## Embarcados Experience 2020

# Segurança em Linux embarcado

Sergio Prado

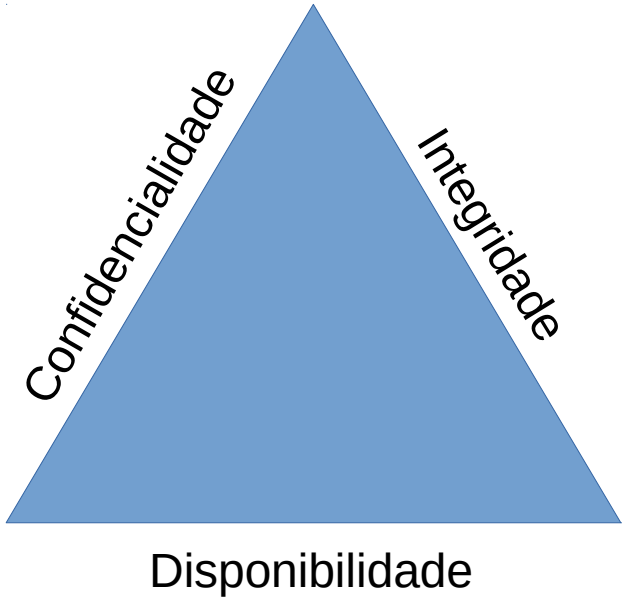
@sergioprado

https://embeddedbits.org

# $ WHOAMI

- Desenvolvedor de software embarcado por mais de 25 anos.
- Team Lead na Toradex.
- Consultor e professor na Embedded Labworks (e-labworks.com).
- Contribuidor de alguns projetos de software livre, incluindo Buildroot, Yocto Project e o Linux kernel.
- Blogger em embeddedbits.org e sergioprado.org.

# INTRODUÇÃO À SEGURANÇA




Segurança da informação é basicamente uma área de gerenciamento de riscos!

# CONCEITOS NA ÁREA DE SEGURANÇA

- **Proprietários**: aqueles que se beneficiam com o produto (usuário, fabricante, empresário, etc).
- **Ativos**: tudo que tem valor para os proprietários (dados, código, reputação, etc).
- **Ameaças**: qualquer coisa que seja capaz de agir contra um ativo de maneira que possa resultar em danos.

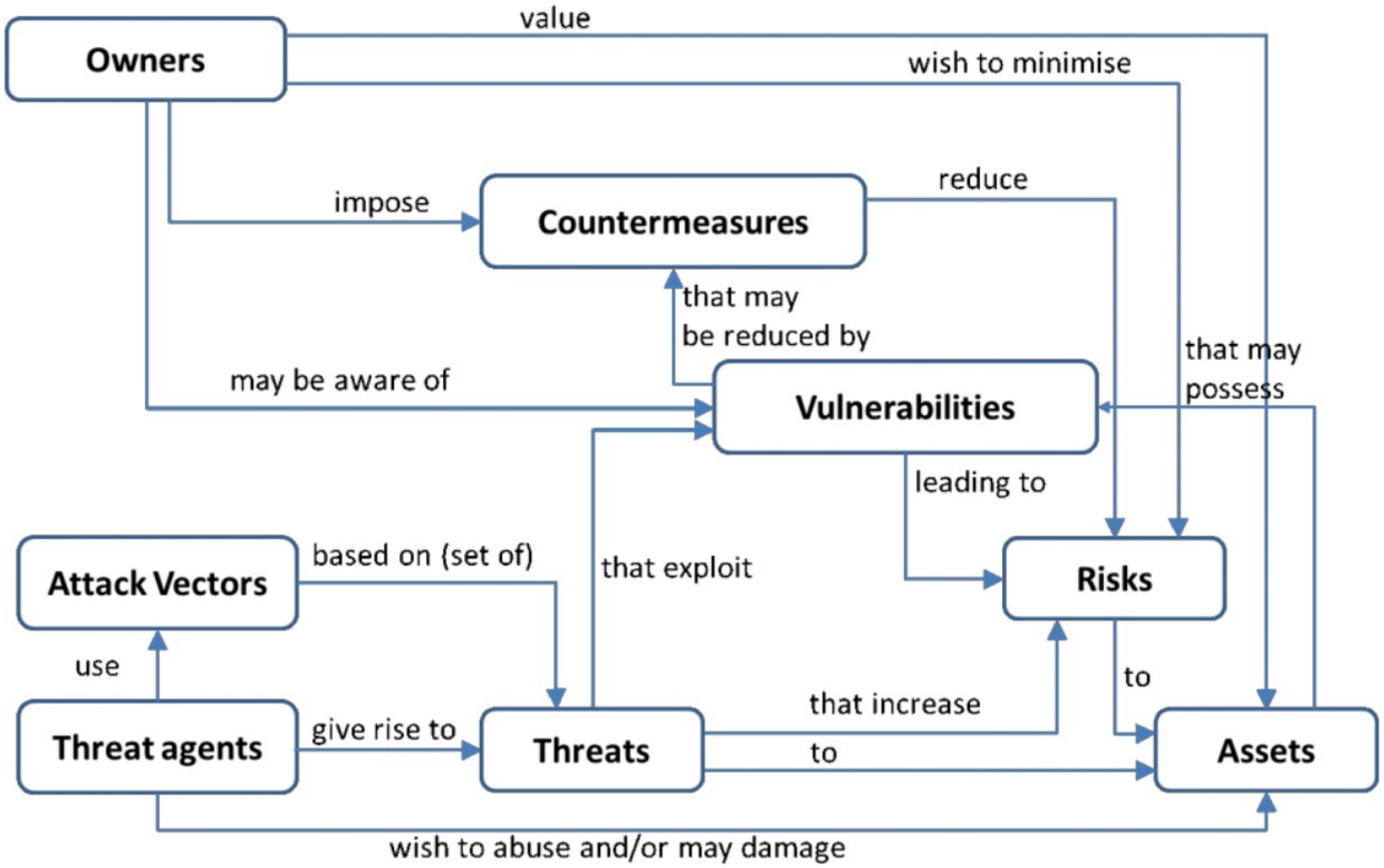
# CONCEITOS NA ÁREA DE SEGURANÇA

- **Agente de ameaça** (threat actor): pessoa ou coisa que pode manifestar uma ameaça (hacker malicioso, governo, etc).

- **Vetores de ataque** (ou superfície de ataque): métodos ou caminhos usados por um agente de ameaça para acessar ou invadir um sistema alvo.

- **Vulnerabilidades**: fraqueza que pode ser explorada por um agente de ameaça.

