UpdaThing: um sistema de atualizações seguro para a Internet das Coisas

Tomás Pinho e Miguel L. Pardal

Instituto Superior Técnico - Universidade de Lisboa

Resumo. A Internet das Coisas tem acumulado interesse da indústria e dos consumidores, e dispositivos desta classe, com recursos limitados, têm sido aplicados a um ritmo acelerado, para tratar problemas identificados em diversos domínios. Os fabricantes têm de garantir o suporte a estes dispositivos e, por isso, têm de atualizar o seu *software* quando um defeito (*bug*) é encontrado e corrigido. Assim, atualizar o *firmware* que corre nestes dispositivos requer um mecanismo de atualizações, que apresenta alguns desafios de segurança, uma vez que pode ser utilizado por intervenientes mal-intencionados para comprometer uma grande quantidade destes dispositivos. Neste artigo, propomos, implementamos e avaliamos um sistema de atualizações de *firmware* para dispositivos *gateway*, que garante autenticidade do código descarregado (integridade e prova de origem), confidencialidade e frescura no processo de atualizações, oferecendo proteção contra as mais relevantes ameaças externas. O nosso sistema tem uso livre, realiza gestão de chaves e configurações de forma automática e incluí um subsistema e configurações para geração de imagens Linux, o que o diferencia de sistemas semelhantes.

Palavras-Chave: Internet das Coisas; Firmware; Actualização de Software; Segurança; Infraestrutura de Chave Pública; Sistemas Embebidos

1 Introdução

Ao longo dos últimos anos, tem havido um aumento na popularidade da Internet das Coisas (IoT) entre consumidores e fabricantes, e esta é agora uma bem-conhecida classe de dispositivos, dispondo de múltiplas bibliotecas de *software* e *kits* de desenvolvimento[17]. Estes sistemas, quando instalados em pequenas empresas ou em ambiente doméstico, são normalmente implementados através de um conjunto de sensores que capturam eventos ou medições e uma IoT *gateway*, um dispositivo com mais capacidades que serve de ponte entre dispositivos com menos recursos, e que operam utilizando protocolos específicos, e a Internet[12]. Os sensores transmitem dados, utilizando protocolos específicos de baixo consumo, à IoT *gateway*, que pode pré-processar os mesmos ou adicionar dados extra obtidos pelos seus próprios sensores (quando os tem). A *gateway* envia, então, os dados obtidos para um Centro de Dados, onde podem ser processados e depois disponibilizados através de outros meios, como Aplicações Web ou Móveis, por exemplo. O papel da *gateway* é ilustrado na Figura 1.

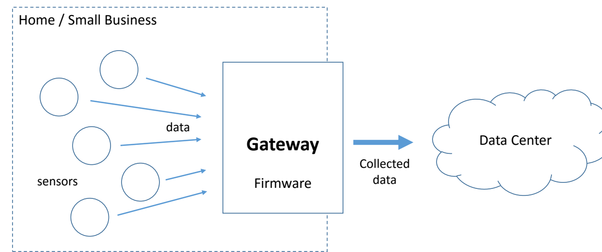

Figura 1. *Gateway* servindo de ponte entre dispositivos limitados e um *Data Center*.

"Todo o software tem bugs" é um facto conhecido da Engenharia de Software[13], por isso, os fabricantes têm de planear e providenciar algum tipo de processo de atualizações que permita corrigir dispositivos que já estão em utilização. As atualizações de *software* em dispositivos IoT são normalmente realizadas através da substituição do seu *firmware* na sua totalidade[5]. No entanto, se o sistema não for corretamente desenhado, este mecanismo pode ser uma porta de entrada para intervenientes maliciosos que podem comprometer estes dispositivos em grandes números, ganhando acesso a dados sensíveis[16].

Este artigo apresenta um mecanismo de atualizações de *firmware* seguro e de código-livre para IoT *gateways*. O nosso sistema assegura prova de origem, integridade e confidencialidade em trânsito das imagens de atualização, sendo ainda fácil de usar e instalar, uma vez que requer pouco conhecimento extra, além de conhecimentos gerais de desenvolvimento de *software* e instalações em sistemas Linux. Avaliámos o nosso sistema medindo o tráfego adicional causado pelo mesmo e o seu consumo de energia, antes de o comparar com outros trabalhos.

