessão registrada do usuário.

O fluxograma do processo de autenticação e autorização é definido na Figura 5, 6 e 7.

Figura 5: Inicialização do aplicativo.



Fonte: Documentação EquinoGestor App.

A autenticação do usuário, como já mencionada, ocorre quando não existe a presença de uma sessão de acesso no dispositivo. Após o envio das credenciais, o servidor realiza o processo de autenticação, se bem-sucedida, a sessão de acesso é registrada no dispositivo e, o primeiro processo de sincronização é iniciado. Em todos os passos deste processo, caso haja alguma falha, o usuário é informado e, a sessão é cancelada.

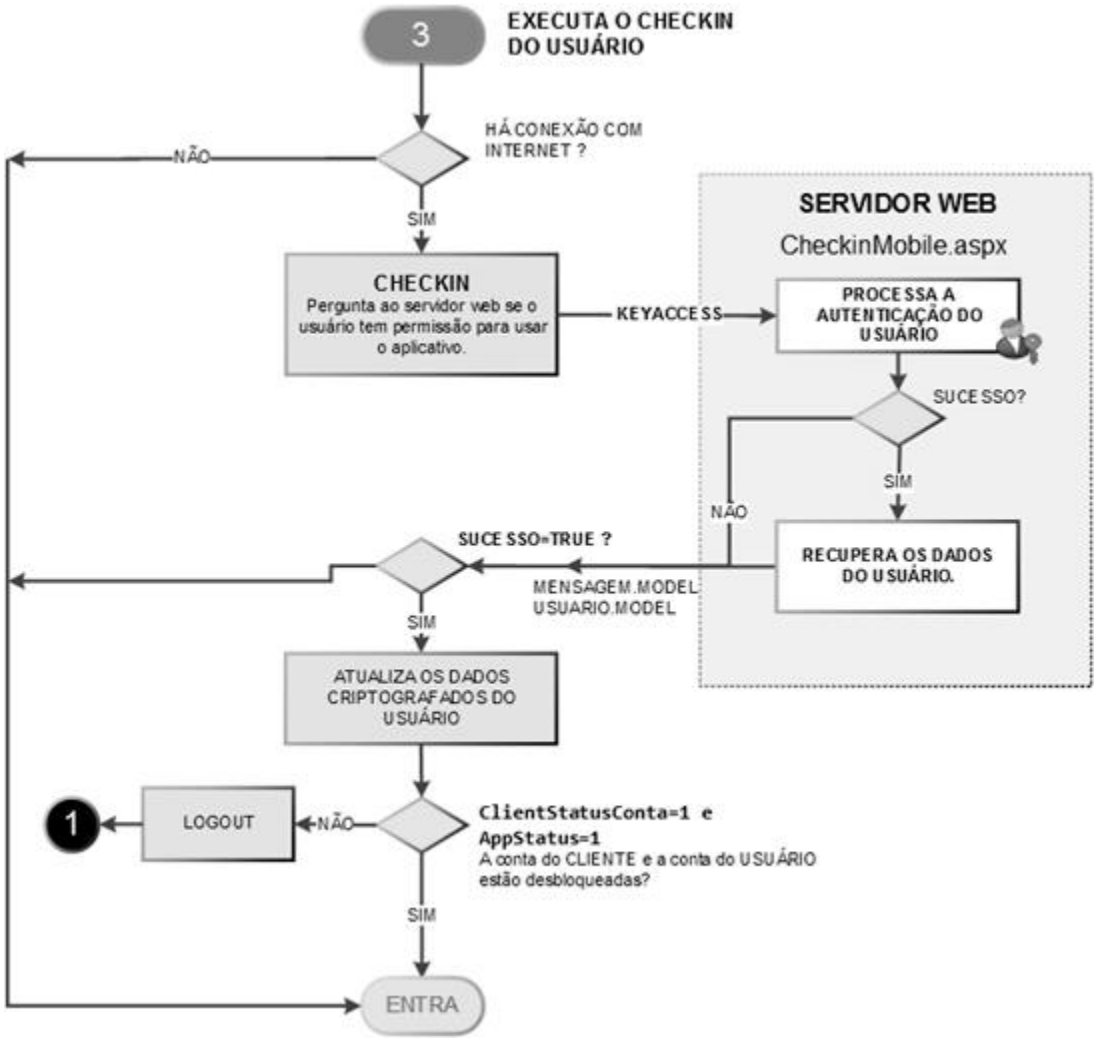
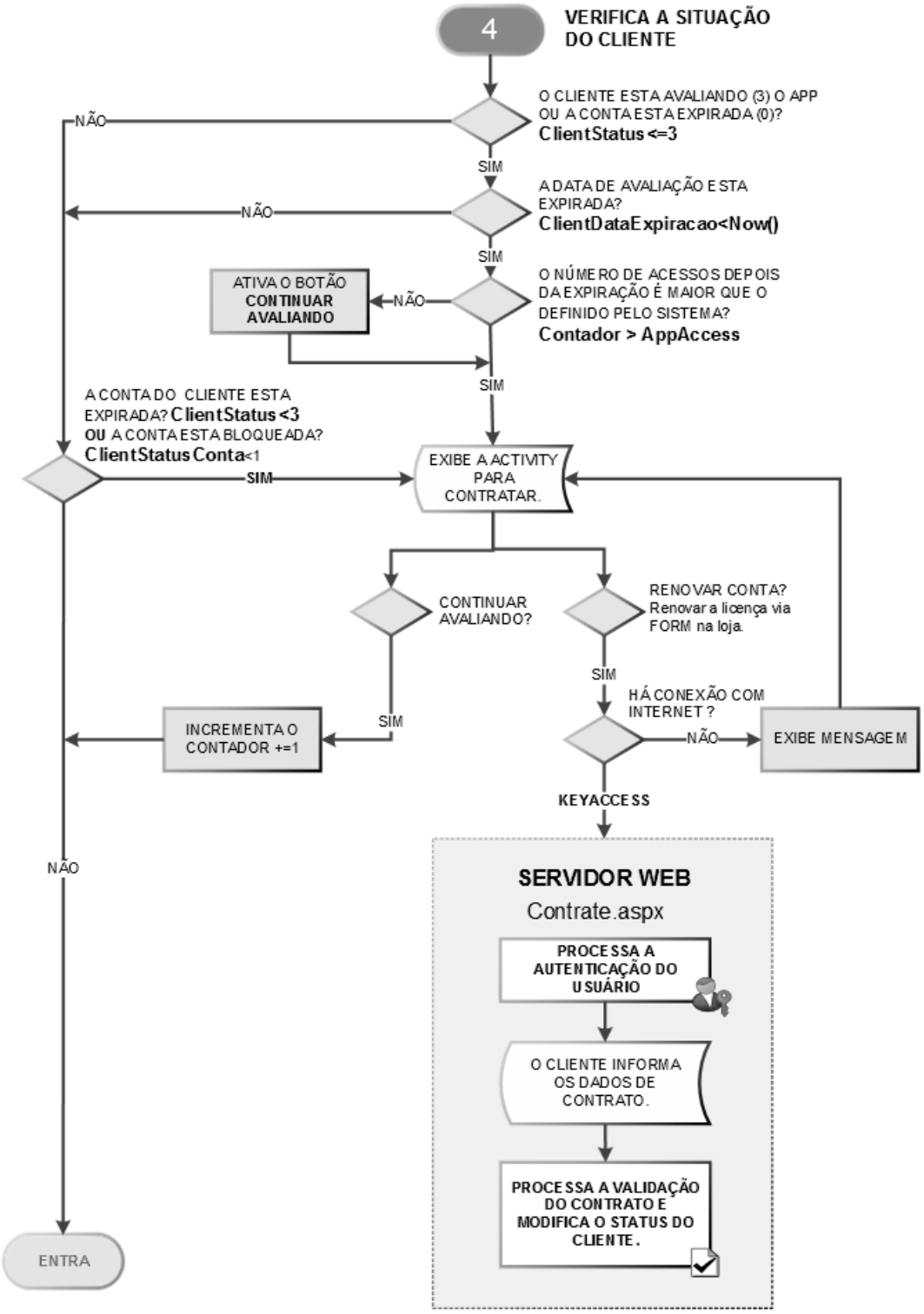