# CONCEITOS NA ÁREA DE SEGURANÇA

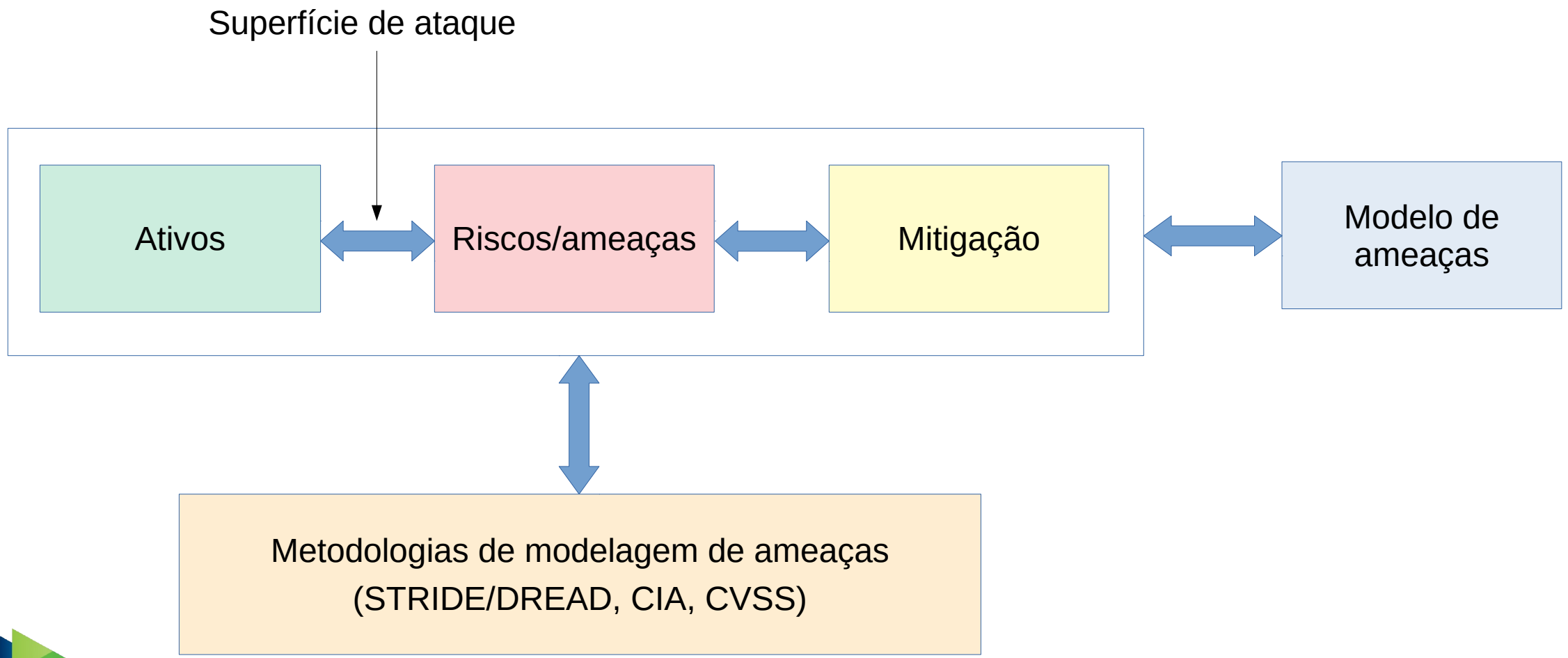



Fonte: Enisa - Hardware Threat Landscape and Good Practice Guide

# MODELAGEM DE AMEAÇAS

- A segurança trata da identificação de ameaças para minimizar os riscos de comprometimento dos ativos.
- A **modelagem de ameaças** (threat modeling) é um processo onde as ameaças potenciais podem ser identificadas, enumeradas e as mitigações podem ser priorizadas.
- É basicamente um processo de análise de riscos em que você avalia o valor de seus ativos e o custo para protegê-los.
- O resultado da modelagem de ameaças é o **modelo de ameaças** (threat model) de seu produto.

# MODELAGEM DE AMEAÇAS



# STRIDE

| Threat | Definition | Property | Example |
|---|---|---|---|
| **S**poofing | Pretend to be someone else. | Authentication | Hack victim's email and use to send messages in name of the victim. |
| **T**ampering | Change data or code. | Integrity | Software executive file is tampered by hackers. |
| **R**epudiation | Claiming not to do a particular action. | Non-repudiation | "I have not sent an email to Alice". |
| **I**nformation Disclosure | Leakage of sensitive information. | Confidentiality | Credit card information available on the internet. |
| **D**enial of Service | Non-availability of service | Availability | Web application not responding to user requests. |
| **E**levation of privilege | Able to perform unauthorized action | Authorization | Normal user able to delete admin account |

Fonte: https://allabouttesting.org/stride-acronym-of-threat-modeling-system/

# DREAD

| Rating | High (3) | Medium (2) | Low (1) |
|---|---|---|---|
| Damage potential | Attacker can subvert security system, get full trust authorization, run as admin, upload content | Leaking sensitive information | Leaking trivial information |
| Reproducibility | Attack can be reproduced every time and does not require a timing window | Attack can be reproduced but only with timing window and particular race situation | Attack is difficult to reproduce, even with knowledge of the security hole |
| Exploitability | A novice programmer could make the attack in a short time | A skilled programmer could make the attack, then repeat the steps. | Attack requires extremely skilled person and in-depth knowledge every time to exploit |
| Affected users | All users, default configuration, key customers | Some users, non-default configuration | Small percentage of users, obscure feature; affects anonymous users |
| Discoverability | Published information explains the attack; Vulnerability is in most commonly used feature | Vulnerability is in seldom-used part of product | The bug is obscure; unlikely that users will work out damage potential |

Fonte: https://www.slideshare.net/SecurityInnovation/threat-modeling-to-reduce-software-security-risk

# EXEMPLO DE MODELAGEM DE AMEAÇAS

| Ameaça | Pontuação | Mitigação |
|---|---|---|
| Qualquer usuário pode fazer login na página web de administração e alterar a configuração do dispositivo | 14 | Implementar um mecanismo de autenticação |
| Uma aplicação pode ser explorada para executar código não autorizado | 13 | Derrube os privilégios da aplicação e execute-a dentro de um contêiner |
| Com acesso físico, um agente de ameaça pode extrair dados do usuário | 12 | Use criptografia para proteger os dados do usuário |
| Usando um ataque MitM, um agente de ameaça pode alterar a imagem do firmware durante o processo de atualização | 11 | Verifique a assinatura da imagem de atualização do firmware |
| Um agente de ameaça pode executar ataques DoS no dispositivo | 11 | Crie regras de firewall para evitar ou minimizar o impacto de ataques DoS |

...