2 Requisitos

O nosso sistema tem como objetivo resolver problemas específicos de segurança provenientes de um sistema de atualizações de *firmware*, por isso a maioria dos requisitos têm que ver com as garantias de segurança:

I Assegurar autenticação de todas as partes que participam no processo de atualização: o dispositivo cliente que se atualiza e o servidor de atualizações;
II Assegurar comunicação segura entre o dispositivo cliente que se atualiza e o servidor de atualizações, garantindo integridade e confidencialidade da imagem transferida;
III Providenciar um sistema fácil de usar para gerar imagens Linux que incluem o *software* desenvolvido pelo fabricante;
IV Simplificar a gestão de chaves para que os programadores de *software* não tenham que a fazer manualmente. Este processo deve incluir todas as chaves e certificados digitais utilizados no processo de atualização.

3 UpdaThing

O nome escolhido para este trabalho foi UpdaThing e inclui todos os componentes ilustrados na Figura 2, organizados em sub-projetos: uma distribuição Linux e as suas ferramentas de desenvolvimento, duas implementações distintas de servidores e uma *daemon* de dispositivo.

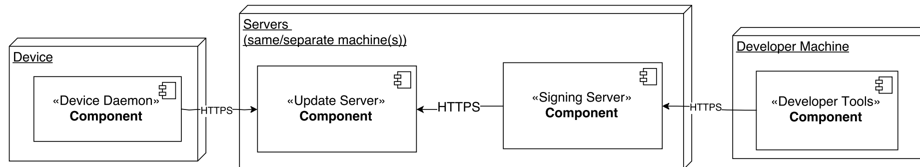

Figura 2. UpdaThing UML Deployment Diagram

Uma vez que há numerosos e distintos *chipsets* e ofertas System-On-a-Chip (SoC) para o desenvolvimento e integração de Internet of Things (IoT) *gateways*, escolhemos desenvolver um sistema de gestão de atualizações para dispositivos IoT que correm o Sistema Operativo (SO) Linux, em ARMv6 e ARMv7, como é o caso dos Raspberry Pi[1], que têm sido utilizados por muitas empresas como o SoC de eleição para os seus produtos.

O Raspberry Pi pode servir como uma IoT *gateway*, uma vez que se liga à Internet, através da sua porta Ethernet integrada (dependendo do modelo) ou um adaptador Wi-Fi Universal Serial Bus (USB), e porque pode servir de interface com outros sensores, através de outros adaptadores USB, como BlueTooth ou ZigBee. Os sensores também podem ser ligados diretamente através dos seus pins de General-Purpose Input/Output (GPIO), passando a tratar-se de uma IoT *gateway* com sensores integrados. O custo destes dispositivos, facilidade de uso e popularidade foram as razões principais pelas quais escolhemos estes SoCs para o nosso trabalho.

3.1 Distribuição uLinux

Escolhemos construir a nossa própria distribuição de Linux uma vez que o nosso objetivo é desenvolver para dispositivos que beneficiam de arrancar ambientes simples e pequenos, e porque pretendíamos controlar o processo de arranque do dispositivo e o seu sistema *init* para aplicar atualizações de *firmware*. Buildroot é um conjunto de *Makefiles* que permite a configuração, compilação e geração de imagens personalizadas de sistemas de ficheiros Linux[1], permitindo-nos gerar estruturas que contêm apenas os pacotes que precisamos.

[1] Raspberry Pi é uma série de computadores de constituídos por uma única placa de circuito impresso do tamanho de um cartão de crédito desenvolvidos no Reino Unido pela Raspberry Pi Foundation. Têm *hardware* parcialmente aberto e relativamente barato, custando entre EUR 5 e 40.

O armazenamento permanente dos nossos dispositivos está dividido em três partições: a primeira partição é onde a *kernel* com o *initramfs* mínimo é guardada e de onde é arrancada pelo Raspberry Pi; a segunda partição é onde está o sistema de ficheiros principal e que é utilizado em modo apenas de leitura; a terceira partição funciona como uma camada em modo de escrita e que é colocada por cima do conteúdo da segunda partição. Estas duas últimas partições são montadas uma em cima da outra pelo processo de *init*.