Figura 6: Check-in do usuário.

Fonte: Documentação EquinoGestor App.

Figura 7: Autorização do usuário.



Fonte: Documentação EquinoGestor App.

O processo de *check-in*, tem início na verificação da internet para a atualização da sessão do usuário de acordo com os dados mais atuais registrado no servidor, a fase três (3) do fluxograma, existe apenas para requisição dos dados cadastrais, caso haja conexão com a internet, dessa maneira, atualiza-se os dados locais, para que não haja falta de integridade entre às duas partes. É através desta atualização também, que o usuário realizará o processo de *Logout* do sistema, caso algum dado tenha sido manipulado, adicionando uma fina camada de segurança na questão da alteração dos dados locais.

A validação do usuário, ocorre de fato na fase quatro (4) do fluxograma, onde as verificações particulares do sistema ocorrem, bloqueando o usuário de acesso e, obrigando-o a contratar o sistema, caso sua conta de cliente esteja em modo avaliação e, a data de expiração do acesso de avaliação esteja expirada, assim como o contador de avaliações for maior ou igual que o limite informado pelo servidor.

- **Avaliação das ameaças:** Através do estudo da lógica aplicada para a autenticação e autorização do usuário, foram detectadas as ameaças recorrentes a confidencialidade, ilustradas na Tabela 2.

Tabela 2: Ameaças à confidencialidade.

Ameaça	Vulnerabilidade	Probabilidade	Impacto
Alteração de contas após primeira sincronização	Falta de restrição de dispositivo	Média	Médio
Manipulação maliciosa dos dados de autorização	Falta de criptografia nos dados	Baixa	Alto
Desconectar dispositivo para não atualização dos dados de autorização	Falta de autorização local	Média	Baixo

Fonte: Próprio autor.

As ameaças apresentadas, foram mitigadas através da restrição do aplicativo para apenas uma conta de usuário, atendendo as exigências do sistema web. Após uma autenticação bem-sucedida, o servidor retorna ao aplicativo a chave de acesso criptografada do usuário, a qual será utilizada para todas as transações, com os dados relevantes de seu registro. Dados os quais, são inseridos no banco de dados do dispositivo para a futura verificação caso o usuário desconecte-se da Internet, garantindo que, as credenciais de acesso as quais ele possui, pertencem ao usuário que está registrado para aquele dispositivo.

Todos os dados do usuário, registrados no dispositivo, passam por um processo de ofuscação de código e criptografia. Dificultando o entendimento e, a manipulação por um usuário mal-intencionado.

A falta de controle a respeito da conexão do dispositivo com a Internet, gerou um obstáculo na autorização do usuário, porém, com a sessão de acesso, os dados referentes a um futuro bloqueio, se necessário, foram registrados, ofuscados e criptografados, de maneira que, na ausência da Internet para a comunicação com o servidor, a autorização de acesso, ocorrida ao abrir o aplicativo, leva em conta os dados de controle registrados no dispositivo.

5.2. INTEGRIDADE

A integridade dos dados é garantida, através de identificadores inseridos no registro de um novo dado. Assim como a presença de chaves primárias no banco de dados, os identificadores comuns entre as diferentes tabelas existentes são:

- ClientId – Identificador único do cliente que utiliza o aplicativo;
- Status – Estado do registro, onde um negativo (-1) representa que o dado está deletado, zero (0), dado inativo no sistema, um (1), ativo no sistema e, dois (2), está ativo no sistema, mas sua edição bloqueada. A regra de utilização deste

atributo, é de que, quaisquer dados registrados com um valor menor que um (1), não devem ser exibidos ao usuário;

- DateTimeInsert – Data e hora a qual o dado foi inserido;
- DateTimeUpdate – Data e hora a qual o dado foi atualizado;
- UserId – Código de identificação do usuário utilizando o sistema;
- Sync – Atributo relevante à sincronização, zero (0) representa que o dado não está sincronizado com o servidor, e um (1), está sincronizado.