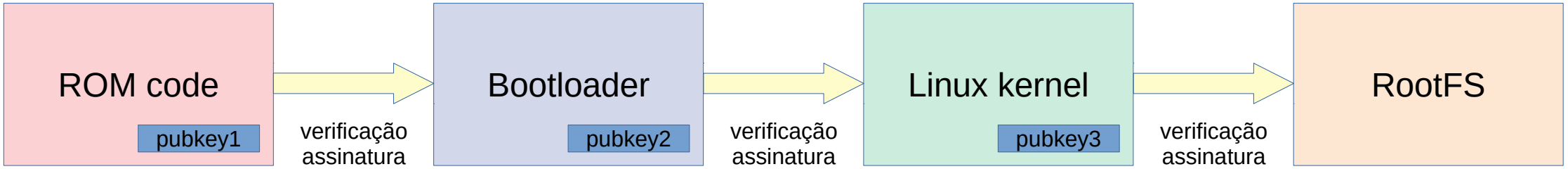
# CONCEITOS DE BOOT SEGURO

- O objetivo de um processo de boot seguro é proteger a integridade e a autenticidade do código.
- Por quê? Para garantir que os binários que você está executando foram criados por uma pessoa ou empresa confiável!
- Tem custos como gerenciamento de chaves, tempo de inicialização, torna mais difícil de desenvolver na plataforma, etc.

# COMO FUNCIONA?

- Baseado na verificação de assinaturas digitais (sem criptografia envolvida).
- A assinatura de cada componente do sistema deve ser verificada (bootloader, kernel, rootfs, etc).
- Isso significa que o primeiro elemento no processo de inicialização autentica o segundo, que autentica o terceiro, etc.
- Isso é chamado de cadeia de confiança (**chain-of-trust**).

# COMO FUNCIONA?



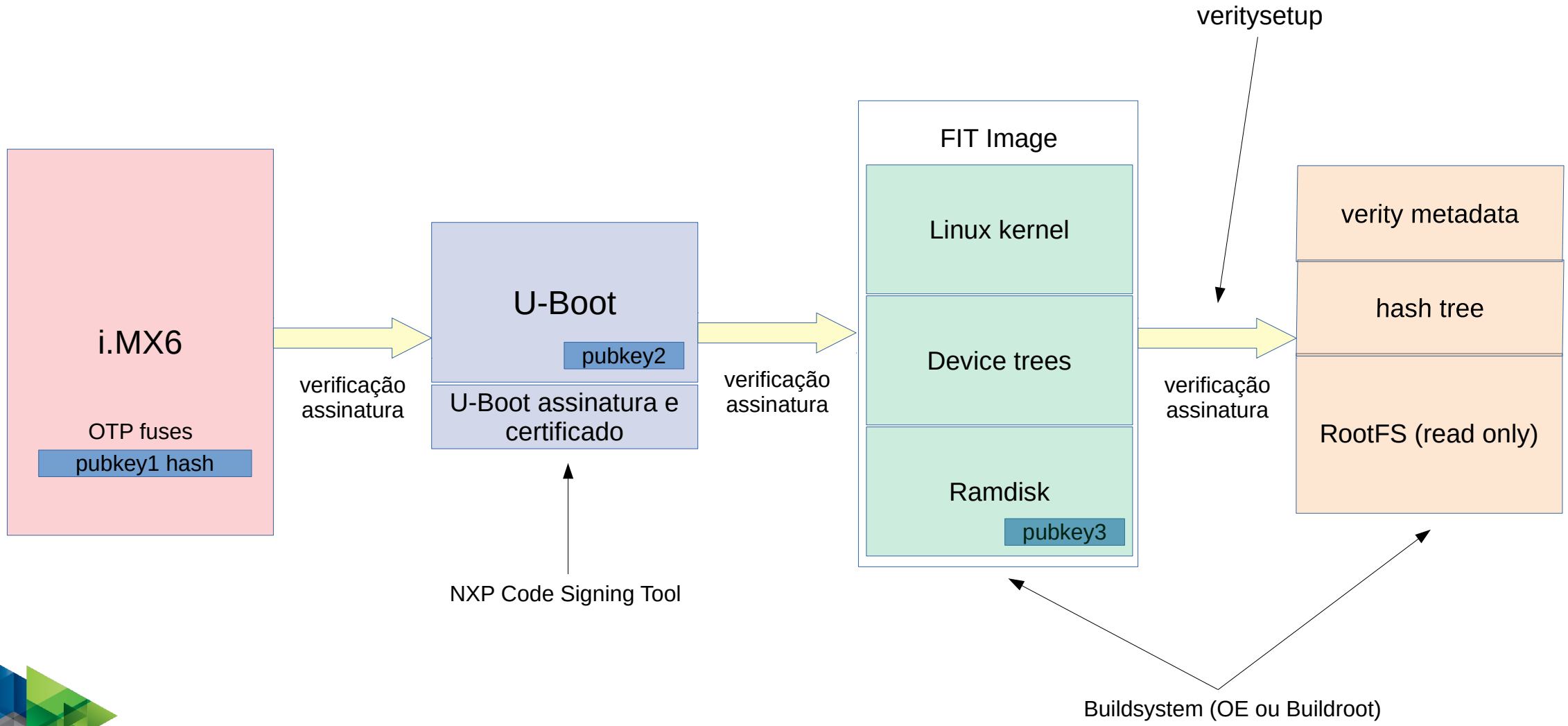
# COMO IMPLEMENTAR?

- Tudo começa no código em ROM dentro do SoC (**root of trust**).
- O código em ROM verificará a assinatura do bootloader.
  - Precisa armazenar a(s) chave(s) pública (por exemplo, **OTP fuses**).
  - Geralmente apenas o hash da chave pública é armazenado.
- O Bootloader (por exemplo U-Boot) verificará a integridade da **imagem FIT**.
  - A imagem FIT é um contêiner para vários binários com suporte a hash e assinatura.
  - Ele contém a imagem do kernel Linux, device tree(s) e a imagem do ramdisk.

# HOW TO IMPLEMENT IT?

- O ramdisk terá a lógica para verificar a assinatura do rootfs e montá-lo:
  - Para isso, é comum utilizar um módulo do kernel chamado **dm-verity**.
  - O device-mapper verity fornece verificação de integridade de dispositivos de bloco.
  - Requer um rootfs somente leitura (squashfs pode ser uma boa solução).
- A partição rootfs deve ser gerada com suporte dm-verity.
  - Outra abordagem seria IMA or dm-integrity para sistemas de arquivos read/write.
- Este é apenas um exemplo de implementação de boot seguro, embora possa ser aplicado a um conjunto diferente de placas e SoCs ARM.

# BOOT SEGURO NO i.MX6



# OOPS...

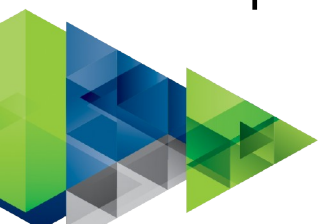
- Nada é 100% seguro!
- Vulnerabilidades na implementação de boot seguro no i.MX6, i.MX50, i.MX53, i.MX7, i.MX28 e Vybrid foram divulgados publicamente em 2017.

  https://blog.quarkslab.com/vulnerabilities-in-high-assurance-boot-of-nxp-imx-microprocessors.html

  https://community.nxp.com/docs/DOC-334996
- As vulnerabilidades foram corrigidas com um novo silício.