A nossa primeira derivação do Buildroot permite-nos gerar um *initramfs* mínimo que contém apenas a kernel *Linux*, uma cópia do Busybox (uma pequena implementação de vários comandos Unix) e o nosso *script* de *init*. Este *script* trata de escrever a imagem previamente transferida para a segunda partição, antes do resto dos serviços terem iniciado, através da verificação de uma localização predefinida do sistema de ficheiros pela existência de uma imagem já verificada, escrevendo-a para a partição de arranque e depois apagando-a, prosseguindo com o arranque do resto do sistema através da mudança do sistema de ficheiros *root* e a inicialização do sistema *init* do próprio Busybox, que arranca o resto dos serviços.

A nossa segunda derivação do Buildroot é responsável por gerar imagens do sistema de ficheiros *root* e contém directivas para instalar pacotes como utilidades de rede, interpretadores de linguagens de programação e a nossa *Daemon* de Dispositivo. Esta é a árvore de código fonte do Buildroot que será utilizada pelos programadores que trabalham na integração das aplicações desenvolvidas pelo fabricante nos dispositivos. Escrevendo um simples *Makefile*, é possível incluir qualquer aplicação no processo de compilação do Buildroot e esta ser incluída na imagem gerada. O programador tem apenas que clonar localmente o nosso repositório, instalar ferramentas de compilação e escrever *make* em cada árvore de ficheiros para gerar imagens funcionais.

3.2 Componentes

A distribuição uLinux é usada pelo fabricante para gerar imagens de Linux funcionais. Uma ferramenta incluída no pacote pode ser utilizada para enviar as imagens resultantes para o Servidor de Assinaturas.

O Servidor de Assinaturas é responsável por receber imagens de *firmware* simples e incluir chaves, certificados e ficheiros de configuração nas mesmas. Estes artefactos fazem parte do processo de gestão de chaves automático e são usados pela *Daemon* de Dispositivo para autenticar imagens e a identidade do Servidor de Atualizações durante o *handshake* do protocolo Transport Layer Security (TLS). Para fazê-lo, o Servidor de Assinatura arranca um *container* Docker[2]que corre um *shell script* próprio que monta a imagem do sistema de ficheiros e copia os ficheiros necessários para a mesma. Depois, o Servidor de Assinaturas calcula a *hash* SHA512 da resultante imagem e assina-a com a sua Chave Privada RSA,

[2] Docker é um projeto de código aberto que automatiza o lançamento de aplicações dentro de *containers* de *software*.

incluindo-a com a imagem. Finalmente, o Servidor de Assinaturas envia este pacote ao Servidor de Atualizações.

O Servidor de Atualizações é responsável por: notificar clientes "vivos" de novas atualizações; servir os pacotes de atualizações através do seu ID quando requisitado; receber mensagens de prova de vida (*I'm Alive*) de dispositivos; e responder a pedidos dos IDs de novas atualizações através de um selo temporal.

A *Daemon* de Dispositivo está responsável por periodicamente enviar mensagens *I'm Alive* para o Servidor de Atualizações, ser notificada da existência de novos pacotes de atualizações, pedir os IDs de novos pacotes de atualizações através de um selo temporal quando arranca e descarregar e verificar imagens de *firmware*. As imagens são verificadas através da abertura do arquivo Tar, fazendo a *hash* da imagem de *firmware* incluída usando SHA512 e verificando se é igual à *hash* assinada que vem dentro do arquivo, depois de verificar a sua assinatura usando a Chave Pública do Servidor de Assinaturas que se encontra na memória permanente do dispositivo. Em seguida, a *Daemon* reinicia o Dispositivo, ficando o *init script* descrito na Secção 3.1 responsável pela escrita da imagem para a partição de arranque do dispositivo. A *Daemon* de Dispositivo também tenta estabelecer regras de reencaminhamento de portas com o *router* ao qual o dispositivo está ligado, através do uso de UPnP[2][3]. Isto permite à *Daemon* de Dispositivo receber notificações *push* diretamente do Servidor de Atualizações, quando se encontra por detrás de um *router* doméstico compatível e que disponha de *firewall* e Network Address Translation (NAT).