A chave primária dos registros varia em sua composição, podendo composta ou simples e, também entre as tabelas do banco, com diferentes nomes relevantes para cada uma.

Para todos os dados inseridos ou atualizado no sistema, os atributos devem ser manipulados, de maneira que, todos sejam preenchidos em uma inserção e, apenas DateTimeUpdate, Status e Sync modificados em uma atualização.

- **Avaliação das ameaças:** A ameaça encontrada recorrente à Integridade, afetaria ambos os sistemas e todos os registros inseridos pelo usuário através do aplicativo.

Tabela 3: Ameaças à integridade.

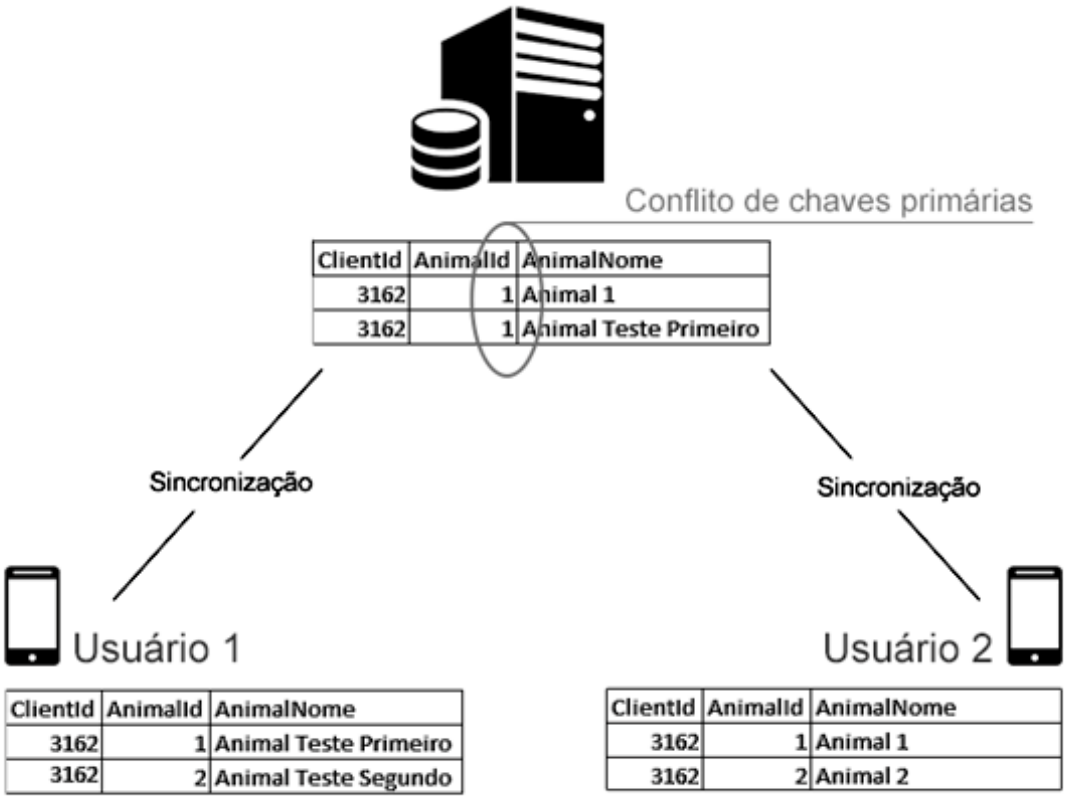
Ameaça	Vulnerabilidade	Probabilidade	Impacto
Conflito de chaves primárias	Utilização do atributo AUTO_INCREMENT nas tabelas do banco de dados	Muito Alta	Alto

Fonte: Próprio autor.

O problema na inserção da chave primária dos registros, afetaria o processo de sincronização. Ao inserir um dado em que sua chave primária contenha o atributo de AUTO_INCREMENT, ocorreria que, se diferentes usuários estivessem utilizando o sistema simultaneamente, gerando novas

chaves a cada inserção, todas, sempre começando do número zero (0) e, se auto incrementando a cada novo registro, geraria um conflito ao sincronizar, pois, ao menos dois (2) usuários enviariam um registro com a mesma chave primária, como ilustrado na Figura 8.

Figura 8: Conflito de chaves primárias.



Fonte: Próprio autor.

Para mitigar o risco apresentado, foi proposto a utilização de um gerador de chaves únicas, de maneira que, diferentes usuários insiram diferentes registros, sempre com chaves primárias exclusivas. A lógica utilizada para a geração do número identificador único, foi construí-lo utilizando o registro de

tempo do dispositivo e, montar uma chave de 40 bits somando, ano, mês, dia, hora, minuto, segundo e milissegundo, multiplicando-os por chaves distintas para que seja mais difícil identificar a data, hora e milissegundo em que a chave foi gerada:

```
public static long GetKey()
 {
     long ano = System.DateTime.Now.Year;
     long mes = System.DateTime.Now.Month;
     long dia = System.DateTime.Now.Day;
     long hor = System.DateTime.Now.Hour;
     long min = System.DateTime.Now.Minute;
     long seg = System.DateTime.Now.Second;
     long mil = System.DateTime.Now.Millisecond;
     long v = ((ano - 2013) * ???) + (mes * ???) + (dia * ???) + (hor * ???) +
(min * ???) + (seg * ???) + mil;
     return v;
 }
```

Os valores apresentados através de três (3) interrogações (?) são as chaves distintas, que foram mascaradas para garantia de segurança. Dessa maneira, ao realizar o processo de sincronização, não haverá conflitos entre as chaves de usuários diferentes, garantindo que os dados manipulados tanto no servidor, como no aplicativo, se mantenham íntegros sempre.