# ENCRIPTAÇÃO DE CÓDIGO E DADOS

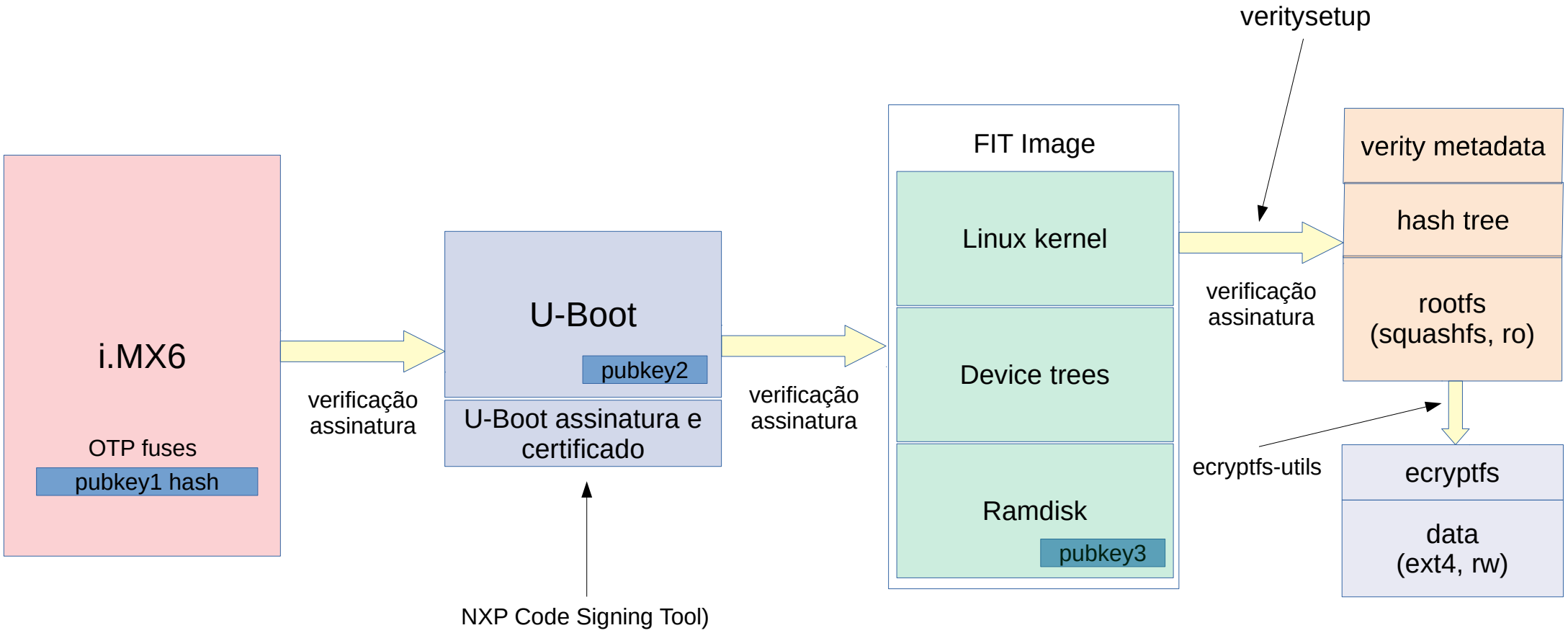
- Embora o boot seguro garanta autenticidade, ele não protege o dispositivo contra falsificação ou impede que agentes de ameaças extraiam código/dados do dispositivo.
- Se você deseja proteger sua propriedade intelectual e/ou garantir a confidencialidade dos dados, você precisará usar encriptação.
- Não é comum criptografar o código (rootfs) em um sistema Linux embarcado (mas você pode querer criptografar suas aplicações).
  - Tome cuidado com a GPLv3 (Tivoization).
- Por outro lado, a confidencialidade e proteção dos dados da aplicação ou do usuário pode ser um requisito, por exemplo, em dispositivos médicos.