Para que seja possível comunicação autenticada entre o Dispositivo e o Servidor de Atualizações, a *Daemon* de Dispositivo tem à sua disposição, mediante inclusão na imagem por parte do Servidor de Assinaturas, uma Certificate Authority (CA) comum a todos os dispositivos com a qual pode assinar certificados para si, quer para atuar nas comunicações como cliente ou servidor, sendo estes gerados, em conjunto com os respetivos pares de chaves público-privada, no primeiro arranque da *Daemon*, se esta detetar a sua falta. Assim, o Servidor de Atualizações apenas confia em certificados assinados por esta CA permitindo dessa forma a utilização dos seus *endpoints* apenas a dispositivos com certificados válidos.

O Servidor de Assinaturas, o Servidor de Atualizações e a *Daemon* de Dispositivo foram implementados em Node.js, que permite rápida prototipagem e que fornece APIs úteis, como implementações de HTTP, TLS e uma biblioteca de Criptografia que satisfaz o nosso caso de uso.

3.3 Protocolos

Nesta secção detalhamos os *endpoints* expostos por cada um dos nossos componentes, as mensagens que são enviadas e recebidas e a lógica sequencial por detrás de cada uma. Como explicado anteriormente, estes *endpoints* são expostos através de HTTPS e beneficiam de todas as garantias dadas pelo protocolo TLS. Os *endpoints* implementados são os seguintes:

[3] Universal Plug and Play (UPnP) é um conjunto de protocolos de rede que permitem que dispositivos ligados a uma mesma rede se descubram sem intervenção humana e estabeleçam automaticamente serviços de rede.

- Servidor de Atualizações
  - GET /updates/{id} - Descarregar um pacote de atualização pelo seu ID. Tem que providenciar um certificado de dispositivo válido.
  - POST /newUpdate - Obter o ID do último pacote de atualizações depois de um dado timestamp. Post body: { timestamp: UNIX Timestamp (Number) }. Tem que providenciar um certificado de dispositivo válido.
  - POST /updates/ - Enviar um novo pacote de atualização. Tem que providenciar um certificado de servidor válido e Auth token de Servidor de Assinaturas (no header HTTP).
  - POST /imAlive - Enviar mensagem *I'm Alive*. Post body: { deviceId: UUID (String), firmwareVersion: Number, port: Number }. Tem que providenciar um certificado de dispositivo válido.
- Servidor de Assinaturas
  - POST /updates/ - Enviar uma nova imagem de *firmware*. Tem que providenciar um Auth token válido de programador.
- *Daemon* de Dispositivo
  - POST /newUpdate - Notificar de um novo pacote de atualizações. Post body: { id: Update ID (Number), timestamp: UNIX Timestamp (Number)}. Tem que providenciar um certificado de servidor válido.

De seguida apresentam-se dois exemplos do uso sequencial dos *endpoints*:

- *Daemon* de Dispositivo tenta atualizar-se no arranque: chama o *endpoint* "POST /newUpdate" do Servidor de Atualizações, se um ID foi retornado, então chama "GET /updates/{id}" para descarregar o pacote e continuar com o processo de atualizações.
- *Daemon* de Dispositivo é notificada de uma nova atualização: o Servidor de Atualizações chama o *endpoint* "POST /newUpdate" exposto pela mesma. Depois, a *Daemon* de Dispositivo chama "GET /updates/{id}" do Servidor de Atualizações para descarregar o pacote e continuar com o processo de atualizações, se a versão é mais recente.

4 Avaliação

4.1 Satisfação de Requisitos

Nesta secção fazemos um breve resumo dos requisitos que foram satisfeitos e como o foram. O Requisito I foi satisfeito através da utilização de certificados X.509 assinados pelas Certificate Authorities na parte de autenticação do *handshake* do protocolo TLS. O Requisito II foi satisfeito pelo uso de TLS que garante confidencialidade e integridade dos dados transferidos. O Requisito III foi satisfeito pelas Ferramentas de Desenvolvimento que fazem parte desta sistema e que permitem aos desenvolvedores configurar e gerar automaticamente imagens completas de Linux usando Buildroot. Finalmente, o Requisito IV foi satisfeito pelo processo de incluir todos os artefactos necessários antes de assinar uma dada imagem de *firmware* no Servidor de Assinaturas.