5.3. DISPONIBILIDADE

A garantia de disponibilidade em quaisquer sistemas é crucial, devido ao fato que, o usuário deve ter a opção de acessar seus dados ou informações, a qualquer momento, independente de Internet ou sinais de telefonia. A mobilidade dos aparelhos telefônicos propõe justamente isso, o acesso a suas informações em quaisquer lugares e de maneira fácil. Um aplicativo não deve limitar esta liberdade.

- **Análise de ameaças:** As principais ameaças referentes à disponibilidade, se devem ao fato da ausência da conexão com a Internet, nos locais onde o aplicativo será mais utilizado. Conforme apresentado na Tabela 4.

  Diversos usuários do software, irão usufruir de suas funções em locais remotos, devido aos haras estarem localizados em áreas rurais, onde nem

sempre há disponibilidade de Internet ou sinal de telefonia, visto que os dados manipulados pelo aplicativo estão salvos em nuvem, nos servidores oficiais do EquinoGestor.

Tabela 4: Ameaças à disponibilidade.

Ameaça	Vulnerabilidade	Probabilidade	Impacto
Falta de conexão com a Internet	Falta de registro de dados localmente	Muito Alta	Médio
Conexão com Internet lenta	Falta de registro de dados localmente	Muito Alta	Médio
Indisponibilidade do servidor	Falta de registro de dados localmente	Muito Baixo	Baixo

Fonte: Próprio autor.

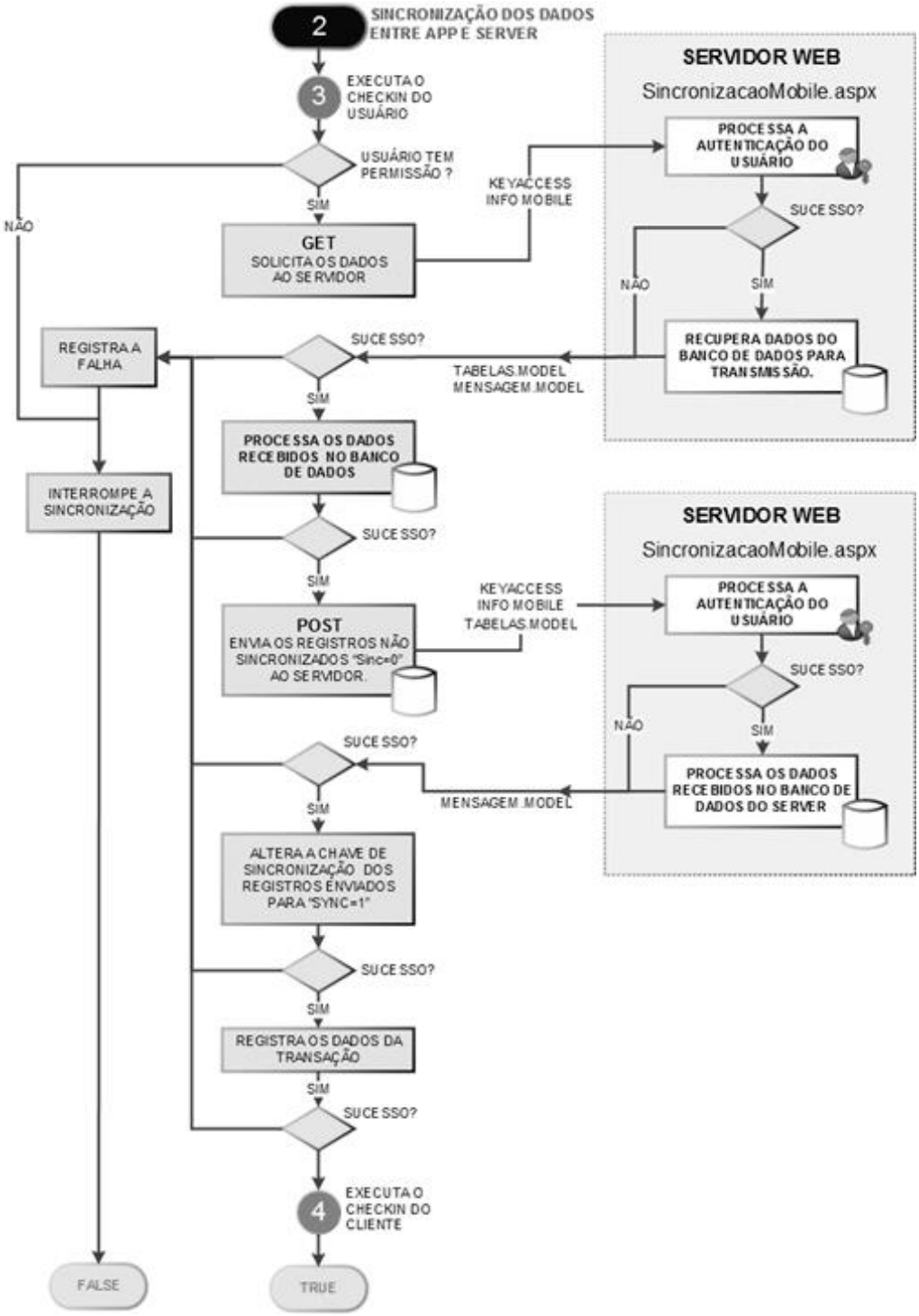
Por estes motivos, se tornou inviável deixar que o usuário possa apenas utilizar o sistema com a presença de Internet ou sinal de telefone.

A solução proposta para as ameaças expostas, foi de que, ao realizar a autenticação pela primeira vez, o usuário é submetido ao processo de sincronização, que estará sempre garantido, por que, a autenticação ocorre apenas com a presença de conectividade.