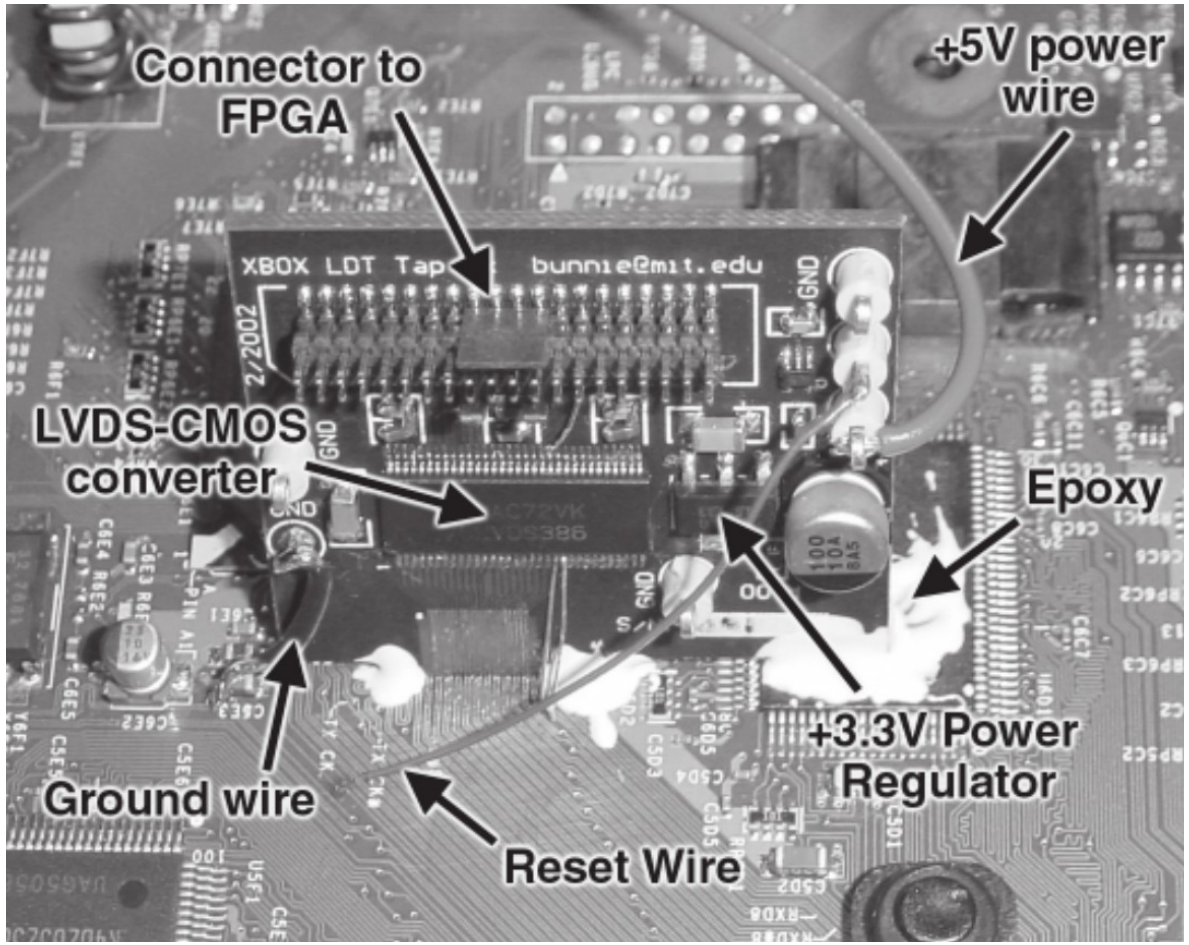
# ENCRIPTAÇÃO DE CÓDIGO E DADOS

- Existem basicamente duas abordagens principais para a encriptação de arquivos no Linux: encriptação do disco inteiro e encriptação de arquivos.
- A encriptação de disco utiliza uma criptografia no nível de bloco e todo o disco ou partição do disco é criptografado.
  - **dm-crypt** é um módulo do Linux kernel que implementa encriptação à nível de bloco.
- A encriptação de arquivo arquivo utiliza uma criptografia no nível do sistema de arquivos, onde cada diretório pode ser criptografado separadamente.
  - **fscrypt** é uma API disponível em alguns sistemas de arquivo (EXT4, UBIFS, F2FS).
  - **eCryptFS** é um sistema de arquivos com suporte à criptografia.

# BOOT SEGURO COM ENCRIPTAÇÃO



# ENCONTRE A CHAVE!




Fonte: https://bunniefoo.com/nostarch/HackingTheXbox_Free.pdf

# ARMAZENAMENTO DA CHAVE PRIVADA

- Um algoritmo de chave simétrica é geralmente usado para criptografia, então a chave privada deve estar armazenada em algum lugar do sistema para descriptografar os dados.
- E a proteção dos dados criptografados é tão segura quanto a proteção da chave para descriptografá-los!
- Em um desktop ou smartphone, a chave usada para criptografar o sistema de arquivos pode ser derivada de uma senha de usuário (passphrase) inserida interativamente.
- Em um sistema embarcado, ela deve ser armazenada criptografada no sistema de arquivos ou em um armazenamento seguro isolado do sistema.

# ARMAZENAMENTO DE CHAVE NO i.MX

- No i.MX, cada processador possui uma chave-mestre exclusiva (pré-programada pela NXP) que só pode ser acessada pelo módulo CAAM (Cryptographic Accelerator and Assurance Module).
- O módulo CAMM pode ser usado para criptografar a chave de criptografia do sistema de arquivos (isso teria que ser feito durante o processo de fabricação).
- A chave criptografada pode ser armazenada na partição de boot ou no rootfs.
- Durante a inicialização, o módulo CAMM seria usado para descriptografar e restaurar a chave de encriptação do sistema de arquivos.

# SECURE ELEMENT E TPM

- Se você não tiver recursos de segurança no processador, poderá obter os mesmos resultados com um hardware externo como um **Secure Element** ou um **TPM**.
- Esses dispositivos externos geralmente fornecem armazenamento seguro, de modo que podem ser usados para armazenar uma chave-mestra que pode ser usada para criptografar/descriptografar a chave de criptografia do sistema de arquivos.
- Esses dispositivos também oferecem muitos recursos de segurança, como geração de números aleatórios, cálculo de hash, criptografia e funções de assinatura, etc.
- **TEE** (Trusted Execution Environment) também pode ser usado para armazenar a chave com segurança (falaremos sobre TEE mais adiante nesta apresentação).

# SECURE ELEMENT

- Um Secure Element é um sistema computacional seguro.
- É basicamente um dispositivo de armazenamento seguro com seus próprios aplicativos seguros (geralmente implementados usando Java Card, mas não necessariamente).
- O que um secure element faz é muito aberto e depende da implementação, mas a maioria deles implementa o padrão **PKCS #11** (Public-Key Cryptography Standard 11).
- Exemplos de secure elements são Smart Cards e SIM cards.

# TPM

- TPM (Trusted Platform Module) é uma especificação e um padrão internacional (ISO/IEC 11889).
- O TPM não é um Secure Element, embora possa ser implementado dentro de um.
- Pode ser implementado em hardware ou software, mas a maioria das implementações é em hardware.
- Fornece um conjunto de recursos de segurança limitados e definidos pelo padrão, incluindo armazenamento seguro e funções criptográficas.

# PROGRAMAÇÃO SEGURA

- Você pode proteger seu código e dados com criptografia, mas se estiver executando uma aplicação com bugs que podem ser explorados, seus ativos ainda estão em risco!

- Se uma aplicação tiver vetores de ataque (entrada do usuário, arquivos de configuração, comunicações de rede, etc), um bug pode ser usado para explorar vulnerabilidades no software.

- Especialmente os programas escritos em linguagens mais inseguras como C/C++, bugs como **buffer overflow** podem ser usados em ataques como **stack smashing** e formatação de strings.

# CVE-2019-14835

- Uma falha de buffer overflow foi encontrada, nas versões **2.6.34 a 5.2.x** do kernel Linux, de forma que um usuário convidado com privilégios capaz de passar descritores com comprimento inválido para o host, pode usar esta falha para aumentar seus privilégios no host.

```
diff --git a/drivers/vhost/vhost.c b/drivers/vhost/vhost.c
index 34ea219936e3..acabf20b069e 100644
--- a/drivers/vhost/vhost.c
+++ b/drivers/vhost/vhost.c
@@ -2180,7 +2180,7 @@ static int get_indirect(struct vhost_virtqueue *vq,
                /* If this is an input descriptor, increment that count. */
                if (access == VHOST_ACCESS_WO) {
                        *in_num += ret;
-                       if (unlikely(log)) {
+                       if (unlikely(log && ret)) {
                                log[*log_num].addr = vhost64_to_cpu(vq, desc.addr);
                                log[*log_num].len = vhost32_to_cpu(vq, desc.len);
                                ++*log_num;
@@ -2321,7 +2321,7 @@ int vhost_get_vq_desc(struct vhost_virtqueue *vq,
                        /* If this is an input descriptor,
                         * increment that count. */
                        *in_num += ret;
-                       if (unlikely(log)) {
+                       if (unlikely(log && ret)) {
                                log[*log_num].addr = vhost64_to_cpu(vq, desc.addr);
                                log[*log_num].len = vhost32_to_cpu(vq, desc.len);
                                ++*log_num;
```

Fonte: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-14835

# ANÁLISE ESTÁTICA DE CÓDIGO

- As ferramentas de análise estática são capazes de analisar o código-fonte (sem executar o programa) para encontrar problemas antes que eles aconteçam.
- Essas ferramentas podem encontrar erros de programa como desreferências de ponteiro nulo, vazamentos de memória, estouro de inteiro, acesso fora dos limites de um buffer, uso antes da inicialização, etc!
- Existem muitas opções boas de código aberto (cppcheck, splint, clang, etc) e comerciais (Coverity, PC-Lint, etc) para análise de código estático.
- Sempre use ferramentas de análise estática para verificar seu código. E não ignore os avisos do compilador!