4.2 Análise STRIDE

Realizámos uma extensa análise STRIDE a uma versão do sistema sem qualquer mecanismo de segurança. Enumeramos brevemente os ataques possíveis mais interessantes: é possível que um atacante realize um ataque Man-in-the-Middle (MitM) e se faça passar por qualquer uma das entidades do sistema (*Spoofing*), tal permitiria forçar o sistema a assinar e disponibilizar uma imagem de *firmware* maliciosa que infetaria todos os dispositivos, ou, no caso em que o ataque seria realizado entre o Dispositivo e o Servidor de Atualizações, forçando a sua instalação diretamente, através dos pedidos de atualização no arranque ou enviando uma notificação *push*, em ambos os casos obtém-se acesso ao Dispositivo (*Elevation of Privilege*), sendo também possível obter acesso a imagens fidedignas desta forma (*Information Disclosure*) ou incapacitar o Dispositivo ou o Servidor de Assinaturas através do carregamento de uma imagem inválida (*Denial of Service*); É possível ainda que um Programador repudie as imagens que envia para o sistema, se não houver autenticação do utilizador (*Repudiation*). Acreditamos que a utilização de TLS como parte do HTTPS e autenticação do programador através do *Auth token* mitigam todos os ataques mencionados desde que os certificados, as suas chaves, a Chave do Servidor de Assinaturas e o *token* do programador não sejam obtidos por um atacante. Concluímos que o nosso sistema é seguro e está protegido contra todos os ataques enumerados.

4.3 Tráfego adicional gerado

Medimos o tráfego adicional que foi gerado para proteger as comunicações do nosso sistema. Para isso, utilizámos o Wireshark[4], que é um conhecido analisador de protocolos, para capturar sessões de atualização de *firmware*, um diagrama da montagem pode ser encontrado na Figura 3.

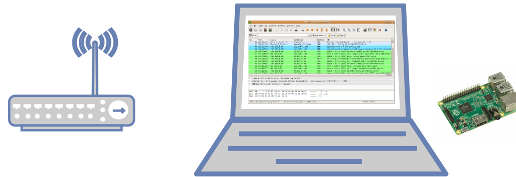

Figura 3. Diagrama de montagem para medições de trafégo adicional.

Na Tabela 1, os leitores podem ver a média de medidas que foram realizadas. Para cada pedido HyperText Transfer Protocol (HTTP), 3 medidas foram realizadas a partir das estatísticas de conversação Transmission Control Protocol (TCP) recolhidas pelo Wireshark. Todos os dispositivos estavam ligados à mesma rede Wi-Fi. Para referência, a imagem que foi transferida para efeitos de teste tinha um tamanho em disco de 140,25MB.

[4] https://www.wireshark.org/

Ação	**Quantidade de Tráfego**
Verificação de imagens mais recentes que o selo temporal da imagem atualmente instalada	7.6 kB
Envio de mensagem *I'm Alive*	7.6 kB
Descarregamento de uma imagem de *firmware*[5]	148 MB
Notificação *push* de uma nova imagem de *firmware*[6]	4.6 kB

Tabela 1. Tráfego causado por cada pedido HTTP

A discrepância entre o valor medido (148 MB) e o tamanho da imagem em disco (140,25 MB) é causada pelo tráfego adicional gerado pelo protocolo TLS, que é no mínimo de 6%[8]. Tal acontece porque o TLS adiciona uma entrada que varia entre 20 e 40 bytes no *header* do pacote, a acrescentar às entradas dos protocolos IP e TCP. Os autores acreditam que os valores medidos de tráfego adicional são desprezáveis com uma ligação moderna à Internet e que são um muito pequeno preço a pagar pelas propriedades de segurança garantidas pelo UpdaThing.

4.4 Consumo de Energia

Como *gateways* IoT são dispositivos que estão ligados 24/7, devemos garantir que a nossa solução é energeticamente eficiente para que não tenha este critério como obstáculo à sua adoção. Por isso, testámos a nossa *Daemon* de Dispositivo em vários estados relativamente ao seu consumo de energia. Colocámos um multímetro em série com o Raspberry Pi 2, com o adaptador Wi-FI USB ligado (TP-Link TL-WN722N), e registámos a corrente que passou pelo circuito. Um diagrama com a nossa montagem pode ser encontrado na Figura 4.