Nesta sincronização, todos os dados fundamentais para o funcionamento correto do aplicativo são baixados do servidor e guardados no banco de dados local do Android (SQLite).

Durante toda a utilização do aplicativo, o usuário estará registrando os dados localmente, para, posteriormente realizar, o processo de sincronização. O método de sincronização pode ser escolhido pelo usuário da seguinte forma: Wi-fi ou 3G.

Figura 9: Sincronização dos dados.



Fonte: Documentação EquinoGestor App.

Nota-se que, o processo de sincronização, começa realizando o *check-in* do usuário, para que todos os seus dados estejam atualizados para de fato sincronizar.

Na fase de “Processamento dos dados recebidos” ocorre a inserção dos dados remotos no banco de dados local do dispositivo, para que estes fiquem acessíveis mesmo sem a conexão com a Internet, pois, agora todos estão registrados localmente, mitigando as ameaças apresentadas.

5.4. WEBSERVICE

A criação de um *webservice* é crucial na sincronização de dados entre o dispositivo móvel e o servidor, é através dele que são enviadas e lidas, as mensagens e dados referente a comunicação remota.

O desenvolvimento do *webservice* para o sistema EquinoGestor, contou com a presença de quatro (4) *endpoints* (URLs responsáveis pelas comunicações):

- *Endpoint* de *Check-in*: Utilizado para a atualização dos dados de sessão;
- *Endpoint* de *Login*: Utilizado para a autenticação de um usuário;
- *Endpoint* de Cadastro: Utilizado para o registro de novos usuários;
- *Endpoint* de Sincronização: Utilizado para a transferência dos dados;

Toda a comunicação com o *webservice,* se resume em enviar requisições do tipo HTTP *GET* e HTTP *POST* para o servidor através dos *endpoints* existentes, para cada tipo de requisição o comportamento do servidor é diferente, mas sempre transferindo um objeto de Mensagem informando se houve sucesso, ou não, na execução do processo solicitado. Como citado no início do estudo de caso, uma Mensagem contém dados a respeito do Sucesso do procedimento, um Código de controle, a Mensagem propriamente dita e, a Data e Hora do ocorrido.

Requisições GET e POST são métodos de comunicação entre o servidor e um cliente, através do protocolo HTTP. Por exemplo, um navegador de Internet, tem o papel de um cliente que, ao acessar um website, envia uma requisição GET para o

servidor presente na URL, a fim de que o servidor responda enviando o conteúdo do website a ser exibido.

Já uma requisição POST, é mais comum no envio de formulários. Quando um cliente, ao preencher e salvar um formulário de contato, cadastro, login, questionário ou qualquer outro tipo, enviará os dados inseridos para o servidor, através de parâmetros que não estão visíveis na URL como no método GET.

A principal diferença entre requisições GET e POST, é a visibilidade dos dados trafegados. Enquanto uma requisição GET transmite os dados através de parâmetros na URL, por exemplo:

http://www.exemplo.com.br/postagem.php?id=1

A requisição *POST* transmite os dados encapsulando-os junto ao corpo da requisição HTTP, dificultando sua visibilidade.

As requisições HTTP GET e HTTP POST, são executadas pelo aplicativo através de dois métodos:

```
public async Task<string> MakeGetRequest(string url, string cookie)
 {
     HttpWebRequest request = (HttpWebRequest)WebRequest.Create(url);
     request.ContentType = "text/html";
     request.Method = "GET";
     request.Headers["Cookie"] = cookie;

     var response = await request.GetResponseAsync();
     var respStream = response.GetResponseStream();
     respStream.Flush();

     using (StreamReader sr = new StreamReader(respStream))
     {
         string strContent = sr.ReadToEnd();
         respStream = null;
         return strContent;
     }
 }
```

```
        public async Task<string> MakePostRequest(string url, string data, string cookie,
bool isJson = true)
        {
            HttpWebRequest request = (HttpWebRequest)WebRequest.Create(url);
            if (isJson)
                request.ContentType = "application/json";
            else
                request.ContentType = "application/x-www-form-urlencoded";
            request.Method = "POST";
            request.Headers["Cookie"] = cookie;
            var stream = await request.GetRequestStreamAsync();
            using (var writer = new StreamWriter(stream))
            {
                writer.Write(data);
                writer.Flush();
                writer.Dispose();
            }

            var response = await request.GetResponseAsync();
            var respStream = response.GetResponseStream();

            using (StreamReader sr = new StreamReader(respStream))
            {
                return sr.ReadToEnd();
            }
        }
```

Nota-se que, em ambos os métodos, é necessário informar a URL no *endpoint* desejado. Em uma requisição HTTP POST, informa-se também, os dados a serem enviados, já que, serão encapsulados no corpo da requisição, diferente dos dados enviados via HTTP GET, que estarão presentes na própria URL através de parâmetros.

- **Análise de ameaças**: As vulnerabilidades detectadas no *webservice*, se resumem na falta de autenticação que ele possa ter, possibilitando o *download* ou *upload* de dados através dos métodos GET e POST, como pode se observar na Tabela 5.

Embora a utilização de *webservice*, seja uma prática comum na comunicação entre servidores e clientes, no cenário deste aplicativo viu-se necessário realizar algum tipo de autenticação para cada requisição transmitida. O webservice funciona através de endereços de URL via HTTP, com isso, qualquer usuário mal-intencionado tem a possibilidade de consumir os *endpoints* fora de contexto, vazando dados sensíveis dos usuários.

Tabela 5: Ameaças ao webservice.

Ameaça	Vulnerabilidade	Probabilidade	Impacto
Download de dados de terceiros através do webservice.	Falta de autenticação no endpoint	Baixa	Alto
Envio de dados maliciosos através do webservice.	Falta de autenticação no endpoint	Baixa	Alto

Fonte: Próprio autor.