# PROTEÇÕES EM TEMPO DE EXECUÇÃO

- Desenvolva e teste as aplicações utilizando proteções em tempo de execução (ASLR, stack canaries, electric fence, ASan, etc).
- **ASLR** (Address Space Layout Randomization) é uma técnica de segurança que organiza aleatoriamente os endereços de memória de um processo (ou do kernel).
- **AddressSanitizer** (ASan) é uma ferramenta de instrumentação criada por pesquisadores de segurança do Google para identificar problemas de acesso à memória em programas C e C++.
- O **Valgrind** pode ajudar a detectar problemas relacionados à memória, como vazamentos e race conditions.

# FERRAMENTAS DE FUZZING

- Fuzzing ou teste de fuzz é uma técnica de teste de software automatizado que envolve o fornecimento de dados inválidos, inesperados ou aleatórios como entradas para um programa.
- O programa é então monitorado em busca de exceções, como travamentos, crashes, vazamentos de memória, etc.
- Diversas ferramentas de fuzzing de código aberto estão disponíveis, incluindo **AFL** (american fuzzing loop) e **syzkaller** (fuzzer do kernel Linux).
- Fuzzing é uma boa técnica para encontrar bugs e vulnerabilidades em aplicações.

# PERMISSÕES

- Uma forma de mitigar os riscos de um programa vulnerável é não executá-lo com privilégios de root (superusuário).
- Mas o problema é que às vezes precisamos de previlégios de root para executar alguma operação privilegiada (configurar o relógio do sistema, usar sockets RAW, etc).
- E então precisamos rodar nosso programa como root, certo? Errado!
- Temos algumas opções para um controle mais granular de permissões no Linux. E uma das soluções é chamada de **capabilities**.

# LINUX CAPABILITIES

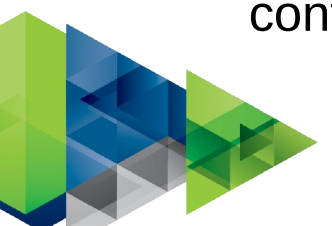
- O Linux divide os privilégios associados ao superusuário em unidades distintas, conhecidas como capabilities, que podem ser ativadas e desativadas independentemente.

- Portanto, a ideia é escrever um programa que seja executado como root, mas que habilite apenas as capabilities que precisa para fazer seu trabalho.

```
$ getcap /usr/bin/ping
/usr/bin/ping = cap_net_raw+ep
```

- Embora as capabilities forneçam um subconjunto dos privilégios de root disponíveis para um processo, eles não são muito flexíveis.

- Se você precisa de mais controle sobre as permissões, deve pensar em usar um tipo de controle de acesso chamado MAC (**Mandatory Access Control**).



# DAC vs MAC

- O Linux tradicionalmente oferece suporte à DAC (Discretionary Access Control).
- Em um sistema baseado em DAC, o acesso aos objetos é restrito com base na identidade dos sujeitos e/ou grupos a que pertencem (tradicionais flags "user", “group” and “other” no Linux).
- Em um sistema baseado em MAC, o sistema operacional restringe a capacidade de um sujeito de acessar ou executar algum tipo de operação em um objeto.
- O MAC é implementado no kernel via Linux Security Modules (LSM).

# LINUX SECURITY MODULES

- LSM é um framework que permite que o kernel Linux suporte uma variedade de modelos de segurança.
- Os mais famosos módulos de segurança que implementam MAC no Linux são AppArmor, SELinux, Smack e TOMOYO.
  - **SELinux** é uma das implementações MAC mais populares (e complexas), desenvolvida inicialmente pela NSA e hoje usada em projetos como Android e Fedora.
  - **AppArmor** também é uma implementação MAC popular (e mais amigável), com suporte da Canonical e usada em algumas distribuições Linux como Ubuntu e Debian.
- Caso precise de um controle mais refinado sobre as permissões de processos no Linux, avalie a possibilidade de adotar um modelo de segurança baseado em MAC.

# SANDBOXING

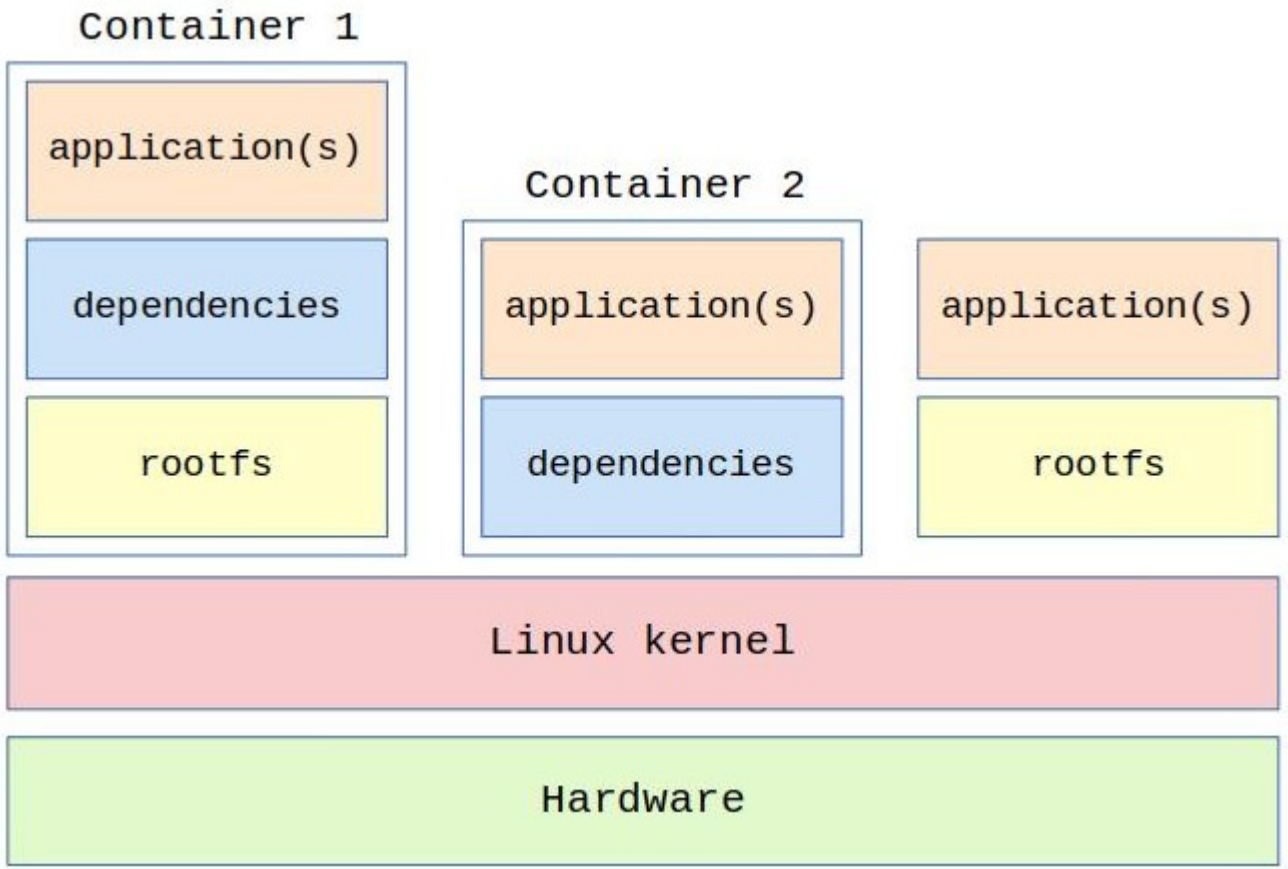
- Às vezes, restringir as permissões não é suficiente para proteger o sistema de uma aplicação vulnerável. E para melhorar a segurança, sandboxing pode ser utilizado para isolar os aplicativos do restante do sistema.
- Possivelmente, a ferramenta de sandboxing mais antiga disponível no Linux é o **chroot**, que não é muito útil em termos de segurança porque isola apenas o sistema de arquivos.
- Virtualização é outra forma de sandboxing, mas os custos são muito altos em termos de consumo de recursos, especialmente em sistemas embarcados.
- Atualmente, duas possíveis soluções para sandboxing em Linux embarcado são Containers e Trusted Execution Environment (TEE).