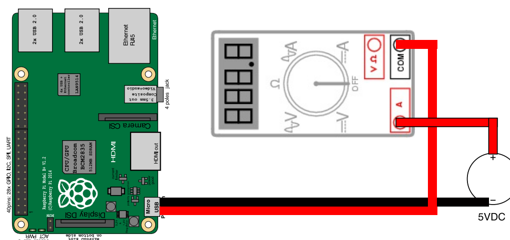

Figura 4. Diagrama de montagem para medições de corrente.

O Raspberry Pi 2 opera a 5V. A média dos resultados que obtivemos encontram-se na Tabela 2, tendo sido realizadas 3 medidas para cada um

[6] A mesma imagem foi descarregada e escrita para memória permanente 3 vezes. Utilizámos modificações diretas da base de dados do Servidor de Atualizações para mudar o selo temporal da imagem para forçar a sua instalação.

[6] A imagem não é transmitida. O Dispositivo é apenas notificado e prossegue a instalação noutro pedido HTTP.

Estado	**Corrente (A)**	**Potência (W)**
Inativo	0.46	2.3
Rádio Wi-fi a piscar (em operação)	0.47	2.35
Arranque da VM Node.Js + *Daemon* de Dispositivo verificando por atualizações	0.52	2.6
Descarregando imagem de *firmware*	0.51	2.55
Verificando imagem de *firmware*	0.50	2.5
Escrevendo imagem de *firmware* para memória permanente	0.34	1.7

Tabela 2. Consumo de energia do Dispositivo em cada estado. A Potência Elétrica é dada por P = V I.

dos estados para calcular a mesma. A diferença que existe entre os estados de funcionamento normal e o de escrever a imagem de *firmware* para a memória tem que ver com a ativação do adaptador Wi-Fi, que não se encontra em funcionamento antes do arranque total do sistema. A diferença entre o Dispositivo num estado inativo e a descarregar ou verificar imagens está na ordem dos deciwatts e os autores acreditam que a mesma é desprezável e não deve ser um facto preocupante em cenários onde o Dispositivo está ligado à rede elétrica, mesmo quando o débito de *download* é menor, demorando o Dispositivo mais tempo a descarregar a imagem.

Imaginando um cenário em que o Raspberry Pi está ligado a uma bateria de 3000mAh, uma capacidade comum para uma bateria de um *smartphone* de gama alta, e utilizando um fator de 0.7 para ter em conta com fatores externas que podem afetar a vida da bateria, o tempo que o Dispositivo demora a gastar a bateria é dado por $Capacidade(Ah) \div Consumo(A) \times 0.7$. Dessa forma, num estado inativo, o Dispositivo permanece ligado durante 4.57h quando ligado à bateria previamente referida, enquanto descarrega imagens consegue permanecer ligado 4.12h e a bateria dura 4.2h a verificar imagens, estes últimos resultados são assumindo que o Dispositivo o faz continuamente, que não é o caso.

Uma vez que o Dispositivo está inativo a maior parte do tempo, que descarregar uma imagem de 120Mb demora alguns minutos numa ligação à Internet moderna, verificar a imagem demora perto de 8s e escrevê-la para a memória permanente demora perto de um minuto, tendo em conta o tempo que o Dispositivo demora a arrancar, todos os outros estados são desprezáveis para o seu consumo energético a longo prazo. Isto depende, claro, se a aplicação principal do Dispositivo (implementada pelos desenvolvedores do fabricante) faz uma utilização intensiva do rádio, caso em que o consumo energético no estado inativo é relativamente maior.

5 Trabalho Relacionado

Foi realizado um levantamento de artigos sobre trabalhos de atualizações de *firmware*. A Tabela 3 apresenta um breve resumo das garantias de segurança dadas por cada sistema.