Para mitigação desta ameaça, foi proposto a utilização de uma chave de acesso, nomeada de *AccessKey,* a qual serve para a autenticação da requisição. A *AccessKey* é gerenciada apenas pelo servidor e, é transmitida ao usuário em tempo de *login,* para que o mesmo tome conhecimento de sua chave de acesso e, possa reutilizá-la em todas as transações que for efetuar. Tornando-se um campo obrigatório em qualquer requisição feita ao servidor. A *AccessKey* é utilizada para identificar o cliente e o usuário da transmissão, limitando o acesso aos *endpoints* do servidor, apenas para aqueles que possuam uma chave válida.

5.5. CRIPTOGRAFIA

A criptografia, se viu necessária para a ocultação dos dados referente a sessão do usuário, a fim de impedir a manipulação para enganar a comunicação com o

servidor, se passando por outro usuário, ou, explorando a autorização de uso do aplicativo.

Devido ao sistema Android, através de um processo de *Root* no aparelho, ser passível de manipulação dos arquivos de sistema referente aos aplicativos, abre-se a possibilidade da manipulação dos dados necessário para o funcionamento correto do sistema. Com isso, foi aplicado uma criptografia proprietária do sistema EquinoGestor em dois lugares:

- ***Shared preferences:*** O arquivo onde se registra as preferências do sistema, é intitulado como *Shared Preferences* e, é utilizado pelo aplicativo para o registro de dados a respeito da sessão do usuário autenticado naquele momento. A criptografia foi aplicada em todos os dados do arquivo, a fim de que, o mesmo só é lido corretamente sendo descriptografado anteriormente. O processo de descriptografar, leva em conta a sintaxe e, o correto preenchimento dos dados. Caso contrário, o aplicativo é encerrado e, o usuário é desconectado.

- **Transações da sincronização:** Como citado na implementação de disponibilidade, o processo de sincronização é dependente de uma chave de acesso para cada usuário, porém, tal chave também necessita de criptografia, para que, um usuário mal-intencionado não consiga manipulá-la para download dos dados referentes a outros usuários.

  A *AccessKey*, é utilizada como um “ingresso” que habilita o usuário de acessar recursos privados do servidor para o download de seus dados salvos em nuvem e, é tratada pelo aplicativo, apenas como uma *Hash*, utilizada somente para retransmitindo-a ao servidor, a fim de informar que o usuário possui o “ingresso” de autorização de acesso.

5.6. IMPLEMENTAÇÃO BÁSICA DO SDL

Como mencionado por Lipner e Howard (2005), o processo SDL é responsável por auxiliar todo o desenvolvimento de um software seguro. Para a implementação

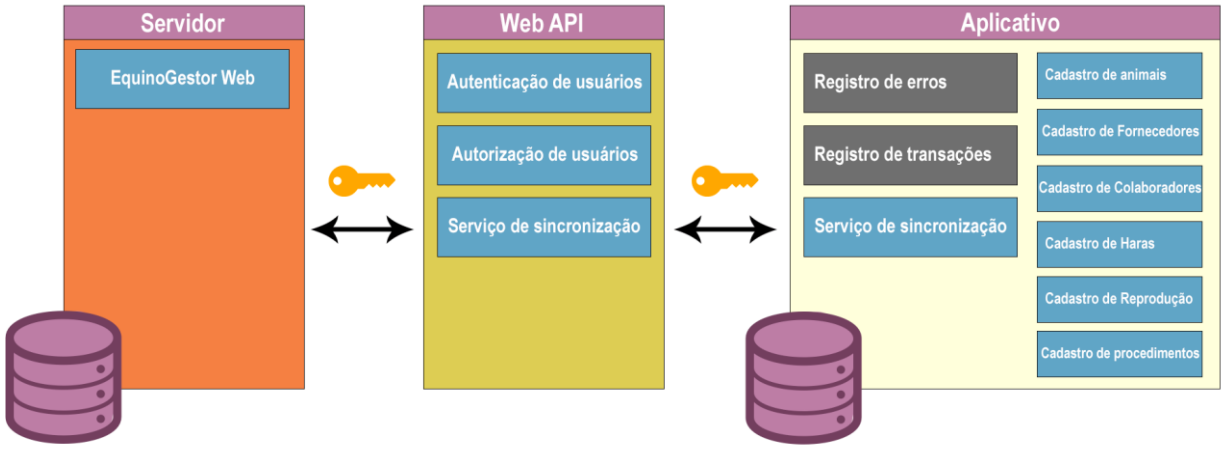
neste projeto, foram necessárias adaptações, de modo a suprir as necessidades da proposta, devido à falta de uma equipe especializada e, o desenvolvimento do software ser realizado de maneira autônoma. A implementação foi realizada da seguinte maneira:

- **Treinamento:** A fase de treinamento, não foi implementada. O treinamento recebido pelo desenvolvedor-autor, se resume ao estudo do tema através de referencial bibliográfico da área.

- **Requisitos:** A fase de requisitos no processo de desenvolvimento, resumiu-se basicamente em definir as obrigatoriedades que o aplicativo deveria apresentar em questões de segurança. Os três principais requisitos foram:
  - Capacidade de acesso à informação, mesmo não estando conectado à Internet, garantindo a disponibilidade.
  - Capacidade de sincronização com o servidor, para manter a integridade dos dados.
  - Restrição do acesso aos dados somente para clientes. Estando estes, não expirados e nem bloqueados, garantindo a confidencialidade.

- **Design:** A implementação da fase de design, foi responsável por arquitetar o funcionamento geral do aplicativo, documentando suas funções, assim como o planejamento do webservice (Web API). A modelagem das superfícies de ataque e das ameaças, também foram definidas na fase de Design.

Figura 10: Arquitetura geral EquinoGestor App.



Fonte: Próprio autor.