# LINUX CONTAINERS

- Um contêiner Linux é um sistema de arquivos mínimo com apenas os componentes de software necessários para executar um aplicativo específico ou grupo de aplicativos.

- Usando alguns recursos do kernel, o contêiner será "executado" de forma completamente isolada do restante do sistema (apenas o kernel é compartilhado).
  - **namespaces** permitem isolar a execução de um processo no Linux (PID, usuários, conexões de rede, pontos de montagem, etc).
  - **cgroups** permitem particionar recursos do sistema (CPU, memória, I/O) por processo ou grupo de processos.
  - **seccomp** permite limitar as chamadas de sistema que um processo pode fazer.

# LINUX CONTAINERS



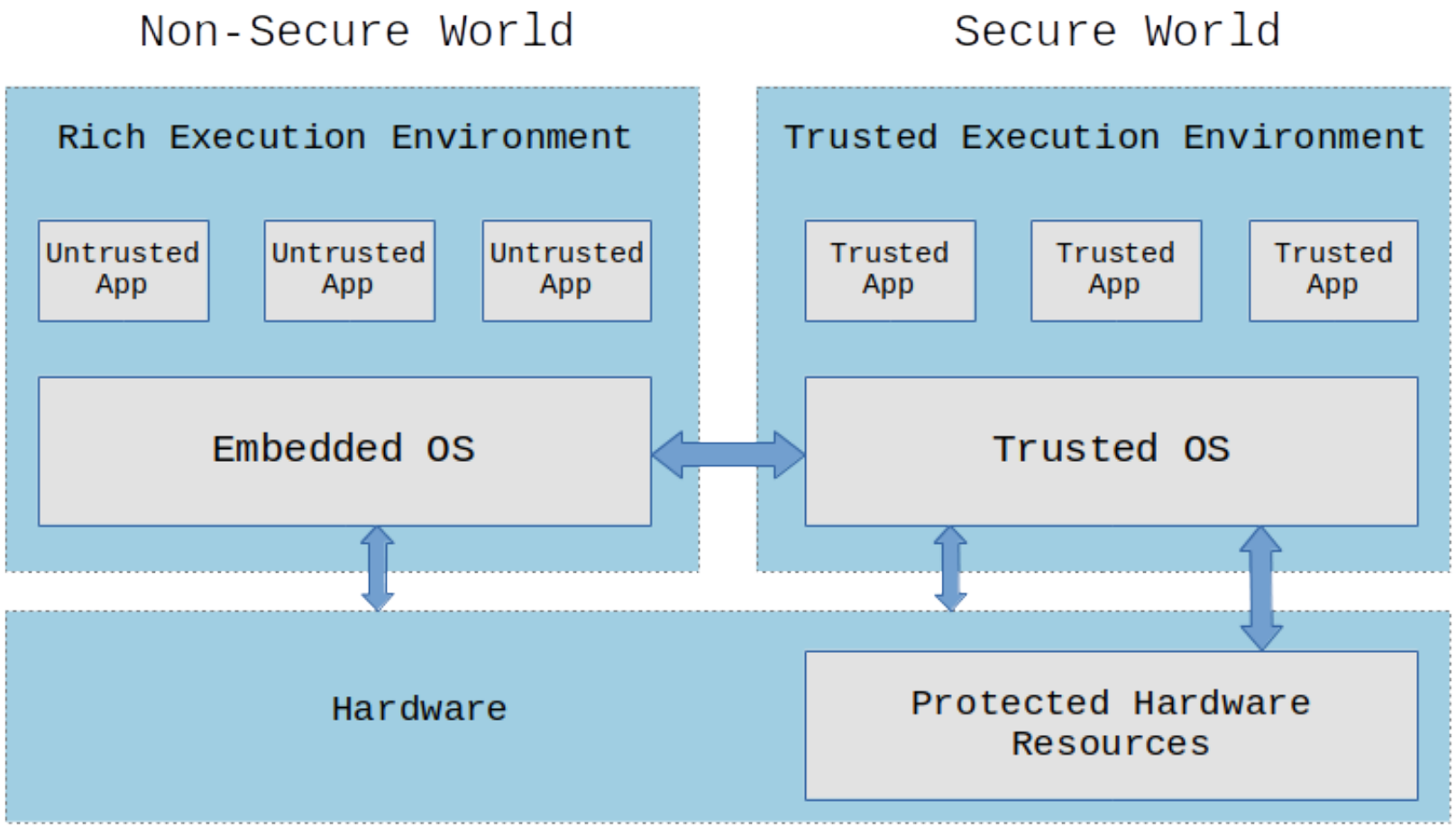
# TEE

- Em um sistema baseado em containers, se o kernel for comprometido, todo o sistema operacional estará em risco. Um ambiente de execução confiável (TEE) pode evitar isso.
- TEE é um ambiente onde o código executado e os dados acessados são isolados e protegidos em termos de confidencialidade (ninguém tem acesso aos dados) e integridade (ninguém pode alterar o código e seu comportamento).
- Muitos dispositivos ao nosso redor usam TEE, incluindo smartphones, decodificadores, consoles de videogame e TVs inteligentes.
- TEE pode ser uma boa solução para armazenar e gerenciar chaves de criptografia, armazenar e gerenciar credenciais e dados confidenciais e proteger informações digitais com direitos autorais.