Tipo de Sistema	Sistema	Autenticação da Origem	Integridade (trânsito)	Confidencialidade (trânsito)	Assinatura do Código	Tipo de Cifra
Artigos	Nilsson e Larson[15]	✓	✓	✓	✓	assimétrica/simétrica
	Costa et al.[6]	✓	✓	✗	✓	assimétrica
	Itani et al.[10]	✓	✓	✗	✗	simétrica
	Katzir e Schwartzman[11]	✓	-	-	-	-
	Cao et al.[4]	✓	✓	✓	✗	simétrica / nenhuma
Exclusivos	Routers Domésticos[9]	✗	✗	✗	✗	✗
	SO Android [3]	✓	✓	✗	✓	assimétrica
	Termóstato NEST[14]	✓	✓	✓	✓	assimétrica
	Router Google OnHub[7]	✓	✓	✓	✓	assimétrica

Tabela 3. Resumo das garantias de segurança dadas por cada sistema estudado.

A maior parte dos sistemas utiliza Assinaturas de Código (alguns através de cifras simétricas) para assegurar a Integridade e a Autenticação da Origem da imagem de *firmware*. Uma vez que consideramos o ato de Assinar Código como a assinatura de um binário utilizando uma chave privada, não contamos alguns sistemas em que tal é feito utilizando cifras simétricas. A Confidencialidade da imagem transferida é garantida através da utilização de chaves simétricas na maior parte dos sistemas estudados.

O sistema proposto por Cao et al.[4] apenas garante Integridade e Confidencialidade da imagem até que seja decifrada. A imagem é depois transferida sem proteções para o implante. Por isso, um atacante que possa escutar fisicamente as comunicações entre o dispositivo atualizador e o implante, pode realizar ataques ao processo de atualização. No entanto, esta situação é altamente improvável e apresenta um grau de risco elevado, uma vez que levantaria suspeições por parte dos profissionais de saúde no local.

O SO Android[3] nem sempre descarrega atualizações por HTTP, sendo o protocolo a usar definido pelo fabricante na URL usada, ficando a confidencialidade da imagem em trânsito dependente da mesma. Embora os pacotes de atualizações normalmente incluam uma *hash* assinada pelo fabricante, que permite garantir integridade e autenticação de código descarregado, o sistema de *recovery*, o sistema responsável pelo processo de atualização, pode não obedecer às normas do projeto Android e não a requerer ou validar.

Não encontrámos nenhum sistema completamente implementado nos artigos consultados que garantisse as quatro propriedades sem deixar espaço para um ataque bem-sucedido. Por exemplo, o sistema de Nilsson e Larson[15] deixa a gestão de chaves para trabalho futuro e funciona com chaves simétricas únicas e

permanentes para comunicações entre os veículos e o portal, o que faz com que o sistema seja mais facilmente atacado, no que toca à confidencialidade da imagem.

Dois sistemas exclusivos mais próximos do que desenvolvemos são o Termóstato NEST[14] e o Router Google OnHub[7]. Ambos se atualizam por HTTPS e utilizam TLS como forma de assegurar integridade e confidencialidade dos dados transferidos, e ambos utilizam Criptografia de Chave Pública para assinatura do código. Não é claro que autentiquem o dispositivos, pelo que a nossa solução difere pela utilização de certificados para proceder a esta autenticação, ter código aberto e permitir a geração de imagens Linux.

6 Conclusão

Com este artigo, propusemos uma solução para um dos grandes problemas de segurança na IoT: como realizar atualizações de *firmware* seguras em dispositivos *gateway*. Estabelecemos requisitos para sistemas destinados à resolução deste problema e escolhemos HTTP por cima de TLS como forma de resolver os problemas de segurança na comunicação, e *hashing* e Criptografia de Chave Pública como forma de garantir a assinatura da imagem transferida. Implementámos o sistema, descrevemos e avaliámos a sua arquitetura com base em três aspetos: os requisitos satisfeitos, medições de tráfego adicional e o consumo de energia, e, por fim, comparação com sistemas já existentes.

Desenvolvimentos futuros nesta área incluem o uso de Trusted Platform Modules (TPM) para guardar palavras-chave e certificados e realizar operações criptográficas nos dispositivos em atualização, aliviando o sistema de ficheiros do armazenamento destes artefactos importantes e dificultando a sua obtenção por um atacante. Também se deverá trabalhar no sentido de desenvolver estratégias de *logging* leves para dispositivos IoT. O desenvolvimento de *daemons* leves para *logging* para dispositivos IoT que integram com soluções como o Logstash[7] seria um projeto de investigação interessante. O nosso sistema não foi testado ou avaliado em dispositivos com menos capacidades, como sensores que comuniquem através de ZigBee ou IEEE 802.15.4, pelo que deverá também ser desenvolvido trabalho nesta vertente. Também não foi desenvolvido a pensar na manutenção de versões de *firmware* diferentes para dispositivos de diferentes arquiteturas, por exemplo, nem é possível a seleção de um dispositivo específico para a atualização do mesmo, pelo que tais possibilidades poderão vir a ser estudadas.