A arquitetura geral do software, interliga o sistema EquinoGestor Web com o aplicativo, através do *webservice* (Web API), pelo serviço de sincronização, transmitindo sempre as chaves de acesso do usuário. O aplicativo conta com diferentes funções desde cadastro de animais, fornecedores, colaboradores, haras, reproduções e procedimentos, até o registro dos erros e transações, efetuadas em conjunto com o sistema web.

- **Superfície de ataque e modelagem de ameaças:** Como ilustrado na Tabela 6, a modelagem da superfície de ataque foi desenvolvida juntamente com as ameaças provenientes de cada superfície, as categorizadas como Android, são referentes a configurações do sistema operacional como, a conexão com a Internet ou rede de dados móveis. Já a superfície classificada como servidor, remete apenas a possibilidade do mesmo estar indisponível, o que em primeira instância, sem a presença da contramedida, afetaria a disponibilidade das informações.

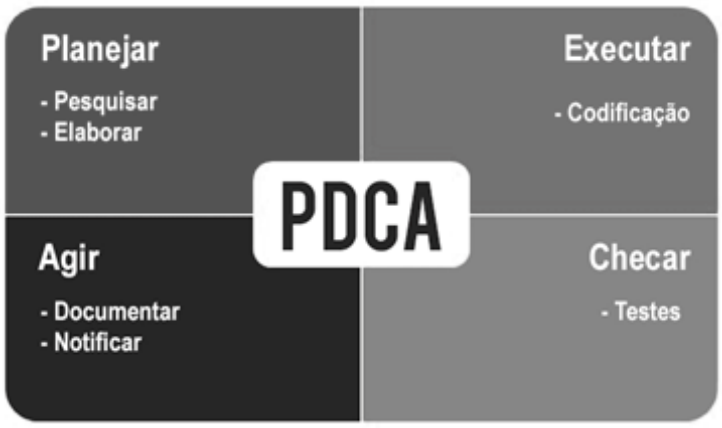
Tabela 6: Superfície de ataque e modelagem de ameaças.

Superfície de ataque	Ameaça	Contramedida
Formulários	• Registro de dados maliciosos	Filtragem e mascaramento dos Inputs
Sessão de acesso	• Alteração de dados	Criptografia e ofuscação dos dados
Banco de dados	• Manipulação dos dados • Conflito de chaves primárias	Chaves únicas baseando-se no registro de tempo
Serviço de sincronização	• Download e Upload de dados de terceiros	Utilização de chaves de acesso
Login	• Alteração de contas após primeira sincronização	Restrição de uma conta de usuário por aplicativo
Android	• Desconectar o dispositivo para burlar a autorização	Registro de *check-in* no próprio aparelho
Android	• Ausência de conexão com a Internet • Conexão de Internet lenta	Registro dos dados no banco de dados local
Servidor	• Servidor indisponível	Registro dos dados no banco de dados local

Fonte: Próprio autor.

- **Implementação:** O padrão de testes, foi definido, baseando-se no modelo PDCA (Planejar, executar, checar e agir), conforme Figura 11.

Figura 11: Ciclo PDCA.



Fonte: Próprio autor.

De maneira que, na fase de **planejamento**, eram pesquisadas e, elaboradas correções aos erros apresentados no relatório.

A **execução**, restringiu-se apenas à codificação das correções.

A **checagem** ocorria quando, após finalizada a implementação das correções, o aplicativo era exportado e, enviado aos funcionários responsáveis pelos testes, orientados a checarem os antigos erros, para que se confirme que a correção foi executada com sucesso.

Para completar o ciclo, a fase **agir**, é responsável pela criação do relatório das novas fases encontradas e, a notificação ao desenvolvedor.

- **Verificação:** A fase de verificação, foi responsável pela revisão das superfícies de ataque e suas contramedidas, assim como, a realização de testes de *Fuzzing* que são responsáveis por prover dados e eventos aleatórios para o aplicativo, a fim de detectar comportamentos anormais, os quais podem gerar vulnerabilidades para um atacante mais experiente, como por exemplo, a injeção de códigos maliciosos no sistema, a partir da parada inesperada do aplicativo.

  Os testes ocorreram de maneira manual desde a fase de implementação. A partir da fase de verificação, a automação foi feita com o uso de uma ferramenta própria do Android SDK, conhecida como Monkey.

  A Monkey é responsável por “estressar” o aplicativo, simulando diversos eventos aleatórios, como toques, cliques, rotações, alterações no volume e muitos outros. Todo o comportamento do aplicativo é registrado para uma análise posterior.

  O comando para a utilização da ferramenta foi:

```
adb shell monkey -p br.com.equinogestor -v 2000
```

  O parâmetro “-p”, é o nome do pacote do aplicativo instalado no dispositivo e, “-v” é o número de comandos aleatórios no qual a ferramenta executará.

  Os resultados, serviram de base para o aprimoramento do aplicativo, tornando-o mais estável e menos suscetível a ataques.

- **Lançamento e Resposta:** A fase de lançamento e a fase de resposta, tiveram uma implementação básica, de maneira que, o aplicativo é revisado periodicamente e atualizado conforme a necessidade dos clientes.

  Todas as falhas registradas pelo aplicativo, são notificadas ao servidor em processo de sincronização. Dessa maneira, ao identificar a causa da falha, todo o ciclo de implementação e verificação é realizado novamente, a fim de que, a descoberta de vulnerabilidades, sempre esteja restrita apenas a empresa, não colocando os clientes e os dados em risco.

6. CONSIDERAÇÕES FINAIS

A partir da análise dos dados e da pesquisa, observa-se a necessidade de uma atenção especial na segurança dos aplicativos desenvolvidos para dispositivos móveis.

Com a crescente presença dos aparelhos atualmente, é crucial que, para uma empresa, os aplicativos auxiliem ou complementem algum sistema existente, porém, de maneira segura, sem pôr em risco as informações organizacionais.