# TEE

- Em um sistema com TEE, temos Untrusted Applications (UAs) em execução em um Rich Execution Environment (REE) e Trusted Applications (TAs) em execução em um Trusted Execution Environment (TEE).
- Somente aplicativos confiáveis executados em um TEE (Secure World) têm acesso completo ao processador principal, periféricos e memória.
- Um isolamento de hardware (provido pelo SoC) protege os TAs de aplicativos não confiáveis em execução no sistema operacional.
- Processadores modernos possuem esta funcionalidade (e.g. ARM's TrustZone, RISC-V's MultiZone, Intel SGX).

# TEE


Non-Secure World
Secure World
Rich Execution Environment
Trusted Execution Environment
Untrusted App
Untrusted App
Untrusted App
Trusted App
Trusted App
Trusted App
Embedded OS
Trusted OS
Hardware
Protected Hardware Resources

# ATUALIZAÇÃO DO SISTEMA E SEGURANÇA

- Apesar de todas as mitigações que vimos até agora, um sistema operacional com milhões de linhas de código certamente terá bugs e vulnerabilidades!
- Ter um sistema de atualização (remota) é muito importante, especialmente para dispositivos conectados.
- O sistema de atualização deve ser projetado nos estágios iniciais do desenvolvimento do produto.
- O que é mais caro: investir tempo para implementar um bom sistema de atualização ou fazer o recall de todas as unidades para corrigir um bug no software do seu produto?

# DESAFIOS

- Segurança (autenticidade, confidencialidade).
- Integridade.
- À prova de falhas contra perda de energia (atomic).
- Uso da largura de banda.
- Velocidade da atualização e tempo de inatividade.
- Recurso para reverter a atualização (rollback).

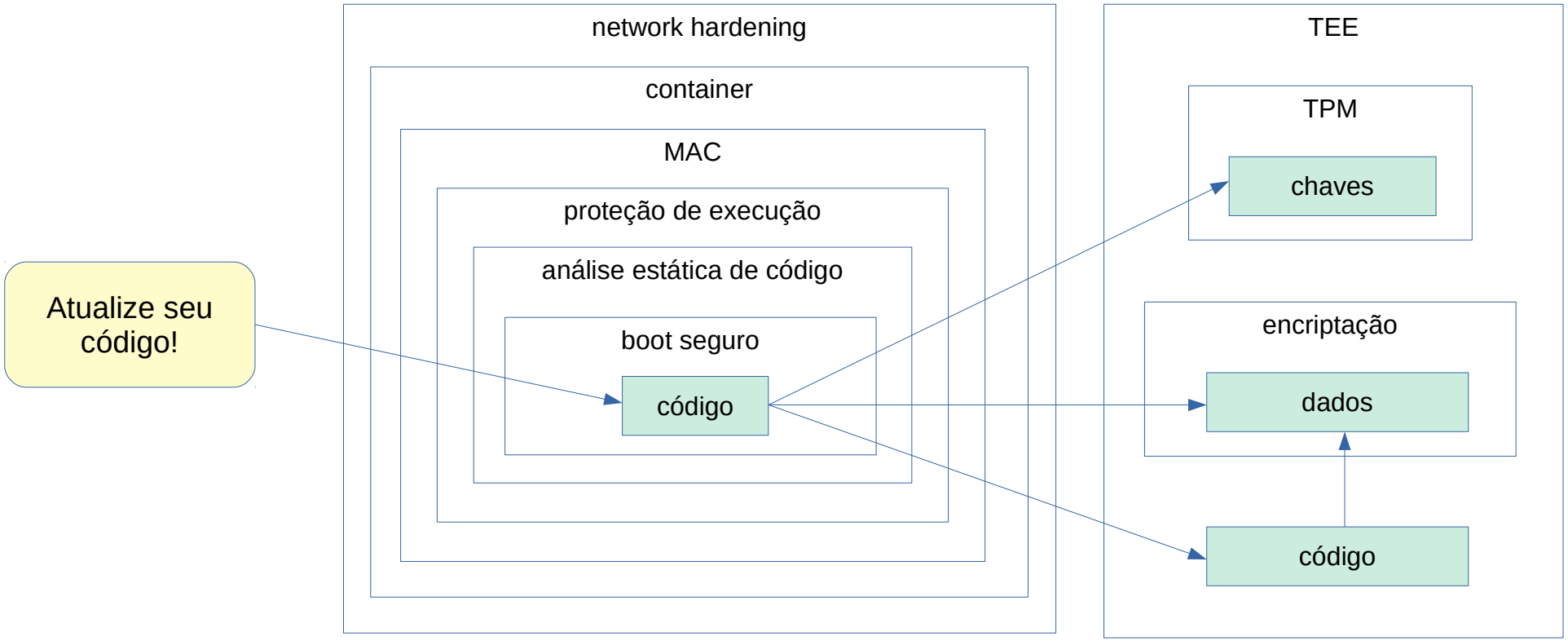
# SEGURANÇA NA COMUNICAÇÃO EM REDE

- Se você estiver fazendo atualizações remotas (OTA), seu dispositivo tem uma conexão de rede (Wi-Fi, Ethernet, etc).
- E se o seu dispositivo tiver uma conexão de rede, você deve se preocupar com a segurança da rede!
- A primeira regra é diminuir a superfície de ataque. Por exemplo:
  - Feche todas as portas TCP/UDP não utilizadas.
  - Desative todos os protocolos não usados (por exemplo, IPv6, PPP, etc).

# NETWORK SECURITY

- Em seguida, torne mais difícil a invasão no seu dispositivo! Por exemplo:
  - Crie regras de firewall (evite conexões de entrada/saída, proteja contra ataques DoS, evite varredura de portas, etc).
  - Use um IDS de rede como o snort para proteção/detecção de intrusão.
  - Comunique-se com dispositivos externos usando uma conexão segura (VPN, SSH reverso, TLS, HTTPS, etc).
  - Configure um limite de frequência de logins (rate limit) para serviços de conexão remota (ssh, web, etc), e assim evitar ataques de força bruta.

# DEFESA EM CAMADAS



# “REGRAS GERAIS” DE SEGURANÇA

- Defesa em profundidade: tenha sempre mais de uma camada ou tipo de defesa.
- A segurança envolve todos os níveis do sistema.
- Utilize sempre o princípio do mínimo privilégio: não dê mais privilégios do que o absolutamente necessário para fazer o trabalho exigido.
- Ofuscação ou "obscuridade" simplesmente não funcionam!
- Não existe um sistema 100% seguro.
  - Esteja ciente de que um invasor precisa apenas encontrar um único problema!

# PROJETE COM SEGURANÇA EM MENTE

- Projete com segurança em mente e esteja ciente dos trade-offs (um sistema deve ser “seguro o suficiente”).
- Identifique ativos, ameaças, vetores de ataque e reduza os riscos.
- Siga as boas práticas de segurança, conheça as técnicas e ferramentas disponíveis e use-as quando necessário.
- Tenha um bom sistema de atualização, monitore as vulnerabilidades do software (CVEs) e corrija/atualize quando necessário.

# Q&A

Sergio Prado
sergio@embeddedbits.org

https://twitter.com/sergioprado
https://www.linkedin.com/in/sprado

Obrigado!