Optámos por conduzir o nosso processo de atualização de *firmware* através da substituição integral de imagens de *firmware* em dispositivos em atualização, o que acarreta ineficiências quando as atualizações não se traduzem em muitas alterações. Um projeto de investigação interessante seria o de expandir o nosso sistema com funcionalidades que realizassem atualizações diferenciais no sistema de ficheiros do dispositivo, substituindo apenas secções sujeitas a alterações.

Concluímos que o nosso sistema poderá ser uma alternativa para fabricantes empenhados na implementação de um sistema seguro de atualizações de *firmware*

[7] Logstash é uma *pipeline* de dados destinada ao processamento de *logs*. https://www.elastic.co/products/logstash

que não pretendam desenvolver software proprietário nem a infraestrutura a isso necessária. Todos os interessados poderão encontrar a totalidade do código do projeto disponibilizado em https://github.com/ulinux-embedded.

Bibliografia

1. Buildroot. https://buildroot.org (14 de Julho de 2016)
2. Open Connectivity Foundation - UPnP. http://openconnectivity.org/upnp (14 de Julho de 2016)
3. Android Open Source Project: Ota updates. https://source.android.com/devices/tech/ota/index.html (14 de Julho 2016)
4. Cao, Y., Hu, C., Ma, B., Hao, H., Li, L., Wang, W.: Secure method for software upgrades for implantable medical devices. Tsinghua Science and Technology (Volume:15 , Issue: 5) (2010)
5. Case, L.: How to update firmware on pcs and peripherals. http://www.pcadvisor.co.uk/how-to/software/how-to-update-firmware-on-pcs-and-peripherals-3237825/ (14 de Julho de 2016)
6. Costa, L.C., Herrero, R.A., Biase, M.G.D., Nunes, R.P., Zuffo, M.K.: Over the air download for digital television receivers upgrade. Journal IEEE Transactions on Consumer Electronics archive, Volume 56 Issue 1, February 2010 (2010)
7. Google: Onhub security features. https://support.google.com/onhub/answer/6309220?hl=en (14 de Julho de 2016)
8. Grigorik, I.: High Performance Browser Networking. O'Reilly (2013)
9. Independent Security Evaluators: Exploiting SOHO routers. http://www.securityevaluators.com/knowledge/case_studies/routers/soho_router_hacks.php (14 de Julho de 2013)
10. Itani, W., Kayssi, A., Chehab, A.: Petra: a secure and energy-efficient software update protocol for severely-constrained network devices. Q2SWinet '09 Proceedings of the 5th ACM symposium on QoS and Security for Wireless and Mobile networks (2009)
11. Katzir, L., Schwartzman, I.: Secure firmware updates for smart grid devices. Innovative Smart Grid Technologies (ISGT Europe), 2011 2nd IEEE PES International Conference and Exhibition on (2011)
12. Konsek, H.: IoT gateways and architecture. The DZone Guide to the Internet of Things (2015)
13. McConnell, S.: Code Complete: A Practical Handbook of Software Construction. Microsoft Press (2004)
14. Nest: Keeping data safe at nest. https://nest.com/security/ (14 de Julho de 2016)
15. Nilsson, D.K., Larson, U.E.: Secure firmware updates over the air in intelligent vehicles. Communications Workshops, 2008. ICC Workshops '08. IEEE International Conference on (2008)
16. Rubenking, N.J.: Securing the internet of things. http://www.pcmag.com/article2/0,2817,2483635,00.asp (14 de Julho de 2016)
17. Suresh, P., Daniel, J., Parthasarathy, V., Aswathy, R.: A state of the art review on the Internet of Things (IoT): history, technology and fields of deployment. Science Engineering and Management Research (ICSEMR), International Conference on (2014)