Como visto, o planejamento de um aplicativo, pode primeiramente, ser o responsável pela identificação básica das vulnerabilidades, porém, é de extrema importância que testes e revisões estejam presentes no decorrer do desenvolvimento, pois, através deles é que realmente se toma conhecimento das falhas mais severas.

Como apresentado no estudo de caso, diversas técnicas podem ser aplicadas, para as mais variadas funções de um aplicativo e, diferentes ferramentas podem ser utilizadas para se aplicar testes automatizados. Embora muitas vulnerabilidades possuam mitigações e correções conhecidas, deve sempre se atentar para a lógica da programação utilizada, a qual pode apresentar brechas que possam ser exploradas.

É importante ressaltar, que a presença de um servidor que se comunique com um aplicativo, adiciona uma nova gama de possibilidades a um usuário malicioso. Dessa maneira, o escopo do desenvolvimento se expande, levando em conta a proteção da comunicação entre as duas partes, de maneira a garantir que, em algum monitoramento, ou interceptação das mensagens, o *hacker* não consiga identificar as informações transmitidas.

Independentemente da não existência de uma segurança perfeita, um desenvolvedor, sempre é capaz de criar algo seguro. Basta que, se atente e se interesse pela segurança de seu software, escrevendo códigos de maneira mais responsável.

REFERÊNCIAS

BATORI, Ricardo. **Boas práticas de desenvolvimento seguro**. 2012. 11 p. Disponível em: <http://www.batori.com.br/whitepaper/BoasPraticasDesenvSeguro-v1.pdf>. Acesso em: 15 mar. 2017.

CVE DETAILS. **Vulnerabilities by date**. Disponível em: <http://www.cvedetails.com /browse-by-date.php>. Acesso em: 14 mar. 2017.

_____ **Vulnerabilities statistics**. Disponível em: <http://www.cvedetails.com/ product/19997/Google-Android.html?vendor_id=1224>. Acesso em: 14 mar. 2017.

GOODRICH, Michael T.; TAMASSIA, Roberto. **Introdução à segurança de computadores.** 1. ed. Porto Alegre: Bookman, 2013.

HADA, Satoshi. Zero-Knowledge and Code Obfuscation. In: OKAMOTO, T. **Advances in cryptology – ASIACRYPT 2000**. Lecture Notes in Computer Science, vol 1976. Springer, Berlin, Heidelberg: Asiacrypt 2000.

HARRIS, Shon. **All-in-one CISSP exam guide.** 6. ed. New York: Mc Graw Hill, 2013. 1432 p.

LEAVITT, Neal. **Mobile security**: finally a serious problem?**.** IEEE Computer Society. 2011. Disponível em: <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp= &arnumber= 5875929>. Acesso em: 16 mar. 2017.

LIPNER, Steve; HOWARD, Michael. **O ciclo de vida do desenvolvimento da segurança de computação confiável.** 2005. Disponível em: < https://msdn. microsoft.com/pt-br/library/ms995349.aspx>. Acesso em: 14 mar. 2017.

MICROSOFT. **SDL process**. Disponível em: < https://msdn.microsoft.com/en-us/library/windows/desktop/cc307406.aspx>. Acesso em: 27 mar. 2017.

_____ **SDL process**: phase 5: Release. Disponível em: <https://msdn.microsoft. com/en-us/library/windows/desktop/cc307420.aspx>. Acesso em: 20 mar. 2017.

_____ **SDL Process**: post-SDL Requirement: Response. Disponível em: <https://msdn.microsoft.com/en-us/library/windows/desktop/cc307411.aspx>. Acesso em: 20 mar. 2017.

OWASP. **Top 10 mobile risks – final list 2014**. Disponível em: <https://www.owasp. org/index.php/Mobile_Security_Project_Archive#tab=Top_10_Mobile_Risks>. Acesso em: 14 mar. 2017.

_____ **Mobile top 10 2014 – m1**. Disponível em: <https://www.owasp.org/index.php/ Mobile_Top_10_2014-M1>. Acesso em: 14 mar. 2017.

OWASP. **Mobile top 10 2014 – m2**. Disponível em: <https://www.owasp.org/index.php/Mobile_Top_10_2014-M2>. Acesso em: 14 mar. 2017.

_____ **Mobile top 10 2014 – m3**. Disponível em: <https://www.owasp.org/index.php/Mobile_Top_10_2014-M3>. Acesso em: 14 mar. 2017.

_____ **Mobile top 10 2014 – m4**. Disponível em: <https://www.owasp.org/index.php/Mobile_Top_10_2014-M4>. Acesso em: 14 mar. 2017.

_____ **Mobile top 10 2014 – m5**. Disponível em: <https://www.owasp.org/index.php/Mobile_Top_10_2014-M5>. Acesso em: 14 mar. 2017.

_____ **Mobile top 10 2014 – m6**. Disponível em: <https://www.owasp.org/index.php/Mobile_Top_10_2014-M6>. Acesso em: 14 mar. 2017.

_____ **Mobile top 10 2014 – m7**. Disponível em: <https://www.owasp.org/index.php/Mobile_Top_10_2014-M7>. Acesso em: 14 mar. 2017.

_____ **Mobile Top 10 2014 – m8**. Disponível em: <https://www.owasp.org/index.php/Mobile_Top_10_2014-M8>. Acesso em: 14 mar. 2017.

_____ **Mobile top 10 2014 – m9**. Disponível em: <https://www.owasp.org/index.php/Mobile_Top_10_2014-M9>. Acesso em: 14 mar. 2017.

_____ **Mobile top 10 2014 – m10**. Disponível em: <https://www.owasp.org/index.php/Mobile_Top_10_2014-M10>. Acesso em: 14 mar. 2017.

_____ **Mobile security project archive**. Disponível em: < https://www.owasp.org/index.php/Mobile_Security_Project_Archive#tab=Secure_Mobile_Development>. Acesso em: 20 mar. 2017.

OLIVEIRA, Diógenes. **Documentação do desenvolvedor EquinoGestor**. 2016. 68 p.